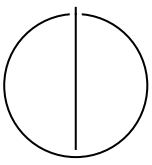
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Space parallelism for shallow water equations on the rotating sphere

Ahmad Traboulsi



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

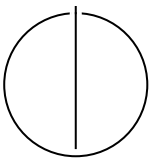
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Space parallelism for shallow water equations on the rotating sphere

Space Parallelisierung für die Flachwassergleichungen auf der rotierenden Kugel

Author:	Ahmad Traboulsi
Supervisor:	Prof. Dr. Hans-Joachim Bungartz, Prof. Martin Schreiber
Advisor:	Keerthi Gaddameedi
Submission Date:	15.11.2025



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.11.2025 Ahmad Traboulsi

Acknowledgments

The authors gratefully acknowledge the computational and data resources provided by the Leibniz Supercomputing Centre (www.lrz.de).

Abstract

Shallow Water Equations (SWE) are a simplified form of the 3D Navier-Stokes equations, reduced to two dimensions under the assumption of a shallow fluid layer. The SWEET (Shallow Water Equations Environment for Tests) framework is a high-performance numerical solver used for atmospheric modeling and geophysical fluid dynamics, utilizing spectral methods based on spherical harmonics. As a lightweight yet extensible testbed, SWEET enables the exploration of temporal and spatial discretization strategies that can inform the development and optimization of full 3D atmospheric dynamical cores. It supports various time integration methods, including several parallel-in-time techniques such as Parareal, thereby enabling temporal parallelism.

In both SWEET and production-level dynamical cores, spatial discretization is crucial for accurately representing fluid motion globally. However, as spatial resolution increases, the associated computational cost grows significantly, often becoming a bottleneck in achieving timely simulation results. Currently, spatial discretization in SWEET benefits from shared-memory parallelism only. This thesis extends SWEET by introducing an MPI-based spatial domain decomposition, which enables concurrent computations across multiple ranks and aims to achieve full space–time parallelism.

The resulting hybrid MPI–OpenMP solver was evaluated using the Galewsky benchmark. Accuracy was maintained across all configurations, confirming numerical consistency. However, performance profiling revealed that global spectral transforms (via the SHTns library) dominate total runtime and cannot be distributed across MPI ranks. Consequently, each rank redundantly performs identical transform computations on full grid copies, while the MPI Allgather communication required to synchronize global fields introduces substantial synchronization and waiting time. As a result, the hybrid configurations do not outperform the OpenMP-only baseline, which achieves the best scaling efficiency within a node.

These findings highlight the algorithmic limitations of global spectral methods for distributed memory systems and underscore the need for distributed spectral transforms to enhance performance and facilitate the implementation of an efficient communication workflow in future work.

Contents

1 Introduction

In weather models and climate research, dynamical cores are the fundamental numerical engines used for capturing the atmospheric behavior of Earth [27]. They integrate a chosen set of governing equations on the sphere forward in time to advance the prognostic state (e.g., atmospheric components such as temperature and pressure) at a given spatial resolution and time step [5]. Therefore, weather and ocean models need to adopt dynamical cores that are accurate, stable, and efficient.

Around the core, physical parameterizations represent other processes that are either subgrid-scale at the chosen grid resolution or are too costly to resolve explicitly at each step. Phenomena such as radiation, cloud microphysics and precipitation, or surface fluxes are delegated to other modules that calculate the tendencies of those specific processes and feed them back into the dynamical core [5]. The resulting physics-dynamics coupling advances the model state fully at each time step.

The design of a dynamical core comprises three intertwined choices: (i) the equation set and the prognostic variables (ii) the spatial representation (iii) the time integration scheme. Specifically, Ullrich et al. in [22] provide a comprehensive review of 11 non-hydrostatic dynamical cores from DCMIP2016, organizing the intercomparison by the aforementioned design criteria.

(i) Equation set and variables: Options include shallow-water equations, hydrostatic primitive equations, and fully compressible non-hydrostatic equations. Typical prognostic variables include wind components, pressure, and temperature. (ii) Spatial representations: Grids such as latitude-longitude grids, cubed spheres, or icosahedral-shaped topologies, and discretization families such as the spectral transform or the finite element/volume method. Trade-offs to keep in mind involve accuracy and conservation, pole handling, communication patterns, and scalability. (iii) Time integration: Schemes such as explicit, semi-implicit, semi-Lagrangian, and parallel-in-time set the cost per step and stability limits.

Different combinations of those three dimensions can yield functional models. The criteria to decide on what to adopt depend on the hardware available, the target scales we want to operate on, and the coupling requirements (such as the previously mentioned physics-dynamics coupling).

SWEET (Shallow Water Equations Environment for Tests) [19] is a lightweight, high-performance framework developed to investigate time integration techniques and

parallelization strategies for PDEs that can be addressed using global spectral methods such as Fourier and Spherical harmonics. In other words, it is built to explore the previously stated design choices. It specifically focuses on the rapid prototyping of time integrators, exploring novel methods to achieve parallelization, and conducting preliminary research on the scalability of certain time integration techniques.

While SWEET already exposes temporal parallelism, its spatial discretization has remained serial (shared-memory parallelism strictly), limiting end-to-end speedups and preventing full space-time concurrency. In this paper, we adopt the Shallow-Water Equations along with the vorticity-divergence formulation within SWEET. Building on top of SWEET's provided framework, we introduce an MPI-based spatial domain decomposition in space so that spatial computation can run concurrently within the time-stepping schemes, to achieve space-time parallelism.

2 Background Theory and Related Work

2.1 Navier-Stokes Equations

The Navier-Stokes equations (NSE) are a set of nonlinear partial differential equations that describe fluid flow [12]. For the equations below and throughout this work, we assume three characteristics of the fluid in question. First, the fluid must be Newtonian, meaning that the viscosity of the fluid must not change if you change the shear rate it is subjected to. Second, the fluid is incompressible, meaning its volume remains constant following the motion. And lastly, the fluid must be isothermal, meaning that heat remains constant as the fluid flows. If these assumptions are met, then we can describe fluid flow by the continuity and momentum equations [21]. The continuity equation,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.1}$$

where it expresses mass conservation; any change to the density is exactly balanced by the mass of the fluid flowing out of or into a point. ρ is the fluid density, $\mathbf{u}$ is the velocity vector, ∇ is the nabla operator (denoting divergence when dotted with a vector), and $\frac{\partial \rho}{\partial t}$ represents the local rate of change of density.

If we assume the fluid is incompressible, this means that its density, ρ, remains constant, as variations in pressure or temperature do not significantly affect the density. Taking that into account, equation (2.1) reduces to equation (2.2) [10],

$$\nabla \cdot \mathbf{u} = 0 \tag{2.2}$$

We also have the momentum equation,

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2.3}$$

Where:

- $\mathbf{u}$ = velocity vector field (in 3D)
- ∇ = gradient operator

- ρ = fluid density
- t = time
- p = pressure
- μ = viscosity
- ∇^2 = Laplacian
- $\mathbf{f}$ = external body forces per unit volume

Equation (2.3) represents what is essentially Newton's second law ($\mathbf{F} = m\mathbf{a}$) applied to fluid motion. The left-hand side captures the total acceleration of the fluid. On the right-hand side, the first term, $-\nabla p$, represents the pressure gradient force, which drives the motion of the fluid from regions of high pressure to low pressure. The second term, $\mu\nabla^2\mathbf{u}$, accounts for the viscous force due to the internal friction of the fluid's particles, resulting in motion resistance between fluid layers. The last term accounts for external forces exerted on the fluid, ranging from gravitational and magnetic forces to Coriolis forces arising from the rotating reference frames, which will be discussed more in this paper.

While there is no universal analytical solution to the NSE (remains an open Millennium Prize Problem) [3], numerical methods in the field of computational fluid dynamics are used to approximate the solution of the above equations in order to simulate fluid flow.

2.2 Shallow-Water Equations

The Shallow-Water Equations (SWE) are simplified equations derived from the conservation of mass and conservation of linear momentum (Navier–Stokes equations), along with additional physical assumptions. SWE is the system used for simulating fluid flow on a rotating sphere, and in more grounded terms, it allows us to approximate parts of the atmosphere on Earth.

The key assumption required for the valid usage of SWE is that the fluid in question is shallow [23, 24]. In other terms, the horizontal length is much larger than the vertical depth, so much so that the vertical acceleration is consequently considered negligible, and therefore, we focus on the effects on the height and horizontal velocities over time, and the vertical dimension is dropped. This small aspect ratio enables us to neglect vertical acceleration (the hydrostatic balance condition) and assume that horizontal velocities are nearly uniform with depth, thereby reducing the system to two dimensions. This allows for significantly less computational complexity and cost compared

to modeling the atmospheric elements using NSE, due to the relative simplicity of working with a 2D system, which makes SWE more favorable for oceanography and atmospheric modeling applications.

Therefore, we derive the system of SWE through depth-integrating the NSE system. Due to the hydrostatic balance assumption, the vertical component of the momentum equation be simplified to describe how pressure changes with depth. Under this assumption, we can now reduce the vertical momentum equation to equation (2.4) [13],

$$\frac{\partial p}{\partial z} = -\rho g \tag{2.4}$$

where

- p: Pressure at depth z
- g: Gravitational acceleration

In this formulation, z is taken to increase upward, and gauge pressure ($p_{\text{atm}} = 0$) is used for simplicity. Gauge pressure refers to the pressure measured relative to the surrounding atmospheric pressure, rather than as an absolute value. Essentially, this equation indicates that pressure decreases with height in a stationary fluid because each layer in the fluid must support the weight of the layers above it.

If we integrate equation (2.4) over z, we can get the equation to calculate pressure at a certain depth with respect to the total height of the water column h,

$$p(z) = \rho g(h - z) \tag{2.5}$$

Building on this relationship, we can now describe the overall motion of the fluid using the two-dimensional shallow water equations (SWE). These equations are obtained by depth-integrating NSE (according to the aforementioned assumptions). Starting with the continuity equation (2.6) [24],

$$\frac{\partial h}{\partial t} + \frac{\partial (uh)}{\partial x} + \frac{\partial (vh)}{\partial y} = 0 \tag{2.6}$$

and the two horizontal momentum equations,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -g\frac{\partial h}{\partial x} + fv \tag{2.7}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -g\frac{\partial h}{\partial y} - fu \tag{2.8}$$

where:

- $h(x, y, t)$: fluid layer thickness (free-surface height above the bottom)
- $u(x, y, t), v(x, y, t)$: depth-averaged horizontal velocity components in x and y directions
- $f = 2\Omega \sin \phi$: Coriolis parameter, with Ω the Earth's rotation rate and ϕ the latitude
- t : time

As a final note, although vertical acceleration is neglected, the vertical velocity does not necessarily equate to zero and can be diagnosed from the continuity equations.

2.2.1 Reformulation of SWE for Spherical Operations

While the shallow water equations in their primitive form are suitable for many applications, global atmospheric and oceanic models often require a formulation adapted to the spherical geometry of the Earth. For computational efficiency, especially in spectral transform methods and spherical harmonics, which we will explore in later sections, it is common to reformulate the equations in terms of other relevant variables.

The so-called vorticity–divergence form is computationally advantageous on the rotating sphere because the prognostic variables are scalars that can be expanded efficiently in spherical harmonics [2, 8]. In this formulation, the equations are written for the time evolution of geopotential, Φ, the vorticity, ζ, and the divergence, δ, replacing the previous horizontal velocity components (u,v).

With that, we can now rewrite the previous SWE system into equations (2.9) and (2.10) [2, 8]:

$$\frac{d\Phi}{dt} = -\Phi \nabla \cdot \mathbf{V} \tag{2.9}$$

$$\frac{d\mathbf{V}}{dt} = -f\mathbf{k} \times \mathbf{V} - \nabla\Phi \tag{2.10}$$

The system is now written compactly in vector form and adapted to the sphere, where the previous velocity coordinates are now jointly expressed through the horizontal velocity vector $\mathbf{V} = iu + jv$, where i and j are the unit vectors in the eastward and northward directions, respectively.

Equation (2.9) includes the geopotential Φ,

$$\Phi = gh$$

It represents the amount of work required to raise the parcel from sea level to a given height against gravity. In simpler terms, it describes the energy level associated with elevation. When expressed as geopotential height, it allows us to compare pressure surfaces in the atmosphere or ocean in terms of an equivalent height above sea level. Mathematically, it is the product of the gravitational acceleration g and the height above a reference level h.

The equations also include the Coriolis parameter f [9], defined as

$$f = 2\Omega \sin \phi$$

where Ω is Earth's rotation rate and ϕ is latitude. It represents the effect of Earth's rotation on moving fluids, deflecting them to the right in the Northern Hemisphere and to the left in the Southern Hemisphere.

Lastly, k represents the unit vector pointing vertically upward (perpendicular to the Earth's surface). It's used in the cross product $k \times V$, which generates a horizontal vector perpendicular to the velocity, i.e., the Coriolis acceleration.

Now, to proceed toward the vorticity–divergence form, we define the two scalar quantities:

$$\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \tag{2.11}$$

Equation (2.11) calculates the relative vorticity [9]. Vorticity is a measure of the local spin or rotation of the flow. In two dimensions, it describes how strongly air or water parcels tend to rotate around a vertical axis. Mathematically, it is defined from the velocity field as the difference between the rate of change of the meridional (north-south) and zonal (east-west) wind components.

$$\delta = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{2.12}$$

Equation (2.12) captures the divergence [9]. Divergence is a measure of how much the flow spreads out or comes together at a given point. In two dimensions, it indicates whether air or water parcels are moving away from each other (divergence) or converging toward each other (convergence).

Equations (2.11) and (2.12) are expressed in Cartesian coordinates to introduce the variables in a simple manner; the more general coordinate-independent definitions of vorticity and divergence that fit any curvilinear system (including a Sphere, which is more appropriate for Earth models) take the form of equations (2.13) and (2.14) [9]:

$$\zeta = \mathbf{k} \cdot (\nabla \times \mathbf{V}) \tag{2.13}$$

$$\delta = \nabla \cdot \mathbf{V} \tag{2.14}$$

On the sphere, ∇, $\nabla\cdot$, $\nabla\times$, and ∇^2 denote the surface gradient, divergence, curl, and Laplacian (Laplace–Beltrami) operator respectively [4].

As a quick overview, the gradient operator ∇, as we have seen before, measures how a scalar field changes across the spherical surface. The divergence operator $\nabla\cdot$ quantifies how much a vector field spreads out from a point, while the curl operator $\nabla\times$ measures its tendency to rotate. Finally, the Laplacian ∇^2 combines these effects to describe how a quantity smooths out over the surface.

The previously stated variables (Φ, ζ, δ) will replace the velocity components in the first version of SWE. To reconstruct the velocity field from these scalars, we invoke the Helmholtz decomposition [4]. This theorem states that any horizontal velocity field can be separated into a non-divergent rotational part and a divergent irrotational part,

$$\mathbf{V} \;=\; \mathbf{k} \times \nabla\psi \;+\; \nabla\chi, \tag{2.15}$$

where ψ is the streamfunction and χ is the velocity potential. The term $\mathbf{k} \times \nabla\psi$ represents the non-divergent rotational part, which means that this component represents circular motion with zero divergence (no net inflow or outflow). The other term, $\nabla\chi$, represents the divergence irrotational part. which describes converging or spreading flow without rotation.

Applying the curl and divergence operators to this expression yields the following relationships (2.16) [4]:

$$\zeta \;=\; \nabla^2\psi, \qquad \delta \;=\; \nabla^2\chi. \tag{2.16}$$

Through this, we can show that the horizontal velocity components can be recovered from their scalar prognostic counterparts through the relationships in (2.16).

Finally, by applying the curl and divergence operators to the momentum equation (2.10), we obtain equations (2.17) and (2.18) [8],

$$\frac{\partial\zeta}{\partial t} \;=\; -\nabla\cdot\Big[(\zeta + f)\,\mathbf{V}\Big] \tag{2.17}$$

$$\frac{\partial\delta}{\partial t} \;=\; \mathbf{k}\cdot\nabla\times\Big[(\zeta + f)\,\mathbf{V}\Big] \;-\; \nabla^2\left(\Phi \;+\; \frac{1}{2}\,|\mathbf{V}|^2\right) \tag{2.18}$$

The curl operator $\nabla\times$ extracts the rotational part of the momentum equation (2.10), which gives us the vorticity tendency equation found in equation (2.17).

The divergence operator $\nabla\cdot$ extracts the convergence or divergence part of the same equation, which also gives us the evolution of the divergence (2.18).

Adding to this system the equation already derived in (2.9), we obtain the formed vorticity-divergence system used in weather simulation.

2.3 Spatial Discretization

In the previous chapter, we introduced the SWE as evolution equations describing fluid flow for continuous fields on the sphere. However, given that there are generally no closed-form solutions for realistic flows [20], we need to provide a finite representation of those fields in space so computers can solve them numerically. A common representation of those values on the sphere can be the traditional latitude-longitude grid, which is widely used in many atmospheric models [5] and partially used in SWEET (e.g., to evaluate nonlinear products efficiently). Given that the prognostic variables (geopotential Φ, vorticity ζ, and divergence δ) we dealt with in the vorticity-divergence SWE form are scalars on the sphere, coupled with the repeated usage of linear operators such as the surface Laplacian operator ∇^2 and its inverse throughout the related equations, the spectral-transform method with spherical harmonics presents a natural fit as a method to discretize space [5]. In the following sections, we will outline its underlying theory, how it is applied in the case of our vorticity-divergence system, and the reason behind utilizing it in SWEET.

2.3.1 Spherical Harmonics

Spherical harmonics (denoted as Y_n^m, with degree n and order $m = -n, ..., n$) are a set of special mathematical functions that form a complete, orthonormal basis for square-integrable scalar functions [1]. In simpler terms, any scalar field on a spherical surface can be represented as a weighted sum of spherical harmonics. Thus, we can represent a scalar field f that takes spherical coordinates, θ representing the co-latitude (angle from the North Pole; $\theta \in [0, \pi]$), and λ representing the longitude (east-west angle; $\lambda \in [0, 2\pi)$), noting that the colatitude can be converted to latitude φ through $\theta = \frac{\pi}{2} - \varphi$.

$$f(\theta, \lambda) = \sum_{n=0}^{N} \sum_{m=-n}^{n} f_n^m \, Y_n^m(\theta, \lambda) \tag{2.19}$$

In the above equation [1, 11], we have a nested summation of degree n that controls the total wavenumber. As n increases, we obtain finer features on the latitude-longitude scale, and the order m controls how many waves wrap around longitude. f_n^m represents the spectral coefficients that act as weights for the spherical harmonic ripple Y_n^m. In their concrete form, Y_n^m [7] is written as,

$$Y_n^m(\theta, \lambda) = P_n^m(\mu)\, e^{im\lambda}, \mu = \cos\theta = \sin\varphi \tag{2.20}$$

Similar to how any periodic function can be written as a sum of a Fourier mode on a one-dimensional circle, a spherical harmonic basis function $P_n^m e^{im\lambda}$ is the natural analog of that on a sphere, where each pair (n, m) represents one mode. Equation (2.20) describes it as the product of the (normalized) associated Legendre polynomial $P_n^m(x)$ and the azimuthal wave $e^{im\lambda}$ with m cycles around the longitude.

Through substituting (2.20) in (2.19), we can reach the Legendre-Fourier form similar to the one written in equation (2.21) [8],

$$\xi(\lambda, \mu) = \sum_{m=-M}^{M} \sum_{n=|m|}^{N(m)} \xi_n^m\, P_n^m(\mu)\, e^{im\lambda}, \tag{2.21}$$

where ξ_n^m are the complex spectral coefficients and the P_n^m are the normalized associated Legendre functions.

As we have specified in the beginning, spherical harmonics are orthonormal functions. Using this principle, we can calculate the spectral coefficients used in the equation (2.21) through [5]

$$\xi_{nm} = \int_{-1}^{1} \frac{1}{2\pi} \int_{0}^{2\pi} \xi(\lambda, \mu)\, e^{-im\lambda} P_n^m(\mu)\, d\lambda\, d\mu \tag{2.22}$$

It is worth noting that in the above equations, we are using triangular truncation T_N where only the modes that satisfy $\{(n, m) : 0 \leq n \leq N, -n \leq m \leq n\}$ are valid [5]. This is to disambiguate its usage from other truncations such as rhomboidal or trapezoidal.

2.3.2 Operations in Spectral Space

Following the explanation in the previous section, we define the spectral space as the finite set of spherical-harmonic coefficients f_n^m. This space represents the counterpart of the physical grid space used for storing values at latitude-longitude points. An important property of spherical harmonics is that they are the eigenfunctions of key surface operators on the sphere [1]. As a result, linear operators in the spectral space act like a diagonal matrix; each mode is scaled on its own, with no cross-terms or coupling between modes. For example, the surface (Laplace–Beltrami) operator (on a sphere of radius a) we used in 2.2.1 becomes equation (2.23) [7]:

$$\nabla^2 Y_n^m = -\frac{n(n+1)}{a^2} Y_n^m \tag{2.23}$$

This means that applying the surface Laplacian, we just scale the same ripple by a certain factor ($-\frac{n(n+1)}{a^2}$), with no mixing with other ripples.

Therefore, applying ∇^2 to an expansion,

$$f(\theta,\lambda) = \sum_{n=0}^{N} \sum_{m=-n}^{n} f_n^m Y_n^m(\theta,\lambda) \tag{2.24}$$

yields equation (2.25) [25],

$$(\nabla^2 f)_n^m = \frac{-n(n+1)}{a^2} f_n^m \tag{2.25}$$

Note that the usage of the previously mentioned triangular truncation limits the diagonal system to even fewer modes than if it were without truncation (the eigenfunction property holds regardless).

In addition to the surface Laplacian, the Poisson inversions that we use to recover the stream function ψ and χ after advancing the prognostic scalars ζ and δ, also benefit from the eigenfunction property as shown in equations (2.26) and (2.27) [8],

$$\psi_{nm} = -\frac{a^2}{n(n+1)} \zeta_{nm}, n \geq 1 \tag{2.26}$$

$$\chi_{nm} = -\frac{a^2}{n(n+1)} \delta_{nm}, n \geq 1 \tag{2.27}$$

Other diagonalized operators include the longitude derivative (2.28) [14]:

$$\partial_\lambda Y_{nm} = i\, m\, Y_{nm} \;\Rightarrow\; (\partial_\lambda f)_{nm} = i\, m\, f_{nm} \tag{2.28}$$

Inserting the above operator into the previously mentioned Helmholtz decomposition (Equation 2.15) in 2.2.1 yields equations (2.29) and (2.30) [13],

$$u = \frac{1}{a\cos\varphi}\, \partial_\lambda \chi - \frac{1}{a}\, \partial_\varphi \psi \tag{2.29}$$

$$v = \frac{1}{a}\, \partial_\varphi \chi + \frac{1}{a\cos\varphi}\, \partial_\lambda \psi \tag{2.30}$$

Note that while the zonal derivative ∂_λ is done spectrally as stated in Equation 2.28, the meridional derivative ∂_φ is not diagonal in spherical harmonics. In practice, we inverse-transform ψ and χ to a Gaussian latitude-longitude grid and evaluate the ∂_φ there (e.g., using finite differences). Equivalently, it can be calculated via stable associated-Legendre recurrences, but in SWEET and therefore this thesis, inverse transforms are adopted.

In short, the main advantages are the computation gains from resorting to operating in spectral space, which turn the linear operators into simplified per-mode multiplications and divisions [8].

2.4 Spectral-transform and Gaussian-grid cycle

Deriving from what we have explained, we can now put the pieces together to describe how we advance a vorticity-divergence formulation most efficiently by alternating between spectral space and a Gaussian latitude-longitude grid [5]. We utilize the spectral space for when linear operators appear (due to the diagonalization mentioned in 2.3.2). In contrast, when non-linear products such as $(\zeta + f)\mathbf{V}$ or $\frac{1}{2}|\mathbf{V}|^2$ appear, it is cheaper and simpler to evaluate them point-wise on a grid [26]. This transformation cycle is how we exploit both mechanisms in a single timestep. We will outline on a high level how this cycle operates.

At the start of each timestep, we have the prognostic variables: the divergence δ_{nm} and the vorticity ζ_{nm}. The first action is to recover the streamfunction ψ_{nm} and the velocity potential χ_{nm}. This is achieved through the Poisson inversions we outlined before in the equations (2.26) - (2.27) with $\psi_{00} = \chi_{00} = 0$.

After that, we inverse transform to a Gaussian latitude-longitude grid and compute the meridional derivative ∂_φ on that grid, and reconstruct the wind velocities u and v from the equations stated (2.29) - (2.30).

Now that we have u and v on the grid (therefore V), we form the nonlinear vorticity flux $[(\zeta + f)V]$, found in the previously defined vorticity-divergence system, compute it point-wise and take its divergence in case of the equation (2.17),

$$(-\nabla.[(\zeta + f)V])$$

or apply the (vertical curl operator to it in the equation (2.18)),

$$(k.\nabla \times [(\zeta + f)V])$$

Additionally, it is used to calculate the Bernoulli function $\Phi + \frac{1}{2}|\mathbf{V}|^2$ in equation (2.18) and the geopotential tendency in equation (2.9):

$$\partial_t \Phi = -\nabla.[\Phi V]$$

In short, the wind components u, v are required each time we need to build the non-linear terms in the SWE.

The grid-point tendencies $\partial_t \zeta$, $\partial_t \delta$, and $\partial_t \Phi$ are then projected back to spherical harmonics using a forward transform. After that, we advance the prognostic coefficients in time with the chosen time integrator, thus completing one spectral-grid cycle.

In summary, we get the best of both worlds when alternating between spectral space and physical latitude-longitude grid: fast linear operators in spectral space and cheap pointwise evaluation of nonlinear products in physical space. However, it is also worth

noting that trade-offs arise due to this cycle, mainly the cost and complexity of the forward and inverse global transform that bridges the spectral and physical spaces, as it is an all-to-all communication that cannot be split.

To conclude, the concepts developed throughout this chapter form the theoretical foundation upon which the implementation of SWEET's shallow-water solver is built. The vorticity-divergence formulation, along with the spectral-transform and Gaussian-grid cycle, define the structure of the model's spatial discretization and computational flow. These formulations translate directly to the data layout and operator sequence implemented in the SWEET framework. The following chapter focuses on how the theoretical formulations are translated into code within SWEET, and how MPI-based domain decomposition can be applied to these computational structures to enable parallel execution.

3 Implementation

3.1 SWEET

SWEET provides multiple options regarding domains, discretizations, and time integrators. It supports periodic boundary conditions such as the bi-periodic plane (2D torus) and on the sphere. For space discretizations, it offers Fourier spectral methods and finite differences (with convolution in spectral space) for the plane, while the sphere uses spherical harmonics. SWEET also includes many time discretization methods, such as Explicit RK, Implicit RK, Leapfrog , Crank-Nicolson, Semi-Lagrangian [15], and several parallel-in-time approaches such as Parareal, PFASST, Spectral Deferred Corrections, alongside more variants. All of the relevant information about the SWEET repository can be found in the online documentation [19].

3.2 Discretization in SWEET

In this section, we will discuss how SWEET implements the practical aspects of the previously explained theory. We will start from a fine-grained perspective of the tools and structures that SWEET provides, and then describe how these functionalities come together to serve the overall framework of spatial discretization.

3.2.1 Sphere2D

SWEET offers mainly two domains in which the grid is discretized, namely the Cartesian grid ($Cart2D$) and the spherical grid ($Sphere2D$). For the purpose of the thesis, we will introduce the spherical domain only (as it is the one on which our implementation is based).

$Sphere2D$ is the representation of SWEET's spherical physical domain over a latitude-longitude grid paired with a spherical-harmonic spectral space. $Sphere2D$ includes what has been mentioned before in the background theory: a Gaussian grid in physical space (where longitudes are uniformly spaced in $[0, 2\pi)$ and latitudes located at the Gaussian quadrature nodes), and a triangularly truncated spherical-harmonic space T_N used for fast, diagonal action of linear operators.

At the configuration level, $Sphere2D :: Config$ (shared by all grid and spectral objects) defines the sphere's geometric and resolution parameters, ensuring consistent mapping between the two representations. It serves as the bridge between the grid-based and spectral-based data structures, as explained in the next section.

3.2.2 Data containers

SWEET defines two complementary classes for representing the data needed. *DataGrid* stores the real values of the Gaussian latitude-longitude grid (physical space), whereas *DataSpectral* stores the complex spherical-harmonic coefficients f_n^m for spectral space (in accordance with triangular truncation). They both share pointers towards a shared configuration, $Sphere2D :: Config$, which defines attributes relevant to the physical and spectral spaces. In the case of *DataGrid*, it uses the number of longitudes and latitudes that control the allocation space for the physical fields. Conversely, *DataSpectral* is dimensioned according to the triangular truncation, storing the maximum spectral modes for degree and order, thus encompassing the counts of the complex coefficient arrays.

These routines are tightly coupled: nonlinear terms are conveniently evaluated in grid space, whereas linear operators are applied in spectral space. Together, they form the backbone structures of SWEET's numerical workflow. Consequently, transform routines between *DataGrid* and *DataSpectral* object containers are required for the flow of the time-stepping algorithm as the solver runs, which we will cover in the next section.

3.2.3 Operators and transforms

The class $Sphere2D :: Operators$ provides differential operators and transform helpers used through the SWE solver. It relies on the previously mentioned $Sphere2D :: Config$ and the SHTns library for the functions it performs. SHTns is an open-source C library that implements fast and accurate Spherical Harmonics Transforms (SHTs) [16]; the transform operations that map scalar or vector field values to spherical harmonic expansions and vice versa. This library represents the core numerical engine behind all grid-spectral conversions in SWEET, abstracting complex and error-prone low-level transform code to focus on the algorithmic side of the solver. This library is integrated through the initialization of SHTns in the $Sphere2D :: Config$ class. Additionally, the library has OpenMP threading built in, providing additional performance boosts with respect to the number of threads dedicated.

Now that we have introduced the SHTns library, we can highlight the helper functions that wrap the transforms provided, and then see how these individual functionalities

complete the solver's flow of execution.

First, we address the functionality of moving from spectral coefficients to physical values on the Gaussian grid, and for that we introduce the $DataSpectral :: toGrid()$ function that performs an inverse spherical-harmonic transform via SHTns.

```
Sphere2D::DataGrid DataSpectral::toGrid() const
{
    /**
         * Warning: This is an in-situ operation.
         * Therefore, the data in the source array will be destroyed.
        */
    Sphere2D::DataSpectral tmp(*this);
    Sphere2D::DataGrid retval(sphere2DDataConfig);

    SH_to_spat(sphere2DDataConfig->shtns, tmp.spectral_space_data,
    retval.grid_space_data);

    return retval;
}
```

Listing 3.1: toGrid(), inverse transform from spectral space to physical grid

The above routine creates a temporary object in order to allow the SHTns function to overwrite the *spectral_space_data* without mutating the original object. After that, it creates a variable *retval* that contains the same configuration (longitude/latitude counts and memory layout). Finally, it uses the inverse transform that takes a pointer to the SHTns configuration object created in $Sphere2D :: Config$, the *spectral_space_data* containing the coefficients f_n^m, and the *grid_space_data* where the synthesized values $f(\lambda_i, \phi_j)$ on the Gaussian grid are written.

SH_to_spat is an inverse scalar spherical harmonic transform (spectral space to physical space transformation) provided by the SHTNs library. The function takes spectral coefficients and synthesizes a real scalar field on the configured longitude-latitude grid.

Next, we examine the *vrtdiv_2_uv* function, which converts vorticity ζ and divergence δ fields into zonal and meridional velocity components (u,v) in physical space.

```
void Operators::vrtdiv_2_uv(
  const Sphere2D::DataSpectral &i_vrt,
  const Sphere2D::DataSpectral &i_div,
  Sphere2D::DataGrid &o_u,
  Sphere2D::DataGrid &o_v
```

```
) const {
  Sphere2D::DataSpectral psi = inv_laplace(i_vrt)*ir;
  Sphere2D::DataSpectral chi = inv_laplace(i_div)*ir;
  SHsphtor_to_spat(sphere2DDataConfig->shtns,
                   psi.spectral_space_data,
                   chi.spectral_space_data,
                   o_u.grid_space_data,
                   o_v.grid_space_data);
}
```

Listing 3.2: vrtdiv_2_uv

The above code implements the Helmholtz decomposition introduced in Section 2.2.1, specifically Equation (2.15), which recovers physical velocity components from the spectral components. It defines variables attainable through the inverse of the equations defined in (2.16); namely, psi, ψ, calculated through

$$\psi = \nabla^{-2}\zeta \tag{3.1}$$

and the chi, χ, calculated through

$$\chi = \nabla^{-2}\delta \tag{3.2}$$

After that, it multiplies both by the inverse radius *ir* to account for the radius scaling in the discrete formulation. With the resultant components, the function then calls the SHTNs library transform routine, *SHsphtor_to_spat*, which numerically realizes Equation 2.15, retrieving the zonal and meridional velocity components (u, v). The documentation of the library regarding this function [17] lists a more general vector spherical harmonic decomposition that includes radial, spheroidal, and toroidal components. In contrast, our current implementation only deals with the toroidal and poloidal parts, as we are working with a purely horizontal flow.

Next, we introduce the complementary routine of the function explained above, *uv_2_vrtdiv*, which performs the inverse. It takes physical velocity components and recovers the corresponding vorticity ζ and divergence δ in spectral space.

```
void Operators::uv_2_vrtdiv(
                const Sphere2D::DataGrid &i_u,
                const Sphere2D::DataGrid &i_v,
                Sphere2D::DataSpectral &o_vrt,
                Sphere2D::DataSpectral &o_div

)   const {
```

```
8       o_vrt.setup_if_required(i_u.sphere2DDataConfig);
9       o_div.setup_if_required(i_u.sphere2DDataConfig);
10
11      spat_to_SHsphtor(
12              sphere2DDataConfig->shtns,
13              i_u.grid_space_data,
14              i_v.grid_space_data,
15              o_vrt.spectral_space_data,
16              o_div.spectral_space_data
17      );
18      o_vrt = laplace(o_vrt)*r;
19      o_div = laplace(o_div)*r;
20  }
```

Listing 3.3: uv_2_vrtdiv

Analogous to 3.2, *uv_2_vrtdiv* implements the inverse path; it realizes the inverse of Equation (2.15) by calling the library function *spat_to_SHsphtor*, recovering the vorticity and divergence fields from the physical velocity components. Once these potentials are obtained, the code applies the spherical Laplacian in lines 18-19 through the *laplace*() helper function, applying the operations denoted in (2.16) (multiplication by r referring to the radius accounts for the radius scaling used internally).

For more clarity, we present the Laplace and inverse Laplace implementations below.

```
1   DataSpectral Operators::laplace(
2                   const Sphere2D::DataSpectral &i_sph_data
3   )       const
4   {
5       Sphere2D::DataSpectral out_sph_data(i_sph_data);
6
7       out_sph_data.spectral_update_lambda(
8               [&](int n, int m, std::complex<double> &o_data)
9               {
10                  o_data *= -(double)n*((double)n+1.0)*(ir*ir);
11              }
12          );
13
14      return out_sph_data;
15  }
```

Listing 3.4: Laplacian function

```
DataSpectral Operators::inv_laplace(
                const Sphere2D::DataSpectral &i_sph_data
)       const
{
    Sphere2D::DataSpectral out(i_sph_data);

    out.spectral_update_lambda(
            [&](int n, int m, std::complex<double> &o_data)
            {
                if (n != 0)
                    o_data /= -(double)n*((double)n+1.0)*ir*ir;
                else
                    o_data = 0;
            }
        );

    return out;
}
```

Listing 3.5: Inverse Laplacian function

These routines implement the Laplacian-Beltrami operator ∇^2 and its inverse on the sphere. They follow the definition introduced earlier in Equation (2.23), where the eigenvalue of the Laplacian was proportional to $\frac{-n(n+1)}{a^2}$. In the implementation, *ir* equates to $\frac{1}{a}$ in the definition. Consequently, each spectral coefficient is multiplied (Laplacian) or divided (inverse Laplacian) by the scaling. The anonymous function, *spectral_update_lambda*, iterates over all the spectral modes (n,m), which can be considered analogous to a longitude-latitude grid point in spectral space, and then applies the eigenvalue operation to a specific mode. It leverages the diagonal property of the Laplacian in spherical-harmonic space, allowing for independent updates to a mode without affecting the others due to the absence of coupling between the orders or degrees. We expand on its functionality further below.

```
void DataSpectral::spectral_update_lambda(
                std::function<void(int,int,TComplex&)> i_lambda
)
{
    SWEET_THREADING_SPACE_PARALLEL_FOR
    for (int m = 0; m <= sphere2DDataConfig->spectral_modes_m_max; m++)
    {
        std::size_t idx = sphere2DDataConfig->getArrayIndexByModes(m, m);
```

```
9           for (int n = m; n <= sphere2DDataConfig->spectral_modes_n_max; n++)
10          {
11              i_lambda(n, m, spectral_space_data[idx]);
12              idx++;
13          }
14      }
15  }
```

Listing 3.6: Spectral Update Lambda

The code in 3.6 exploits the triangular truncation ($0 \leq m \leq n \leq N$) and the contiguous storage order used by $Sphere2D :: Config$. For each fixed m, it computes the starting linear index using the *getArrayIndexByModes* helper function, shown in 3.7. Now that we have the linear index towards where the first valid $m = n$ exists, we loop over the n to N sequentially using the inner loop, applying the user-provided lambda function to each complex coefficient denoted by the specific mode (m, n). Since the previously defined Laplacian operator is diagonal in spherical-harmonic space (along with the other operators previously mentioned in 2.3.2), we can have the generalized update function *spectral_update_lambda* in order to apply these operations mode-wise without any inter-mode coupling. This allows for parallel traversal with a complexity of $O(N^2)$, along with the application of OpenMP parallelism to speed up the iteration, through the applied OpenMP wrapper macro, *SWEET_THREADING_SPACE_PARALLEL_FOR*. To be able to visualize the data we are working with more, we include an explanation of the *getArrayIndexbyModes* function.

```
1   std::size_t getArrayIndexByModes(
2           int n,
3           int m
4   )       const
5   {
6       SWEET_ASSERT(n >= 0);
7       SWEET_ASSERT(n >= m);
8
9       return (m*(2*spectral_modes_n_max-m+1)>>1)+n;
10  }
```

Listing 3.7: Get Array by Index Modes

To efficiently store and access the spectral coefficients, SWEET flattens the natural two-dimensional array of spherical-harmonic modes (m,n) into a one-dimensional linear memory layout. Since the logic follows a triangular truncation according to

$0 \leq m \leq n \leq N$, the spectral grid forms a sparse 2D matrix in an upper triangular form to represent the valid coefficients. Figure 3.1 provides an example layout of such a grid. The figure highlights the valid coefficients, filled with their corresponding (n, m) mode indices, while shaded cells indicate invalid combinations where $m > n$.

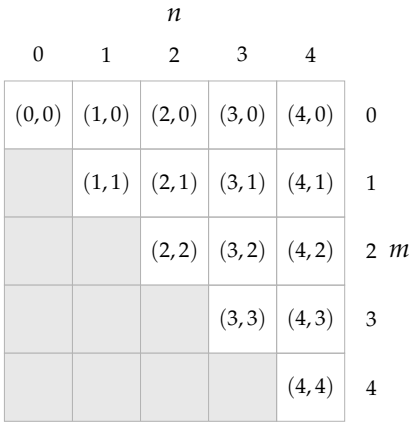


Figure 3.1: Triangular storage for spherical–harmonic coefficients: valid modes satisfy $0 \leq m \leq n \leq N$. Each fixed-m row is contiguous in memory; the row start offset is $\frac{m(2N-m+1)}{2}$.

SWEET stores the triangular matrix of spectral coefficients using a 1D array by arranging the 2D grid in a compact, m-row-wise, contiguous manner. Therefore, *getArrayByIndexModes* relies on the following index formula in order to translate a specific mode (n, m) to a 1D idx,

$$\mathrm{idx}(n, m) = \frac{m(2N - m + 1)}{2} + n, \tag{3.3}$$

where N is the maximum degree of truncation.

For further visual clarity, figure 3.2 shows what the data looks like in its 1D form.

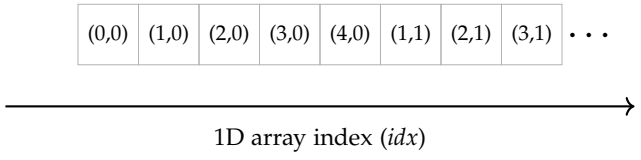


Figure 3.2: Flattened one-dimensional array representation of the spectral coefficients in SWEET. The layout follows the m-row-wise memory order shown in Figure 3.1.

It is worth noting that the physical grid counterpart to the discussed spectral one also follows a similar storage layout. The 2D longitude-latitude grid in physical space is converted into a 1D array, either longitude-contiguous or latitude-contiguous, depending on the configuration variables, and stored accordingly.

Additionally, SWEET overloads basic arithmetic operations on both grid and spectral containers to express common operations as single, data-parallel kernels. Below are some examples of the functions relevant to the implementation discussed in later sections.

```
const DataGrid& DataGrid::operator*=(
                const double i_value
)       const
{
        SWEET_THREADING_SPACE_PARALLEL_FOR_SIMD
        for (std::size_t idx = 0; idx <
    sphere2DDataConfig->grid_number_elements; idx++)
                grid_space_data[idx] *= i_value;

        return *this;
}

DataGrid DataGrid::operator*(
                const double i_value
)       const
{
        DataGrid out(sphere2DDataConfig);

        SWEET_THREADING_SPACE_PARALLEL_FOR_SIMD
        for (std::size_t i = 0; i <
    sphere2DDataConfig->grid_number_elements; i++)
                out.grid_space_data[i] = grid_space_data[i]*i_value;

        return out;
}

DataGrid DataGrid::operator+(
                const DataGrid &i_sph_data
)       const
{
        check(i_sph_data.sphere2DDataConfig);

        DataGrid out(sphere2DDataConfig);

```

```
33          SWEET_THREADING_SPACE_PARALLEL_FOR_SIMD
34          for (std::size_t idx = 0; idx <
    sphere2DDataConfig->grid_number_elements; idx++)
35                  out.grid_space_data[idx] = grid_space_data[idx] +
    i_sph_data.grid_space_data[idx];
36
37          return out;
38  }
```

Listing 3.8: Overloaded Arithemetic Operators in DataGrid

The in-place $operator* =$ multiples each grid value by a constant factor, whereas $operator*$ returns a new *DataGrid* with the scaled result, which is useful when the left operand must remain unchanged. All three loops utilize the OpenMP-based macro *SWEET_THREADING_SPACE_PARALLEL_FOR_SIMD*, speeding up the computations.

```
1   DataSpectral DataSpectral::operator+(
2                   const Sphere2D::DataSpectral &i_sph_data
3   )       const
4   {
5           check(i_sph_data.sphere2DDataConfig);
6
7           Sphere2D::DataSpectral out(sphere2DDataConfig);
8
9
10          SWEET_THREADING_SPACE_PARALLEL_FOR_SIMD
11          for (int idx = 0; idx <
    sphere2DDataConfig->spectral_array_data_number_of_elements; idx++)
12                  out.spectral_space_data[idx] = spectral_space_data[idx] +
    i_sph_data.spectral_space_data[idx];
13
14          return out;
15  }
16
17  DataSpectral DataSpectral::operator*(
18                  const Sphere2D::DataSpectral &i_sph_data
19  )       const
20  {
21          check(i_sph_data.sphere2DDataConfig);
22
23          Sphere2D::DataGrid a = getSphere2DDataGrid();
24          Sphere2D::DataGrid b = i_sph_data.toGrid();
```

```
25
26          Sphere2D::DataGrid mul(sphere2DDataConfig);
27
28          SWEET_THREADING_SPACE_PARALLEL_FOR_SIMD
29          for (std::size_t i = 0; i <
    sphere2DDataConfig->grid_number_elements; i++)
30                  mul.grid_space_data[i] =
    a.grid_space_data[i]*b.grid_space_data[i];
31
32          Sphere2D::DataSpectral out(mul);
33
34          return out;
35  }
```

Listing 3.9: Overloaded Arithemetic Operators in DataSpectral

Similar to the *DataGrid* class, the *DataSpectral* container also overloads arithmetic operators to simplify operations on spectral coefficients. The addition operator ($operator+$) performs mode-wise addition, updating each spherical-harmonic coefficient independently. Since these coefficients are stored in contiguous memory, the operation is easily parallelized and benefits from SIMD execution, just as the grid-space version does.

The multiplication operator ($operator*$) follows a pseudo-spectral approach rather than a direct coefficient-wise multiplication. Both input fields are first transformed from spectral space to physical space, where the element-wise multiplication is performed. The result is then transformed back to spectral space to form the output.

Together, the *DataGrid* and *DataSpectral* operators provide a unified and efficient way to perform arithmetic operations directly within SWEET, enabling simple expressions of physical and spectral relationships while maintaining parallel efficiency. This provides the building blocks for linear combinations and scalings, while nonlinear products are routed through the grid using SHTNs spherical transforms. This framework maintains the code's expressiveness and performance, slotting naturally into the transform-operator pattern established earlier and the flow within a single timestep, as described in the next section.

3.2.4 Flow within a Timestep

Up to this point, we have described the building blocks that constitute SWEET's computational core, including the grid and spectral data structure, spherical operators, and transform routines that bridge both representations. These components work together in order to enable the solver's numerical workflow. They define how physical quantities (i.e., vorticity, divergence) are stored, manipulated, and exchanged between

physical and spectral space.

We now bring these components together by describing the flow of spatial operations within a single timestep. In SWEET, time integration is handled by modular timestepping classes, each corresponding to a specific numerical method. For SWE on the sphere, the Eulerian timestep update function is the routine used by the Runge-Kutta (RK) timestepping method as a form of a forward update. We will cover how this update works in a single time step in this section, and then explain its continuous application within the entire RK framework in a later section.

The Eulerian timestep update function is the right-hand side evaluator for one stage. In other words, it calculates the tendencies (instantaneous slopes) of the prognostic variables we are dealing with (i.e., vorticity). In SWEET, this functionality is implemented in the *PDESWESphere2DTS_ln_erk* :: *euler_timestep_update_pert* function.

```
1  void PDESWESphere2DTS_ln_erk::euler_timestep_update_pert(
2                  const sweet::Data::Sphere2D::DataSpectral &i_phi_pert,
3                  const sweet::Data::Sphere2D::DataSpectral &i_vrt,
4                  const sweet::Data::Sphere2D::DataSpectral &i_div,
5
6                  sweet::Data::Sphere2D::DataSpectral &o_phi_pert_t,
7                  sweet::Data::Sphere2D::DataSpectral &o_vrt_t,
8                  sweet::Data::Sphere2D::DataSpectral &o_div_t,
9
10                 double i_simulation_timestamp
11 )
12 {
13         double gh0 = shackPDESWESphere2D->gravitation *
       shackPDESWESphere2D->h0;
14
15         sweet::Data::Sphere2D::DataGrid phi_pert_phys =
       i_phi_pert.toGrid();
16
17         sweet::Data::Sphere2D::DataGrid ug, vg;
18         ops->vrtdiv_2_uv(i_vrt, i_div, ug, vg);
19
20         sweet::Data::Sphere2D::DataGrid vrtg = i_vrt.toGrid();
21
22         using namespace sweet;
23
24         sweet::Data::Sphere2D::DataGrid u_nl = ug*(vrtg+fg);
25
26         sweet::Data::Sphere2D::DataGrid v_nl = vg*(vrtg+fg);
27
```

```
28          ops->uv_2_vrtdiv(u_nl, v_nl, o_div_t, o_vrt_t);
29
30
31          o_vrt_t *= -1.0;
32
33          sweet::Data::Sphere2D::DataGrid tmpg = 0.5*(ug*ug+vg*vg);
34
35          sweet::Data::Sphere2D::DataSpectral e = phi_pert_phys+tmpg;
36
37
38          o_div_t -= ops->laplace(e);
39
40          u_nl = ug*(phi_pert_phys + gh0);
41          v_nl = vg*(phi_pert_phys + gh0);
42
43          ops->uv_2_vrtdiv(u_nl,v_nl, e, o_phi_pert_t);
44
45          o_phi_pert_t *= -1.0;
46
47  }
```

Listing 3.10: Euler Timestep Update function

The shallow-water update implemented above follows the vorticity-divergence formulations explained in section 2.2.1, as well as unfolded chronologically with respect to the provided code in section 8.1 of [18], in conjunction with the equations provided in [28].

Before we dive into the body of the function, we will explain the purpose of the function through its signature. This function acts as a right-hand side (RHS) evaluator, in other words, a "tendency routine". This means that it merely calculates the time derivatives (slopes), which represent how the physical variables that we deal with in the vorticity-divergence formula tend, without updating the state of these variables. In a simplistic mathematical expression, it is just computing the derivative of a certain variable y, $\frac{dy}{dt}$, from y at time t.

For the inputs, we start with the geopotential perturbation variable, Φ', which is connected to the geopotential we mentioned previously in Equation (2.9) in section 2.2.1. Specifically, the geopotential we discussed is comprised of two parts expressed below,

$$\Phi = g\,h = \underbrace{g\,h_0}_{\Phi_0\ (constant)} + \underbrace{g\,(h - h_0)}_{\Phi'\ (perturbation)} . \tag{3.4}$$

Φ' is the dynamically relevant variable that changes in time, whereas Φ_0 is a constant that vanishes in the derivation, which is why gradients see the perturbation essentially, leading us to use it specifically in this timestep function.

The function essentially takes the prognostic state variables, vorticity ζ, divergence δ, and geopotential perturbation Φ', which we can summarize in a prognostic state

$$\mathbf{y}(t) = \left[\, \Phi'(t),\ \zeta(t),\ \delta(t) \,\right]^\top$$

The function returns the tendency of the above variables $(\partial_t\Phi', \partial_t\zeta, \partial_t\delta)$.

The input variables are saved in their respective *DataSpectral* objects: *i_phi_pert*, *i_vrt*, and *i_div*. The function's signature also includes the placeholder objects, *o_phi_pert_t*, *o_vrt_t*, and *o_div_t*, which refer to the tendency equations we derived in 2.2.1, reiterated below

$$\frac{\partial \Phi'}{\partial t} = -\nabla \cdot \left[(\Phi_0 + \Phi')\, \mathbf{V} \right]$$

$$\frac{\partial \zeta}{\partial t} = -\nabla . \left[(\zeta + f)\, \mathbf{V} \right]$$

$$\frac{\partial \delta}{\partial t} = \mathbf{k} \cdot \nabla \times \left[(\zeta + f)\, \mathbf{V} \right] \ - \ \nabla^2 \left(\Phi' + \frac{1}{2} |\mathbf{V}|^2 \right)$$

Now that we understand why the function is used, we explain how it calculates the state's tendency by connecting the code to the previous theory.

It starts by calculating the reference geopotential $\Phi_0 = gh_0$ at line 15, which will be used later as part of the expression $\Phi = \Phi' + \Phi_0$.

After that, it invokes the *toGrid*() function (code block 3.1) to synthesize the physical perturbation grid $\Phi'(\lambda, \varphi)$ from the spectral input perturbation *i_pert_phys*, which makes it cheaper due to it being a nonlinear product that is more computationally friendly in physical space.

This is followed by creating grids *ug* and *vg* on line 17, which reference the physical wind velocities to be recovered using the spectral vorticity and divergence. In order to achieve that, it calls the routine *vrtdiv*_2_*uv* (code block 3.2). It synthesizes the physical absolute pointwise vorticity *vrtg*.

On lines 24 and 25, it calculates the expression

$$\mathbf{V} * (\zeta + f)$$

which is a repeatedly used part of the tendency equation system explained before. f, or fg as mentioned in the code, refers to the Coriolis parameter. It calculates the

absolute-vorticity flux $(\zeta + f)v$ through using the zonal (ug) and meridional (vg) wind components

$$\mathbf{V}(\zeta + f) = (u(\zeta + f), v(\zeta + f))$$

After that, it takes these components and applies the routine *uv_2_vrtdiv* (code block 3.3) in order to compute the vorticity and divergence updates (ζ', δ'). These are realized through applying the divergence operator to the flux (in the case of ζ') and the curl operator for the divergence for δ'. We then finalize ζ' by multiplying its grid by the negative sign.

Following the computations needed for δ', it calculates $\frac{1}{2}|\mathbf{V}|^2$ using ug and vg on line 33, followed by adding the result to the geopotential perturbation Φ', applying Laplace (code block 3.4), and subtracting it from the final δ' result (which already contains the term $\mathbf{k} \cdot \nabla \cdot \left[(\zeta + f)\,\mathbf{V}\right]$ from the operation on line 28), thus finalizing δ'.

Now that the calculations for the vorticity and divergence tendencies are complete, the code proceeds with the requirements for the geopotential perturbation.

The rest of the code constructs the geopotential flux ΦV. This is done through the expressions evaluated on lines 40 and 41, which reflect

$$\Phi\mathbf{V} = (u(\Phi_0 + \Phi'), v(\Phi_0 + \Phi'))$$

(remembering that $\Phi_0 = gh_0$)

Next, $uv\_2\_vrtdiv(u\_nl, v\_nl, e, o\_phi\_pert\_t)$ applies the spherical harmonic transform to the flux and returns its divergence to spectral space, then applies the negative sign by multiplying the spectral coefficients by their negative in line 45, completing the expression

$$\frac{\partial \Phi'}{\partial t} = -\nabla \cdot \left[(\Phi_0 + \Phi')\,\mathbf{V}\right]$$

and therefore, all the necessary tendencies have been calculated.

Stepping back, the overall purpose of this function is tied to the concept of space discretization. The solver first discretizes the space using the logic of this function, and then turns the PDEs into a big system of ODEs in time. Every operation inside it is a spatial operation that evaluates the state's variables at a specific time. This function then delivers the spatially-discretized $(\partial_t \Phi', \partial_t \zeta, \partial_t \delta)$ to the time discretization scheme (e.g., Runge-Kutta) that then uses these tendencies to advance the solution in time.

3.2.5 Galewsky Barotropic-Instability Benchmark

Before delving into the implementation details, it is helpful to frame what and why we use the Galewsky benchmark, as well as what we will actually be running in the next sections, given that our first milestone is to test the SWE solver and profile its performance.

To assess dynamical cores on the rotating sphere beyond steady and weakly nonlinear flows (similar to [28]), we include the Galewsky barotropic-instability benchmark [6]. This setup prescribes a single-layer SWE on the rotating sphere with a balanced mid-latitude jet (similar to a jet stream) and a small free-surface perturbation that perturbs the flow. This poke causes the smooth jet to become unstable and break into thin vorticity filaments over a few days. Because the recipe is simple to set up and creates a complex and realistic flow, it is ideal for testing whether a solver's spatial operators produce sensible results.

We base all the subsequent experiments on this Galewsky setup, using the parameter choices recommended by the default SWEET Galewsky tutorial.

3.3 Performance Evaluation of the Original Implementation

To identify computational bottlenecks in the original SWEET implementation, we conducted a performance analysis using the Intel VTune Profiler on the LRZ Linux cluster. This profiling established a baseline for runtime behavior and guided the subsequent optimization efforts. Detailed profiling parameters and execution settings are provided in the Appendix. We present the results below.

Routine Name	**Effective Time (%)**
_an22_l	11.3
split_north_south_real_imag	6.5
n1_12	5.6
_sy1_hi2_l	4.6
_sy2_hi2_l	3.4
_an12_l	2.8
SH_2scal_to_vect	0.5
Total approximate (SHTns-related)	$\approx$ **34.7**

Table 3.1: Effective time distribution of spherical harmonic transform–related routines (8-thread configuration).

Routine Name	Effective Time (%)
sweet::Data::Sphere2D::DataGrid::operator*._omp_fn.0	7.4
sweet::Data::Sphere2D::DataGrid::operator+._omp_fn.0	6.2
sweet::Data::Sphere2D:Operators::laplace...	3.8
std::operator*<double>	2.4
sweet::Memory::parmemcpy._omp_fn.0	2.0
std::forward<int>	1.5
std::function<void(...)>::operator()	1.5
sweet::Data::Sphere2D::DataSpectral::operator*._omp_fn.0	1.5
sweet::Data::Sphere2D:Operators::inv_laplace...	1.4
std::complex<double>::operator*=	1.2
std::operator+<double>	1.1
std::complex<double>::operator/=	0.9
Total approximate (non-SHTns–related)	$\approx$ **30.9**

Table 3.2: Effective time distribution of non-SHTns (non–spherical harmonic) routines in the SWEET solver (8-thread configuration).

Collectively, these results indicate that the spherical harmonic transformation routines are the principal computational bottleneck in SWEET's spatial backend. While highly optimized for shared-memory execution due to SHTNs library OpenMP support, they are effectively encapsulated from SWEET's side. Because their implementation and data structures are managed internally by the library, manual MPI-based domain decomposition cannot be applied to them directly. Furthermore, the global nature of spherical harmonic transforms inherently requires all-to-all data access, making it infeasible to split their work across distributed processes without modifying the library itself.

Consequently, our parallelization strategy must focus on the portions of the solver that remain outside the SHTns boundary, namely the gridpoint-based and spectral-based operations that handle other operations. These components operate on local data, with their logic directly written in SWEET, allowing us to insert MPI code that enables partitioning of these operations.

While these routines (Table 3.2) collectively consume a smaller portion of total runtime than the SHTns transforms, they still represent a substantial share of computation within SWEET's spatial backend. Unlike the encapsulated SHTNs kernels, these routines are fully visible at the source level and operate on contiguous memory buffers. This makes them both accessible and meaningful targets for distributed-memory parallelization.

From the profiling results summarized in Table 3.2, we observe that many of the non-SHTns routines correspond directly to operations implemented inside the Euler-timestep update and its supporting spectral kernels. In particular, the grid arithmetic routines (e.g, $operator*$) and the spectral differential operators (e.g., $laplace$) are invoked repeatedly inside the *euler_timestep_update_pert* function during the evaluation of the vorticity-divergence tendencies. Moreover, both the Laplacian and its inverse rely internally on the *spectral_update_lambda* mechanism, which iterates over all the spherical-harmonic modes. Therefore, these hotspots are going to be the focus of the applied MPI-based modifications in the next section.

On a final note, it is important to note that the percentages in Tables 3.1 and 3.2 do not add up to 100%. This is expected, as Vtune reports only the dominant routines exceeding its sampling significance threshold. Numerous smaller routines, runtime functions, and system-level overheads are omitted or grouped under aggregated entries. Nevertheless, the identified functions capture the majority of the computational load and are sufficient to guide subsequent parallelization analysis.

3.4 Implementation of MPI-Enabled Spatial Discretization

3.4.1 Domain Decomposition Strategy

To enable distributed-memory parallel execution, we employ a domain decomposition strategy to partition the data among MPI ranks. The decomposition is designed to be lightweight, minimally invasive, and compatible with SWEET's existing data structures. Each MPI rank operates on a distinct portion of the global grid or spectral array, performing local computations independently before collectively exchanging data as needed. This allows the solver to parallelize key computational kernels, particularly those associated with the ERK timestep and spectral updates, without altering the mathematical formulation of the shallow-water equations.

3.4.2 Build-time switch and code layout

SWEET is compiled with SCons. When the build flag $--sweet-mpi=enable$ is passed (as in our baseline build), the SCons configuration defines the preprocessor macro *SWEET_MPI* and links the code against the MPI library supplied on the LRZ system. Concretely, the flag:

```
scons ... --sweet-mpi=enable ...
```

results in the compilation with $-DSWEET\_MPI = 1$ and the appropriate MPI include paths and libraries. If the flag is omitted or set to *disable*, the macro is not defined, and the binary is built without any MPI dependency.

We exploit this build-time switch to keep the original serial/OpenMP code path intact and layer MPI logic alongside it. ALL MPI-specific code is guarded with **#if SWEET_MPI** so that:

- With $--sweet-mpi = disable$, the original single-node behaviour is preserved (i.e., for validation)
- With $--sweet-mpi = enable$, the alternative paths supporting domain decomposition and collective communication are compiled.

3.4.3 Integration in the ERK Time-Stepping Routine

The central routine modified to support distributed-memory parallelism exists within *euler_timestep_update_pert()* function. As described before, this function computes the nonlinear vorticity-divergence tendencies during the Runge-Kutta stage of the shallow-water equations. The code block below shows the MPI-enabled implementation, where the **#if SWEET_MPI** guard isolates distributed-memory operations from the original shared-memory path.

```
1  void PDESWESphere2DTS_ln_erk::euler_timestep_update_pert(
2      const sweet::Data::Sphere2D::DataSpectral &i_phi_pert,
3      const sweet::Data::Sphere2D::DataSpectral &i_vrt,
4      const sweet::Data::Sphere2D::DataSpectral &i_div,
5      sweet::Data::Sphere2D::DataSpectral &o_phi_pert_t,
6      sweet::Data::Sphere2D::DataSpectral &o_vrt_t,
7      sweet::Data::Sphere2D::DataSpectral &o_div_t,
8      double i_simulation_timestamp
9  )
10 {
11 #if SWEET_MPI
12     // --- MPI setup
       ------------------------------------------------------------
13     int mpi_rank, num_mpi_ranks;
14     MPI_Comm mpi_comm = MPI_COMM_WORLD;
15     MPI_Comm_rank(mpi_comm, &mpi_rank);
16     MPI_Comm_size(mpi_comm, &num_mpi_ranks);
17
18     const double gh0 = shackPDESWESphere2D->gravitation *
       shackPDESWESphere2D->h0;
```

```
19      sweet::Data::Sphere2D::DataGrid phi_pert_phys = i_phi_pert.toGrid();
20
21      sweet::Data::Sphere2D::DataGrid ug, vg;
22      ops->vrtdiv_2_uv(i_vrt, i_div, ug, vg);
23
24      sweet::Data::Sphere2D::DataGrid vrtg = i_vrt.toGrid();
25
26      // Scratch arrays (grid space)
27      sweet::Data::Sphere2D::DataGrid
28          u_nl(ug.sphere2DDataConfig),
29          v_nl(ug.sphere2DDataConfig),
30          u_nl_step_2(ug.sphere2DDataConfig),
31          v_nl_step_2(ug.sphere2DDataConfig),
32          tmpg(ug.sphere2DDataConfig);
33
34      // --- 1D block decomposition of the grid array
        ---------------------------
35      const int total_size =
36          ug.sphere2DDataConfig->grid_num_lat *
        ug.sphere2DDataConfig->grid_num_lon;
37
38      const int base = total_size / num_mpi_ranks;
39      const int rem  = total_size % num_mpi_ranks;
40
41      auto count_of = [&](int r){ return base + (r < rem ? 1 : 0); };
42      auto disp_of  = [&](int r){ return r*base + std::min(r, rem); };
43
44      std::vector<int> counts(num_mpi_ranks), displs(num_mpi_ranks);
45      for (int r = 0; r < num_mpi_ranks; ++r) {
46          counts[r] = count_of(r);
47          displs[r] = disp_of(r);
48      }
49
50      const int s     = displs[mpi_rank];
51      const int limit = s + counts[mpi_rank];
52
53      // Raw pointers for tight loops
54      double * __restrict__ unlg  = u_nl.grid_space_data;
55      double * __restrict__ vnlg  = v_nl.grid_space_data;
56      double * __restrict__ tmpgG = tmpg.grid_space_data;
57      double * __restrict__ u2g   = u_nl_step_2.grid_space_data;
58      double * __restrict__ v2g   = v_nl_step_2.grid_space_data;
59
60      const double * __restrict__ ugp   = ug.grid_space_data;
```

```
61      const double * __restrict__ vgp   = vg.grid_space_data;
62      const double * __restrict__ vrtgp = vrtg.grid_space_data;
63      const double * __restrict__ fgp   = fg.grid_space_data;
64      const double * __restrict__ phip  = phi_pert_phys.grid_space_data;
65
66      // --- Local nonlinear updates on this rank (hybrid: MPI - OpenMP)
        --------
67      #pragma omp parallel for simd schedule(static)
68      for (int g = s; g < limit; ++g) {
69          const double uu  = ugp[g];
70          const double vv  = vgp[g];
71          const double sum = vrtgp[g] + fgp[g];
72          const double ph  = phip[g];
73
74          unlg[g]  = uu * sum;
75          vnlg[g]  = vv * sum;
76          tmpgG[g] = 0.5 * (uu*uu + vv*vv);
77          u2g[g]   = uu * (ph + gh0);
78          v2g[g]   = vv * (ph + gh0);
79      }
80
81      // --- Nonblocking all-gathers to restore global fields
        -------------------
82      MPI_Request req_u, req_v, req_tmpg, req_u2, req_v2;
83
84      MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
85                      u_nl.grid_space_data, counts.data(), displs.data(),
        MPI_DOUBLE,
86                      mpi_comm, &req_u);
87
88      MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
89                      v_nl.grid_space_data, counts.data(), displs.data(),
        MPI_DOUBLE,
90                      mpi_comm, &req_v);
91
92      MPI_Wait(&req_u, MPI_STATUS_IGNORE);
93      MPI_Wait(&req_v, MPI_STATUS_IGNORE);
94
95      MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
96                      tmpg.grid_space_data, counts.data(), displs.data(),
        MPI_DOUBLE,
97                      mpi_comm, &req_tmpg);
98
99      MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
```

```
100                     u_nl_step_2.grid_space_data, counts.data(),
    displs.data(), MPI_DOUBLE,
101                     mpi_comm, &req_u2);
102
103     MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
104                     v_nl_step_2.grid_space_data, counts.data(),
    displs.data(), MPI_DOUBLE,
105                     mpi_comm, &req_v2);
106
107     ops->uv_2_vrtdiv(u_nl, v_nl, o_div_t, o_vrt_t);
108     o_vrt_t *= -1.0;  // (Step 1e)
109
110     MPI_Wait(&req_tmpg, MPI_STATUS_IGNORE);
111     sweet::Data::Sphere2D::DataSpectral e = phi_pert_phys + tmpg;
112
113     o_div_t -= ops->laplace(e);
114
115     MPI_Wait(&req_u2, MPI_STATUS_IGNORE);
116     MPI_Wait(&req_v2, MPI_STATUS_IGNORE);
117
118     ops->uv_2_vrtdiv(u_nl_step_2, v_nl_step_2, e, o_phi_pert_t);
119     o_phi_pert_t *= -1.0;
120
121 #else
122     // ---------------------- Original serial/OpenMP path
    ----------------------
123     // covered in previous sections
124 #endif
125 }
```

Listing 3.11: `euler_timestep_update_pert()`: ERK stage update with MPI-based domain decomposition (guarded by `SWEET_MPI`).

Let us examine what is happening in this block now. The function can be divided into several main regions, each serving a distinct role in enabling hybrid MPI–OpenMP parallelism. Given that we have explored the logic of the original routine (code block 3.10), we will focus on the added parts that enable us to achieve the new distributed routine.

```
int mpi_rank, num_mpi_ranks;
MPI_Comm mpi_comm = MPI_COMM_WORLD;
MPI_Comm_rank(mpi_comm, &mpi_rank);
MPI_Comm_size(mpi_comm, &num_mpi_ranks);
```

We begin with the standard MPI initial configuration by binding to *MPI_COMM_WORLD* and retrieving the current process identifier *mpi_rank* and the total number of processes *num_mpi_ranks*. This establishes the distributed context and determines which portion of the global data this rank is responsible for.

```
sweet::Data::Sphere2D::DataGrid
u_nl(ug.sphere2DDataConfig),
v_nl(ug.sphere2DDataConfig),
u_nl_step_2(ug.sphere2DDataConfig),
v_nl_step_2(ug.sphere2DDataConfig),
tmpg(ug.sphere2DDataConfig);
```

These temporary arrays mirror the intermediate fields in the original spatial formulation but are made explicit to support distributed execution. The arrays *u_nl* and *v_nl* store the nonlinear flux terms $u(\zeta + f)$ and $v(\zeta + f)$, computed locally on each rank before being gathered for the global *uv_2_vrtdiv* operation. The array *tmpg* holds the energy term $\frac{1}{2}(u^2 + v^2)$, which is locally evaluated and later all-gathered before applying the Laplacian. Finally, *u_nl_step_2* and *v_nl_step_2* represent the geopotential tendency terms $u(\Phi' + gh_0)$ and $v(\Phi' + gh_0)$, computed separately to make the communication pattern explicit and avoid overwriting earlier data. All arrays share the same configuration as *ug*, ensuring identical layout and compatibility with the original SWEET operators.

```
int total_size =
    ug.sphere2DDataConfig->grid_num_lat * ug.sphere2DDataConfig->grid_num_lon;

int base = total_size / num_mpi_ranks;
int rem  = total_size % num_mpi_ranks;

auto count_of = [&](int r){ return base + (r < rem ? 1 : 0); };
auto disp_of  = [&](int r){ return r*base + std::min(r, rem); };

int local_n = count_of(mpi_rank);

std::vector<int> counts(num_mpi_ranks), displs(num_mpi_ranks);
for (int r = 0; r < num_mpi_ranks; ++r) {
    counts[r] = count_of(r);
    displs[r] = disp_of(r);
}

const int s = displs[mpi_rank];
const int limit = s + counts[mpi_rank];
```

We apply a 1D block decomposition over the flattened latitude-longitude grid of size *total_size*. The division is even, and any remainder is distributed to the first *rem* ranks to maintain near-perfect load balance. The pair of arrays $(counts, displs)$ encodes, for every rank, how many elements it owns and where its block begins in the global array. This design avoids any per-step broadcasts since every rank deterministically computes the same $(counts, displs)$ locally. Subsequently, the variables s and *limit* define the rank's closed-open processing window in the global indexing, allowing all ranks to share the same loop definition without requiring extra local reindexing.

```
double       * __restrict__ unlg  = u_nl.grid_space_data;
double       * __restrict__ vnlg  = v_nl.grid_space_data;
double       * __restrict__ tmpgG = tmpg.grid_space_data;
double       * __restrict__ u2g   = u_nl_step_2.grid_space_data;
double       * __restrict__ v2g   = v_nl_step_2.grid_space_data;

const double * __restrict__ ugp   = ug.grid_space_data;
const double * __restrict__ vgp   = vg.grid_space_data;
const double * __restrict__ vrtgp = vrtg.grid_space_data;
const double * __restrict__ fgp   = fg.grid_space_data;
const double * __restrict__ phip  = phi_pert_phys.grid_space_data;
```

For the loops, we bind raw pointers to contiguous buffers and mark them *__restrict__* to enable better auto-vectorization as a minor enhancement. This also simplifies access syntax, eliminating the need for repeated long member expressions and enhancing readability.

```
#pragma omp parallel for simd schedule(static)
for (int g = s; g < limit; ++g){
    const double uu  = ugp[g];
    const double vv  = vgp[g];
    const double sum = vrtgp[g] + fgp[g];
    const double ph  = phip[g];

    unlg[g]  = uu * sum;
    vnlg[g]  = vv * sum;
    tmpgG[g] = 0.5 * (uu*uu + vv*vv);
    u2g[g]   = uu * (ph + gh0);
    v2g[g]   = vv * (ph + gh0);
}
```

This is the local nonlinear update on each rank for its subrange $[s, limit)$. We utilize OpenMP with *simd* to exploit core-level parallelism, while MPI provides inter-

rank parallelism, resulting in a hybrid OpenMP-MPI mode. The computed products correspond to the nonlinear parts of the vorticity/divergence tendencies, as well as the geopotential tendency. This is an excellent target for domain decomposition due to the embarrassingly parallel nature of the work over the grid points.

```
MPI_Request req_u, req_v, req_tmpg, req_u_step_2, req_v_step_2;
MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
                u_nl.grid_space_data, counts.data(), displs.data(),
                 ↪  MPI_DOUBLE, mpi_comm, &req_u);
MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
                v_nl.grid_space_data, counts.data(), displs.data(),
                 ↪  MPI_DOUBLE, mpi_comm, &req_v);

MPI_Wait(&req_u, MPI_STATUS_IGNORE);
MPI_Wait(&req_v, MPI_STATUS_IGNORE);

MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
                tmpg.grid_space_data, counts.data(), displs.data(),
                 ↪  MPI_DOUBLE, mpi_comm, &req_tmpg);
MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
                u_nl_step_2.grid_space_data, counts.data(), displs.data(),
                 ↪  MPI_DOUBLE, mpi_comm, &req_u_step_2);
MPI_Iallgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL,
                v_nl_step_2.grid_space_data, counts.data(), displs.data(),
                 ↪  MPI_DOUBLE, mpi_comm, &req_v_step_2);
ops->uv_2_vrtdiv(u_nl, v_nl, o_div_t, o_vrt_t);
```

After local computation, each rank must see the full global fields before applying operators that conceptually act on the whole sphere (e.g, transform back to spectral space, Laplacian). We therefore issue non-blocking all-gathers to assemble the global arrays in-place. *MPI_Iallgatherv* accommodates uneven block sizes (which is suitable for our implementation that might lead to minor uneven load distributions). The non-blocking behavior enables us to overlap computation operations and gather operations, maximizing work across the ranks. As soon as the velocity fields (*u_nl*, *v_nl*) are globally available, we synchronize their request using *MPI_WAIT* due to the subsequent operator call *uv_2_vrtdiv* on these grids. In contrast, the remaining quantities (e.g. *tmpg*, *u_nl_step_2*, *v_nl_step_2*) are not consumed until later in the update, so their gathers can continue in the background until we issue an *MPI_WAIT* on them before their usage. Using *MPI_IN_PLACE* avoids extra staging buffers by letting each rank retain its local segment and and the allgather fills in the rest according to the *counts* and *displs*.

Complementing this context, all calls to *laplace* (refer 3.4) and *inv_laplace* (refer 3.5), whether inside the ERK update directly or within the *uv_2_vrtdiv* (refer 3.3) and *vrtdiv_2_uv* (refer 3.2), rely on the *spectral_update_lambda* mechanism (refer 3.6) to apply their operations across spectral coefficients. This abstraction allows each diagonal operation in spectral space, such as scaling by the Laplacian eigenvalues, to be expressed as a simple elementwise lambda acting over the (m, n) spectral indices. In the following section, we extend this implementation of this function to a parallel version.

3.4.4 Integration in the Spectral Update routine

We will now walk through the parallelization of the *spectral_update_lambda* routine.

```
1
2  inline int fast_find_row(std::size_t idx, std::size_t N)
3  {
4      const double B = double(2*N + 3);
5      double m0 = std::floor((B - std::sqrt(B*B - 8.0*double(idx))) * 0.5);
6      std::size_t m = int(m0);
7
8      // correct by at most one step
9      while ( ((m+1) * (2*N - (m+1) + 3) >> 1) <= idx ) ++m;
10     while ( (m * (2*N - m + 3) >> 1) > idx ) --m;
11     return m;
12 }
13
14 void DataSpectral::spectral_update_sub(TComplex *buf,
15                                         std::size_t global_offset_idx,
16                                         std::size_t local_len,
17
       std::function<void(int,int,TComplex&)> f)
18 {
19     if (local_len <= 0) return; // safety check
20         std::size_t M = sphere2DDataConfig->spectral_modes_m_max;
21     std::size_t N = sphere2DDataConfig->spectral_modes_n_max;
22     std::size_t end = global_offset_idx + local_len;
23
24     // find starting row m for the first global index g
25     std::size_t m = fast_find_row(global_offset_idx, N);
26     // get the exact global index of the start of row m
27     std::size_t row_start = (m * (2*N - m + 3)) >> 1;
28     // starting n = m + offset within row
```

```
29      std::size_t n = m + global_offset_idx - row_start;
30
31      // process until weve consumed local_len items
32      std::size_t pos = 0;
33      while (global_offset_idx < end)
34      {
35
36          std::size_t row_end = row_start + (N - m + 1); // end of row m
37          std::size_t take = std::min(end, row_end) - global_offset_idx; //
        make sure we don't exceed the row end
38
39                  // we work through n elements of this row
40                  // SWEET_THREADING_SPACE_PARALLEL_FOR
41          for (std::size_t k = 0; k < take; ++k)
42              f(n + k, m, buf[pos + k]);
43
44          global_offset_idx += take; // advance global index
45          pos += take; // advance local buffer position
46
47                  // advance to next row
48          row_start = row_end;
49          ++m;
50          n = m; // at row start, n=m
51          if (m > M) break; // safety
52      }
53  }
54  void DataSpectral::spectral_update_lambda(
55                  std::function<void(int,int,TComplex&)> i_lambda
56  )
57  {
58          #if SWEET_MPI
59
60                  int mpi_rank;
61                  int num_mpi_ranks;
62                  MPI_Comm mpi_comm = MPI_COMM_WORLD;
63                  MPI_Comm_rank(mpi_comm, &mpi_rank);
64                  MPI_Comm_size(mpi_comm, &num_mpi_ranks);
65
66                  std::vector<int> counts, displs;
67                  counts.resize(num_mpi_ranks);
68                  displs.resize(num_mpi_ranks);
69                  std::vector<TComplex> local_spectral_space_data;
70
71
```

```
72              int total =
    sphere2DDataConfig->spectral_array_data_number_of_elements;
73              auto chunk_for_rank = [&](int r)->std::pair<int,int>{
74                      int lo = (total * r) / num_mpi_ranks;
75                      int hi = (total * (r+1)) / num_mpi_ranks;
76                      return {lo, hi - lo};
77              };
78
79
80              for (int r = 0; r < num_mpi_ranks; ++r) {
81                      auto [offset, count] = chunk_for_rank(r);
82                      counts[r] = count;
83                      displs[r] = offset;
84              }
85
86              local_spectral_space_data.resize(counts[mpi_rank]);
87
88              std::copy(spectral_space_data + displs[mpi_rank],
89                                      spectral_space_data + displs[mpi_rank] +
    counts[mpi_rank],
90                                      local_spectral_space_data.begin());
91
92              spectral_update_sub(local_spectral_space_data.data(),
    displs[mpi_rank], counts[mpi_rank], i_lambda);
93
94              MPI_Allgatherv(local_spectral_space_data.data(),
    counts[mpi_rank], MPI_C_DOUBLE_COMPLEX,
95
    spectral_space_data, counts.data(), displs.data(),
    MPI_C_DOUBLE_COMPLEX,
96                                                      mpi_comm);
97
98          #else
99
100             SWEET_THREADING_SPACE_PARALLEL_FOR
101             for (int m = 0; m <=
    sphere2DDataConfig->spectral_modes_m_max; m++)
102             {
103                     std::size_t idx =
    sphere2DDataConfig->getArrayIndexByModes(m, m);
104                     for (int n = m; n <=
    sphere2DDataConfig->spectral_modes_n_max; n++)
105                     {
106                             i_lambda(n, m, spectral_space_data[idx]);
```

```
107                                 idx++;
108                             }
109                         }
110
111             #endif
112     }
```

Listing 3.12: spectral_update_lambda() and helper functions

Let us dissect what is happening in this block now. The function follows the non-MPI/MPI sections division, similar to the format used in the previously explained ERK function.

```
int mpi_rank;
int num_mpi_ranks;
MPI_Comm mpi_comm = MPI_COMM_WORLD;
MPI_Comm_rank(mpi_comm, &mpi_rank);
MPI_Comm_size(mpi_comm, &num_mpi_ranks);

std::vector<int> counts, displs;
counts.resize(num_mpi_ranks);
displs.resize(num_mpi_ranks);
std::vector<TComplex> local_spectral_space_data;

int total = sphere2DDataConfig->spectral_array_data_number_of_elements;
auto chunk_for_rank = [&](int r)->std::pair<int,int>
    int lo = (total * r) / num_mpi_ranks;
    int hi = (total * (r+1)) / num_mpi_ranks;
    return lo, hi - lo;
    ;

for (int r = 0; r < num_mpi_ranks; ++r)
    auto [offset, count] = chunk_for_rank(r);
    counts[r] = count;
    displs[r] = offset;


local_spectral_space_data.resize(counts[mpi_rank]);

std::copy(spectral_space_data + displs[mpi_rank],
          spectral_space_data + displs[mpi_rank] + counts[mpi_rank],
          local_spectral_space_data.begin());
```

We begin with a similar MPI preparation we had in the previous code block (3.11).

Each rank retrieves its ID and total process count from the *MPI_COMM_WORLD*, then initializes the *counts* and *displs* arrays to describe how the global spectral array is divided. The total number of spectral coefficients, *total*, is obtained from the configuration object. Using this value, a small lambda function *chunk_for_rank* computes a near-even partition of the array across all ranks by integer division, each rank receiving a certain range of the global array. The resulting *counts* and *displs* arrays therefore store, respectively, the number of spectral coefficients assigned to each rank and the corresponding starting offsets within the global array. We will use these arrays in subsequent logic and later for the *MPI_Allgatherv* operation. After that, we allocate a local array according to the local rank's count, and we copy the portion it is responsible for from *spectral_space_data* into *local_spectral_space_data*.

```
spectral_update_sub(local_spectral_space_data.data(), displs[mpi_rank],
↪  counts[mpi_rank], i_lambda);
```

This call will apply the user-applied update lambda to the local slice of spectral coefficients owned by this rank.

```
void DataSpectral::spectral_update_sub(TComplex *buf,
                                       std::size_t global_offset_idx,
                                       std::size_t local_len,
                                       std::function<void(int,int,TComplex&)> f)
{
    if (local_len <= 0) return;
    std::size_t M = sphere2DDataConfig->spectral_modes_m_max;
    std::size_t N = sphere2DDataConfig->spectral_modes_n_max;
    std::size_t m = fast_find_row(global_offset_idx, N);
    .
    .
    .
}
```

After the data partitioning and local copying step, each MPI rank proceeds to process the subset of spectral coefficients assigned to it. While the coefficients are physically stored as a flat, linear array in memory, they logically represent a two-dimensional triangular domain defined by the spectral indices (n, m). Therefore, the function must translate each linear index into its corresponding pair of spectral coordinates before applying the update.

To perform this translation efficiently, the helper function $fast\_find\_row()$ determines the starting row m for the given global offset determined by the starting index of

a specific rank. What $fast\_find\_row()$ does is use the offset along with the maximum spectral degree N to invert the triangular packing analytically in order to give us which row m a given linear index belongs to in the spectral layout.

```
inline int fast_find_row(std::size_t idx, std::size_t N)
{
    const double B = double(2*N + 3);
    double m0 = std::floor((B - std::sqrt(B*B - 8.0*double(idx))) * 0.5);
    std::size_t m = int(m0);

    // correct by at most one step
    while ( ((m+1) * (2*N - (m+1) + 3) >> 1) <= idx ) ++m;
    while ( (m * (2*N - m + 3) >> 1) > idx ) --m;
    return m;
}
```

Instead of searching row by row, we use a shortcut to find the correct row quickly. First, due to the triangular layout and since for every m, the valid values of n are $n = m$, $m + 1$, ... N, this means that each row's length is $L(m) = N - m + 1$.

To locate the starting point of any row, we need to sum the elements of the preceding rows. In order to that, we use the summation $S(m) = \sum_{k=0}^{m-1} L(k)$. Substituting $L(k) = N - k + 1$ and simplifying the summation, we reach the row start index formula:

$$row\_start\_index(m) = \frac{m(2N - m + 3)}{2}$$

With the above equality, we can also state that the idx we are passing to this function exists between $S(m) \leq idx < S(m + 1)$. Therefore, in order to solve for idx we set the equation

$$idx = \frac{m(2N - m + 3)}{2} \Leftrightarrow m^2 - (2N + 3)m + 2idx = 0$$

With this quadratic equation, now we solve for m,

$$m = \frac{(2N + 3) \pm \sqrt{(2N + 3)^2 - 8idx}}{2}$$

After that, we take the floor of the resultant m (given that it would be a real number and we do not want to overshoot). This is the calculation of $m0$ in the code.

Because we used the floor of a floating-point number, our result may be off by at most one row. To fix it, we invoke two final checks. The first one checks $S(m + 1) \leq idx$, meaning that the next row starts before the index idx, which means we underestimated,

and we increase the row count. However, if our current row start index is above idx, that means we overshot, and we decrease it by 1.

The purpose of this process is to avoid looping over the rows to find the one where idx starts, and instead solve it analytically, resulting in an $O(1)$ search rather than an $O(N)$ row search.

Now that we emphasized how $fast\_find\_row()$ works, let us continue the logic of $spectral\_update\_sub()$.

```
void DataSpectral::spectral_update_sub(TComplex *buf,
                                       std::size_t global_offset_idx,
                                       std::size_t local_len,
                                       std::function<void(int,int,TComplex&)> f)
{
    .
    .
    std::size_t end = global_offset_idx + local_len;
    std::size_t row_start = (m * (2*N - m + 3)) >> 1;
    std::size_t n = m + global_offset_idx - row_start;

    std::size_t pos = 0;
    while (global_offset_idx < end)
    {

        std::size_t row_end = row_start + (N - m + 1);
        std::size_t take = std::min(end, row_end) - global_offset_idx;

        for (std::size_t k = 0; k < take; ++k)
            f(n + k, m, buf[pos + k]);

        global_offset_idx += take;
        pos += take;

        row_start = row_end;
        ++m;
        n = m;
        if (m > M) break;
    }
}
```

After retrieving m, we compute its start in the flattened array, row_start, using the previously derived formula that sums the lengths of all preceding rows $L(k)$. We then obtain the starting column as $n = m + (global\_offset\_idx - row\_start)$, i.e., how far

our offset lies into row m. With (n, m) established, the *while* loop walks through the local slice of spectral coefficients. In each iteration, we compute *row_end* and *take* to determine the number of elements remaining in the current row to process. The inner for loop applies the update lambda $f()$ to those *take* elements, passing the correct spectral indices $(n + k, m)$ and the corresponding value from the local buffer. Afterward, we advance $global\_offset\_idx$ and *pos*, set $row\_start = row\_end$, increment m, and reset $n = m$, thereby moving to the next row. This preserves the triangular (n, m) structure while iterating over a flat array segment.

With this, all major hotspots identified during the profiling stage have been addressed. Through the introduction of MPI-based domain decomposition and targeted updates to the ERK time-stepping and spectral update routines, the solver now supports distributed-memory execution alongside OpenMP's shared-memory parallelism. Each modification was designed to respect SWEET's original data layout and algorithmic structure ensuring functional consistency with the baseline implementation. In the next sections we will study the accuracy and profiling of the implementation.

4 Results and Evaluation

4.1 Accuracy Validation

To verify that the MPI-based domain decomposition preserves numerical correctness, we compare the MPI build against the original serial/OpenMP baseline on the Galewsky setup. For both executables, we keep all the compile and run-time options identical, changing only the output format to CSV (**- -output-file-mode=csv**). In this manner, we get a CSV file specific to a variable (vorticity, divergence, phi perturbation) with its longitude-latitude grid values at a specific timestep (the timestep of the output can be configured at run-time using the $-o$ flag; we set it to 3600-second intervals).

Before the detailed comparison, a soft validation was performed using the standard program log output that reports the minimum and maximum field values at each save step. This information is printed in the binary output mode, returning the minimum and maximum value of each grid and timestep. This output was compared between the original and the MPI-based implementation and found to match.

We then compare the serial and MPI outputs field by field and time by time using a small Python script. The script reads the SWEET grid CSV (skipping header lines) of the original output and the MPI-based output for the same variable at the same time step, and then verifies that the values at the same longitude-latitude index match exactly.

Across all saved times and variables (*div*, *vrt*, *phi_pert*), the MPI outputs matched the baseline exactly. This confirms that the domain decomposition and communication steps do not change the numerical result relative to the original code path.

4.2 Performance Profiling and Analysis

Following the verification of numerical correctness in the previous section, this section evaluates the runtime behavior and scaling characteristics of the MPI-enabled spatial decomposition. The goal of this analysis is to understand how the introduction of distributed-memory parallelism affects the performance of SWEET's shallow-water solver. Measurements were obtained for the Galewsky barotropic-instability benchmark on the LRZ cluster, using identical numerical settings across the serial, OpenMP, and

MPI versions.

All performance experiments were conducted on the LRZ computing cluster using the Galewsky benchmark. The same numerical parameters were used for the serial/OpenMP and MPI configurations to ensure comparability. The simulations were performed at a spectral truncation of T255, corresponding to both spherical harmonic degrees and orders of 255. This spectral resolution will correspond to a physical resolution of 768 × 384 grid points. Each run integrated the system over a total simulated time of 691200 seconds (8 days) using a fixed time step of 150 seconds, with model outputs written every 3600 seconds. A more in-depth description of the complete set of parameters and system specifications is provided in Appendix A.

To understand how the existing spatial discretization behaves under shared-memory parallelism, we profile the OpenMP-enabled SWEET solver. By varying the number of OpenMP threads, we examine how different parts of the spatial backend scale, which operations remain serial, and where domain decomposition (MPI) could be introduced to improve scalability.

We profile while sweeping the OpenMP thread count via setting the environment variable $OMP\_NUM\_THREADS$

$$\texttt{OMP\_NUM\_THREADS} \in \{1, 2, 4, 6, 8, 16, 32\}$$

For each setting, we record the wall-clock runtime from the program output and collect VTune data exported in CSV format. The Galewsky setup and all executable flags match the baseline configuration of the previous section.

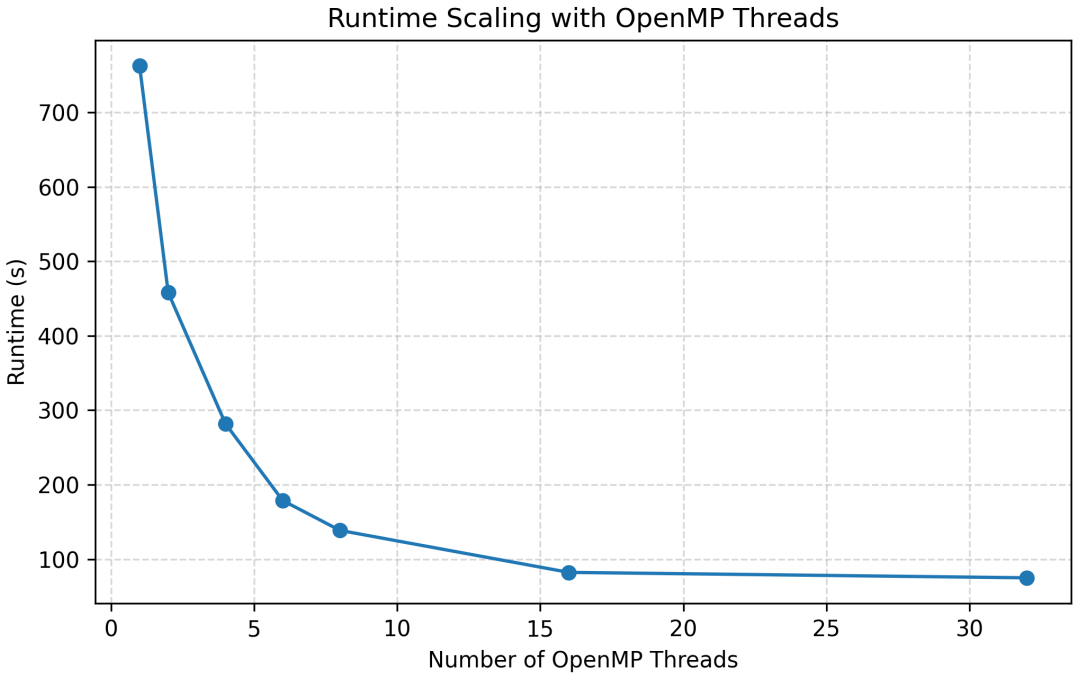


Figure 4.1: Runtime scaling of the original SWEET shallow-water solver under OpenMP parallelism.

The results show an almost exponential decrease in runtime up to eight threads, beyond which improvements start to taper off. At 32 threads, the total speedup reaches roughly tenfold relative to the serial case. This behavior shows that OpenMP effectively accelerates computations that are supported.

To evaluate the effect of distributed-memory parallelization, the SWEET solver was executed using a hybrid MPI–OpenMP configuration. In these experiments, each MPI rank launched a fixed number of OpenMP threads, and the total number of ranks was varied to distribute the spherical domain across multiple processes. The same Galewsky setup and numerical parameters as in the OpenMP-only tests were used. We first examine the effect of introducing distributed-memory parallelism by fixing the number of OpenMP threads per rank to eight and varying the number of MPI ranks from one to eight. This configuration enables us to evaluate how the MPI domain decomposition affects performance when each rank utilizes all available thread-level parallelism on its subdomain.

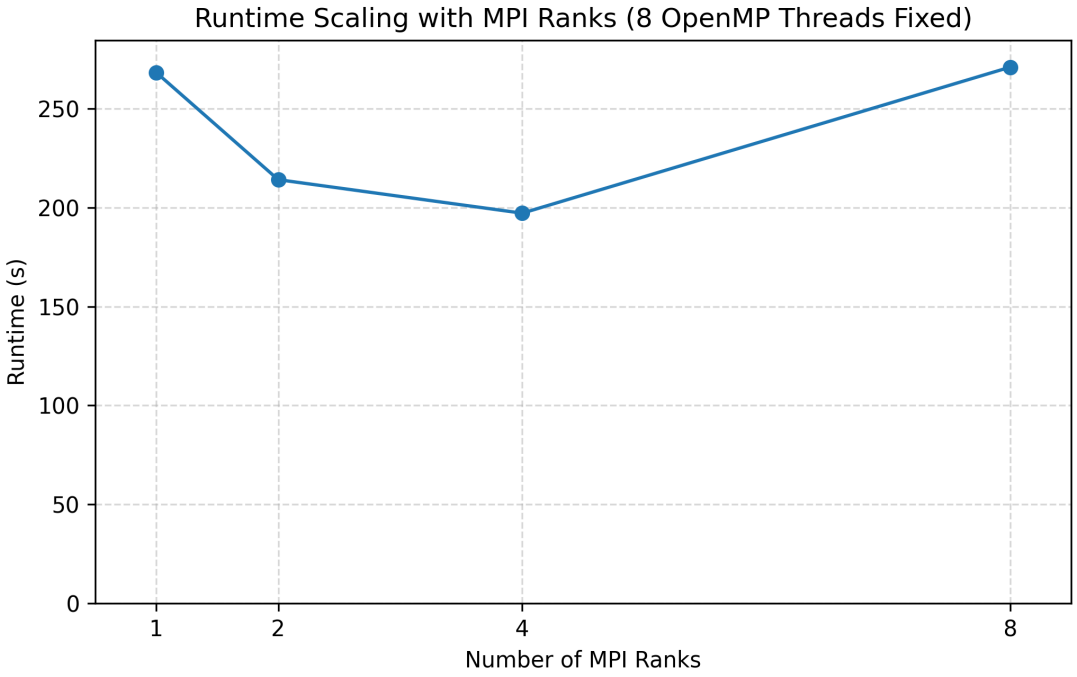


Figure 4.2: Runtime scaling of the MPI-based SWEET with fixed 8 OpenMP Threads.

While the expectation was that distributing computations would accelerate execution, the MPI-based version exhibited longer run times than the baseline versions. In our measurements, the OpenMP-only solver at eight threads (Figure 4.1) achieved an elapsed time of 138.6 s. When the same configuration was extended with MPI-based domain decomposition (Figure 4.2), the total runtime did not improve. The hybrid MPI–OpenMP runs yielded elapsed times of 268.2 s, 214.0 s, 197.1 s, and 270.9 s for one, two, four, and eight ranks, respectively. Although a slight reduction is observed up to four ranks, the overall performance remains below that of the OpenMP baseline. This confirms that introducing distributed-memory communication within the same computational workload incurs additional synchronization and data-exchange costs that offset the potential benefits of parallelization.

The results indicate the benchmark, at the standard spatial resolution, does not generate sufficient per-rank workload to amortize the MPI overheads. Moreover, the spherical harmonic transforms, which are responsible for the solver's runtime, are not distributed across ranks, resulting in each process performing redundant computations. Consequently, the hybrid parallelization offers no measurable improvement over the shared-memory execution.

To further clarify this behavior, we next compare the hybrid MPI-OpenMP configurations with the OpenMP-only baseline at equivalent total core counts.

Layout	MPI Ranks	Threads per Rank	Runtime [s]
Original	1	8	**138.6**
MPI-Based	2	4	315.7
MPI-Based	4	2	443.7
MPI-Based	8	1	717.1

Table 4.1: Runtime comparison of eight-core OpenMP and MPI–OpenMP configurations.

Table 4.1 compares the runtime of all eight-core layouts. The OpenMP-only solver clearly outperforms every hybrid configuration, completing in 138.6 s compared with 315–717 s for the MPI–OpenMP variants. As the number of ranks increases, the runtime grows substantially due to additional communication and synchronization phases. This indicates that introducing MPI parallelism yields no benefit and can even degrade performance. Furthermore, within the hybrid MPI–OpenMP configurations, it is evident that performance improves as the number of OpenMP threads per rank increases and the number of MPI ranks decreases. The configuration with two ranks and four threads per rank performs substantially better than the eight-rank, single-thread setup. This indicates that the solver benefits more from thread-level parallelism than from fine-grained domain decomposition, suggesting that the OpenMP layer maintains better scaling efficiency than the MPI layer.

Profiling results obtained from Intel VTune further clarify why the OpenMP-only configuration performs best. As shown in Table 4.2, the highest time-consuming individual functions according to the VTune bottom-up view belong to the SHTns spherical harmonic transform routines.

Routine	Effective Time (%)
_an22_l	7.7
split_north_south_real_imag	4.1
n1_12	3.7
n1_4	2.9
_sy1_hi2_l	2.9
_sy2_hi2_l	2.7
_an12_l	2.1
t1sv_2	1.9

Table 4.2: Top runtime-contributing functions in the 8×8 configuration (VTune Hotspots).

Although the names are ambiguous, the tracing of those library calls belongs to the SHTNs transforms. Although no single kernel exceeds ten percent of the total runtime within a single process, these SHTNs-related routines collectively account for around a quarter of the per-rank execution time. When considering that the computation is redundantly performed by all ranks, we would conclude that adding MPI processes does not reduce the wall-clock cost of the transform stage; instead, it increases the total CPU work (aggregate core-seconds) and introduces additional synchronization and communication, which in practice keeps runtime flat or makes it longer despite any increase in computational resources. The SHTns kernels remain the principal bottleneck because they do not benefit from domain decomposition and cannot be parallelized across ranks; instead, each process performs identical spectral transforms on a full grid copy.

Moreover, this observation aligns with the earlier runtime measurements. The OpenMP-only configuration achieves the best performance because its thread-level parallelization efficiently accelerates the local spectral transforms without introducing inter-process synchronization or data replication. In contrast, the hybrid and MPI-heavy configurations expend additional time performing redundant SHTns operations and managing inter-rank communication overhead, resulting in the domination of total execution time and limiting the scalability in the hybrid solver. Although the domain decomposition driven optimizations yield speedups within individual physical and spectral grids operations, these improvements diminish when considering the entire solver workflow. The cumulative overhead of the spectral transform routines explained previously and the global gather operations we will discuss outweighs the localized savings, leading to only minor net reductions in total runtime.

The next performance limiter observed arises from the *MPI_Allgatherv* operation,

which is responsible for exchanging prognostic fields between ranks through each timestep. This collective introduces a significant synchronization phase because each process must receive the full grid data from all others before proceeding, which is compounded by the fact that it is used multiple times for multiple grids. We zoom in on the cost of these gathers through VTune's MPI Busy Wait Time figures expressed in the grid below.

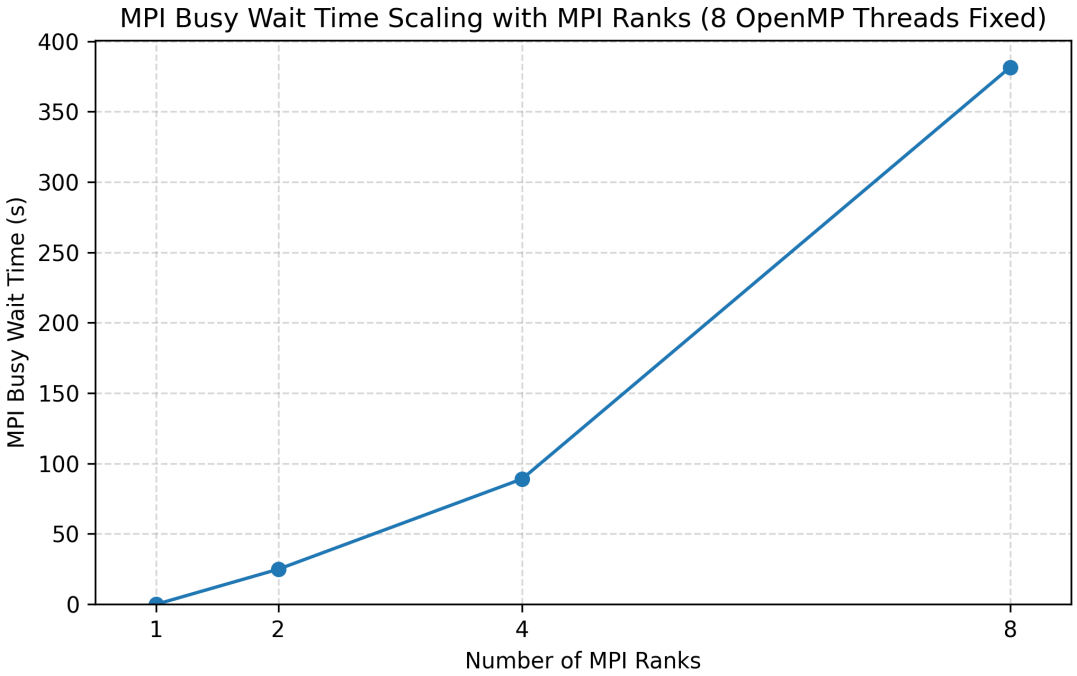


Figure 4.3: MPI Busy Wait Time scaling of the MPI-based SWEET with fixed 8 OpenMP Threads.

In VTune terminology, MPI Busy Wait Time represents the duration during which a process is actively spinning within the MPI library while waiting for communication to complete. Although all the ranks perform the same amount of local computation, the busy-wait time increases with the number of ranks because of the collective communication pattern employed by the solver. In the current implementation, each process participates in an *MPI_Allgatherv* operation that exchanges the entire global grid rather than only local subdomains. Consequently, the amount of data transferred per rank does not increase with domain decomposition, while the number of communicating peers and synchronization points grows.

The monotonic rise the Busy Wait Time in Figure 4.3 confirms that communication and synchronization overheads increase substantially with rank count. Since the benchmark problem size is fixed, this additional time cannot originate from computation and therefore represents pure communication cost introduced by the global field exchange.

Adding more MPI ranks fragments the grid into smaller local domains, but the solver's Allgather design requires that each rank ultimately receives the complete global grid. This creates an all-to-all communication pattern that increases in time complexity as the number of ranks increases. Consequently, the benefits of smaller per-rank subdomains is offset by the surge in communication volume and synchronization delays. Moreover, collective operations complete only once all ranks have finished exchanging data, so timing variations that could arise due to the network contention of exchanging full physical and spectral grids in between multiple ranks can further contribute to this wait time.

It is worth noting that experimenting with gathering the data to the main rank, executing the spectral transforms, and redistributing the data was performed in order to see if we can lower the exchange overhead. However, this version only preserved or worsened the execution time since the main process became a serialization bottleneck during both the collection and redistribution phases. Concentrating the communication on a single rank eliminated the concurrent data exchange of the Allgather but introduced a sequential gather–scatter pattern that saturated the main rank's memory bandwidth and network links. In addition, the SHTns transforms that follow the communication phase still require the full global field on every process, forcing the data to be broadcast again after the transform. As a result, the total communication volume remained effectively unchanged, and the additional serialization led to higher overall latency. These experiments confirmed that the communication overhead is not solely a consequence of the collective implementation but is fundamentally tied to the global data dependency of the spectral transform routines. Without a distributed version of the SHTns transforms that can operate on partitioned spectral data, any attempt to reduce the communication frequency merely shifts the bottleneck without improving scalability.

Finally, as evidenced by the code written and the performance measurements, the solver invokes the Allgather mechanism whenever full global grids are required, which occurs multiple times within a single time step. These gathers take place within the main time-stepping routine (*euler_timestep_update_pert*) and inside the *spectral_update_lambda* function. This intermittent communication, composed of repeated gathers before and after the global spectral routines, is a major contributor to the degraded performance, alongside the inherent computational bottleneck of the spherical harmonic transforms themselves. In an ideal implementation, where the

SHTns library supported distributed spectral transforms, it would not be necessary to assemble the full grids prior to these routines. Instead, the solver could follow a more efficient scatter–compute–gather pattern, in which each rank performs computations on its local subdomain and only the final results are collected by a designated root process for domain merging. This would eliminate the repeated all-to-all communication phases and significantly reduce synchronization overhead.

4.3 Discussion and Future Work

The analysis of the hybrid parallel implementation provides several insights into the current performance limitations and the necessary steps for improving scalability. The results show that while shared-memory parallelization through OpenMP is effective in accelerating local computations, extending parallelism to the distributed-memory layer via MPI does not yield additional benefits. Instead, the MPI layer introduces synchronization and communication overhead that outweighs the computational gains, leading to longer tunrimes even when more total cores are employed.

A primary factor limiting scalability is the reliance on shared-memory global spectral transforms. The SHTns routines dominate the computational workload but cannot be distributed across ranks without modifying the source code of the library or replacing it with a version that supports MPI. While most of the solver's computations, such as the physical-space updates and flux calculations, were successfully distributed across MPI ranks, the spherical harmonic transform routines could not be parallelized in the same way. Consequently, each rank redundantly executes identical transform computations, increasing total CPU work without reducing wall-clock time. This design makes the solver inherently communication-bound rather than compute-bound, with total runtime largely determined by data movement and synchronization costs rather than arithmetic throughput.

The use of the *Allgather* mechanism compounds this limitation. Because each rank must receive the entire global grid before and after every forward or inverse transform operation, the communication pattern scales poorly with increasing process count. VTune profiling revealed that the MPI Busy Wait Time in *MPI_Allgatherv* grows sharply with the number of ranks, confirming that synchronization imbalance and collective communication dominate execution time. Furthermore, OpenMP acceleration, while improving local performance, exposes these communication bottlenecks even more strongly as it places the bottleneck further on communication, as it finishes computation before the non-blocking gathers finish exchange. Together, these effects suggest that the current hybrid strategy is unable to achieve strong scaling beyond a single shared-memory node.

Future development should therefore focus on reducing or eliminating the solver's dependence on global data exchanges. The most impactful improvement would be to introduce a distributed spectral transform capability, either by extending SHTns or integrating an alternative library that supports domain-decomposed spherical harmonics. Such an enhancement would enable each MPI rank to operate only on a portion of the spectral domain, removing the need for repeated full-grid gathers and substantially lowering communication volume.

Distributed spherical transforms would then enable us to restructure the solver from intermittent communication points to a proper scatter-compute-gather workflow that could eliminate redundant synchronization. Rather than performing full-grid gathers before each transform, subdomains could be exchanged only once per timestep or at defined coupling points, ideally while maintaining the current non-blocking collective nature to overlap communication and computation.

Ultimately, this work demonstrates that achieving scalable hybrid performance requires addressing the algorithmic dependencies between computation and communication. Thread-level parallelism alone provides limited gains when global synchronization dominates execution. Moving toward a distributed spectral framework and, in turn, a more efficient communication model would enable the solver to scale efficiently across nodes, forming the basis for a next-generation, high-performance spatial discretization within the shallow-water equation model.

Abbreviations

DCMIP2016 Dynamical Core Model Intercomparison Project 2016
ECMWF European Centre for Medium-Range Weather Forecasts
FFT Fast Fourier Transform
FFTW Fastest Fourier Transform in the West
LAPACK Linear Algebra Package
LRZ Leibniz Supercomputing Centre
MPI Message Passing Interface
NCAR National Center for Atmospheric Research
ODE Ordinary Differential Equation
OpenMP Open Multi-Processing
PDE Partial Differential Equation
PFASST Parallel Full Approximation Scheme in Space and Time
Parareal Parallel-in-time algorithm "Parareal"
REXI Rational Approximation of Exponential Integrators
RK Runge–Kutta
SCons Software Construction tool (build system)
SDC Spectral Deferred Correction
SHTns Spherical Harmonic Transform library
SWE Shallow Water Equations
SWEET Shallow Water Equations Environment for Tests
VTune Intel VTune Profiler
XBraid Multigrid-in-time framework

List of Figures

List of Tables

Bibliography

[1] K. Atkinson and W. Han. *Spherical Harmonics and Approximations on the Unit Sphere: An Introduction*. Vol. 2044. Lecture Notes in Mathematics. Springer, 2012. DOI: 10.1007/978-3-642-25983-8.

[2] W. Bourke. "An Efficient, One-Level, Primitive-Equation Spectral Model." In: *Monthly Weather Review* 100 (1972), pp. 683–689. DOI: 10.1175/1520-0493(1972)100<0683:AEOPSM>2.3.CO;2.

[3] Clay Mathematics Institute. *Navier–Stokes Equation (Millennium Prize Problem)*. 2022.

[4] J. B. Drake and D. X. Guo. *A Vorticity–Divergence Global Semi-Lagrangian Spectral Model for the Shallow Water Equations*. Tech. rep. ORNL/TM-2001/216. Oak Ridge, TN: Oak Ridge National Laboratory, 2001.

[5] ECMWF. "IFS Documentation CY47R1 - Part III: Dynamics and Numerical Procedures." eng. In: 3. ECMWF, 2020 2020. DOI: 10.21957/u8ssd58.

[6] J. Galewsky, R. K. Scott, and L. M. Polvani. "An initial-value problem for testing numerical models of the global shallow-water equations." In: *Tellus A: Dynamic Meteorology and Oceanography* 56.5 (2004), pp. 429–440. DOI: 10.1111/j.1600-0870.2004.00071.x.

[7] H. E. Haber. *The Spherical Harmonics*. Lecture notes, UC Santa Cruz. 2012.

[8] J. J. Hack and R. Jakob. *Description of a Global Shallow Water Model Based on the Spectral Transform Method*. Tech. rep. NCAR/TN-343+STR. Boulder, CO: National Center for Atmospheric Research, 1992.

[9] J. R. Holton. *An Introduction to Dynamic Meteorology*. 4th. Amsterdam: Elsevier Academic Press, 2004. ISBN: 0-12-354015-1.

[10] P. K. Kundu, I. M. Cohen, and D. R. Dowling. *Fluid Mechanics*. 6th. Amsterdam: Academic Press, 2015. ISBN: 9780124059351.

[11] J. D. McEwen and Y. Wiaux. "A novel sampling theorem on the sphere." In: *IEEE Transactions on Signal Processing* 59.12 (2011), pp. 5876–5887. DOI: 10.1109/TSP.2011.2166394.

[12] D. McLean. *Understanding Aerodynamics: Arguing from the Real Physics*. Print ISBN: 978-1-119-96751-4; Online ISBN: 978-1-118-45419-0. Chichester: John Wiley & Sons, 2012. ISBN: 9781119967514. DOI: 10.1002/9781118454190.

[13] NOAA Geophysical Fluid Dynamics Laboratory. *The Shallow Water Equations*. https://www.gfdl.noaa.gov/wp-content/uploads/files/user_files/pjp/shallow.pdf. Technical note with derivation and spherical-form SWE.

[14] R. L. Parker. *Spherical Harmonics (SIO 239 Notes)*. Lecture notes, UC San Diego. 2003.

[15] P. S. Peixoto and M. Schreiber. "Semi-Lagrangian Exponential Integration with Application to the Rotating Shallow Water Equations." In: *SIAM Journal on Scientific Computing* 41.5 (2019), B903–B928. DOI: 10.1137/18M1206497. eprint: https://doi.org/10.1137/18M1206497.

[16] N. Schaeffer. "Efficient Spherical Harmonic Transforms aimed at pseudo-spectral numerical simulations." In: *arXiv preprint* arXiv:1202.6522v5 (2015). physics.comp-ph.

[17] N. Schaeffer. *Vector Spherical Harmonics as implemented in SHTns*. https://nschaeff.bitbucket.io/shtns/vsh.html.

[18] M. Schreiber. *Shallow Water Equations on the Sphere: Formulations in SWEET*. Tech. rep. Documentation PDF in the SWEET repository; last updated 13 July 2023. SWEET Project, Inria, 2023.

[19] M. Schreiber and S. contributors. *SWEET — Shallow Water Equation Environment for Tests*. https://sweet.gitlabpages.inria.fr/sweet-www/. 2025.

[20] E. Süli. *A Brief Introduction to the Numerical Analysis of PDEs*. Lecture notes, University of Oxford. 2018.

[21] C. Terquem. *Fluids*. Lecture notes, Department of Physics, University of Oxford. Third Year, Part B1, Hilary Term 2021. 2021.

[22] P. A. Ullrich, C. Jablonowski, J. Kent, P. H. Lauritzen, R. Nair, K. A. Reed, C. M. Zarzycki, D. M. Hall, et al. "DCMIP2016: a review of non-hydrostatic dynamical core design and intercomparison of participating models." In: *Geoscientific Model Development* 10 (2017), pp. 4477–4509. DOI: 10.5194/gmd-10-4477-2017.

[23] G. K. Vallis. *Atmospheric and Oceanic Fluid Dynamics: Fundamentals and Large-Scale Circulation*. 2nd ed. Cambridge: Cambridge University Press, 2017.

[24] C. B. Vreugdenhil. *Numerical Methods for Shallow-Water Flow*. Water Science and Technology Library. Dordrecht: Springer, 1994. ISBN: 978-0-7923-3164-3.

[25] A. T. Weaver and S. Ricci. *Constructing a background-error correlation model using generalized diffusion operators*. Tech. rep. ECMWF Technical Memorandum 371. ECMWF, 2003.

[26] N. P. Wedi. *The Spectral Transform Method (ECMWF lecture slides)*. ECMWF training slides. 2015.

[27] D. L. Williamson. "The Evolution of Dynamical Cores for Global Atmospheric Models." In: *Journal of the Meteorological Society of Japan. Ser. II* 85B (2007), pp. 241–269. ISSN: 0026-1165. DOI: 10.2151/jmsj.85B.241.

[28] D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, and P. N. Swarztrauber. "A Standard Test Set for Numerical Approximations to the Shallow Water Equations in Spherical Geometry." In: *Journal of Computational Physics* 102.1 (1992), pp. 211–224. DOI: 10.1016/S0021-9991(05)80016-6.

Appendix

Appendix A - Detailed Experimental Setup

All experiments and performance measurements were conducted on the Leibniz Supercomputing Centre (LRZ) cluster using the CoolMUC-4 system, specifically the *cm*4*_inter* partition of the SuperMUC-NG Linux cluster. Each compute node in this partition is equipped with Intel Zeon Platinum 8480+ (Sapphire Rapids) processors, providing 112 physical cores and approximately 488 GiB of usable main memory per node. The $cm4_inter$ partition is intended for interactive and short test jobs, allowing allocations of 1 to 4 nodes, 1 to 112 cores per job, and a maximum runtime of 8 hours. Job allocations were performed interactively using the LRZ-recommended *salloc* command. For the original configuration that relies solely on Open-MP support, job resources were requested with

```
salloc -M inter -p cm4_inter -N 1 --ntasks=1 --cpus-per-task=64 -t 02:00:00
```

This allocation reserves one compute node and provides 64 CPU cores to a single task, allowing for a varied number of Open-MP threads (up to 64) to test with.

The baseline run is carried out using the same setup and parameters as the Galewsky benchmark included in the SWEET repository. Those conditions are specified in the file *sweet/tutorials/galewsky/galewsky_benchmark/*1*_benchmark_create_jobs.py*. As discussed before, this benchmark is a widely used validation and performance test for the shallow water equations on the sphere.

To produce a reproducible binary for profiling, I compile SWEET with SCons, the project's build system. SCons drives a parameterized build of the spherical shallow-water solver (*programs/PDE_SWESphere*2*D*) and records build options in the program name for traceability. The compilation command used to generate an executable follows

```
scons --program=programs/PDE_SWESphere2D --mode=debug --debug-symbols=disable
↪ --simd=enable --fortran-source=disable --lapack=disable
↪ --program-binary-name= --sweet-mpi=disable --threading=omp
↪ --rexi-thread-parallel-sum=disable --benchmark-timings=enable
↪ --rexi-timings-additional-barriers=disable --rexi-allreduce=disable
↪ --parareal=none --parareal-scalar=disable --parareal-cart2d=disable
↪ --parareal-sphere2d=disable --parareal-cart2d-swe=disable
↪ --parareal-cart2d-burgers=disable --xbraid=none --xbraid-scalar=disable
↪ --xbraid-cart2d=disable --xbraid-sphere2d=disable
↪ --xbraid-cart2d-swe=disable --xbraid-cart2d-burgers=disable
↪ --libpfasst=disable --eigen=disable --libfft=enable --libsph=enable
↪ --mkl=disable --cart2d-spectral-space=disable
↪ --cart2d-spectral-dealiasing=disable --sphere2d-spectral-space=enable
↪ --sphere2d-spectral-dealiasing=enable --libxml=disable --gui=disable
↪ --quadmath=disable
```

Without going into too much detail, the chosen configuration reflects an attempt to keep the build reproducible, lightweight, and directly aligned with the profiling objectives. Most of the optional flags simply disable auxiliary frameworks (such as Parareal, XBraid, PFASST, GUI, or XML) and numerical libraries (MKL, LAPACK, Fortran sources) that are not required for this study. This ensures that the resulting executable contains only the components necessary for the Galwesky benchmark and that any performance variations observed originate from the core solver rather than unrelated dependencies.

Among the relevant compile-time options, the flag $--mode=debug$ produces a debug build of the solver that retains symbol information, allowing profiling tools such as Intel VTune Profiler or Valgrind to accurately resolve function names, stack traces, and source lines.

Similarly, the flag $--sweet-mpi=disable$ ensures that only shared-memory parallelism via OpenMP is active (coupled with the flag $--threading=omp$). When profiling an MPI-based version of the solver, we set this flag such that $--sweet-mpi=enable$.

Finally, $--libsph=enable$ and $--libfft=enable$ ensure that the SHTns and FFTW libraries are used to enable the spectral-transform backends.

After an executable is generated, we can run it with the following flags,

```
./PDE_SWESphere2D_COMP_spspec_spdeal_fft_benchtime_mpi_thomp_debug -v 0 -d 12
↪ -M 256 -N -1 --benchmark-name=galewsky --instability-checks=1
↪ --pde-viscosity=0.0 --timestepping-method=ln_erk --timestepping-order=4
↪ --timestepping-order2=4 --dt=150.0 -t 691200 -o 3600
↪ --output-file-mode=bin
```

In the above command, the values assigned from the parameters are taken from the previously mentioned Galewsky job generation file; the crucial ones include:

- $--benchmark-name=galewsky$ specifies which benchmark SWEET will be run against
- $-M = 256$ sets the spectral truncation, i.e., the maximum total degrees (N) and orders (M) such that they equate 255. Setting $-N = -1$ as well instructs the SWEET framework to compute a compatible Gaussian grid, ensuring that the spatial discretization matches the previously defined spectral truncation.
- $-t = 691200$ specifies the total simulation duration in seconds, equivalent to eight days of simulated time, while $-o = 3600$ outputs data every hour.

The previous process outlines a normal execution of SWEET; however, in the profiling section, we use the Intel Vtune profiler to benchmark the original implementation by using the *hotspots* mode it supports. Concretely, runs are performed with

```
vtune -q -collect hotspots -knob sampling-mode=sw   -result-dir vtune_results
↪ --   ./PDE_SWESphere2D_COMP_spspec_spdeal_fft_benchtime_thomp_debug   -v 0
↪ -d 12 -M 256 -N -1 --benchmark-name=galewsky --instability-checks=1
↪ --pde-viscosity=0.0 --timestepping-method=ln_erk --timestepping-order=4
↪ --timestepping-order2=4 --dt=150.0 -t 691200 -o 3600
↪ --output-file-mode=bin
```