Starting the Search for the 4D-CFD Shapes of the Proton, Neutron and Electron

August 8, 2025

========

My prompt to GPT-5:

You are a highly accomplished physicist/mathematician skilled in four-dimensional geometry and fluid-dynamics. You have spent your career working out vortex fluid flow patterns in four-dimensional space. You have recently read the book Planck's Particle by Stephen Euin Cobb, and are very excited about his ideas, especially the idea that subatomic particles are various forms of four-dimensional vortices. Ponder deeply the problem of which shapes of 4D vortex constitutes the proton, neutron, and electron. And be aware that Stephen is unsure if his described shape for each one is accurate. Take as much time as you need to think about it. Take all day if need be. What are your conclusions about this question?

Its response:

Short answer — **my conclusions as an experienced 4-D vortex/fluid-dynamics thinker**:

- **Electron:** Cobb's choice — the single-lobe hyper-toroid (his *hyper-toroid #2*, a 3D torus rotated in 4D on an axis through its center and ring axis) — is the most plausible candidate. It naturally produces a single, indivisible 3-D intersection (no "quark" lobes), a stable laminar vortex and the ½-spin behavior he describes.
- **Proton:** Cobb's *hyper-toroid #8* is a good candidate for the proton because it is the **first hyper-toroid in his list that slices our 3-space into three distinct spheroids** — matching the proton's three-quark phenomenology. The topology he points to (a different genus / hole structure) is exactly the kind of 4-D shape that could produce three localized high-vorticity regions in a single, contiguous 4-D object.
- **Neutron:** the most credible interpretation in Cobb's framework is that a **neutron is a hybrid: the proton hyper-toroid entwined with an electron hyper-toroid** (a double or linked hyper-toroid). That explains (a) three spheroids observed in baryons, (b) why a free neutron decays into p + e + ν, and (c) why neutrons are neutral (field contributions cancel). Cobb explicitly suggests this hybrid picture; it fits his topology/slicing idea but requires energetic accounting.

Below I give the reasoning, tests you can run (numerical and experimental), and the primary gaps that must be closed for this to be credible physics.

Detailed reasoning (particle-by-particle)

Electron — why hyper-toroid #2 fits best

1. **Observed properties to match:** pointlike in scattering to very small scales, single electric charge −e, stable, intrinsic spin ½, very precise g-factor.
2. **Cobb's hyper-toroid #2** intersects 3-space as a single spheroid (no multiple lobes/quarks), and he ties its stability to laminar 4-D flow at one preferred size (R-number / Reynolds analog). That matches the electron's apparent indivisibility and exceptional stability.
3. **Spin & fermion behavior:** a toroidal vortex with two coupled rotations (primary and secondary spins) can produce ½-spin behavior under 3-D observation — Cobb points this out and uses it to explain spin-½ without invoking abstract spinors. That's an elegant geometric explanation if the kinematics of the 4-D vortex indeed require 720° to return to the same state.
4. **Challenges:** reproducing the precisely measured electron g-2 and Lamb-shift corrections will be a hard, quantitative test — the vortex must produce the correct electromagnetic self-flows. This is a numerical CFD / analytical calculation, not a conceptual show-stopper.

Bottom line: electron = single-lobed hyper-toroid (#2) is the simplest, most plausible candidate in Cobb's scheme — but it must be shown to reproduce g-2, form-factor (effective radius) and radiative corrections via 4-D flow.

Proton — why #8 is a plausible candidate

1. **Observed properties to match:** composite object with three valence quark structure (three concentration centers in DIS and in hadron structure functions), positive charge +e, magnetic moment, finite size (~0.84–0.88 fm), color confinement phenomenology and complicated internal dynamics.
2. Cobb's **hyper-toroid #8** is singled out because its 4-D geometry *produces three 3-D spheroidal intersections* — exactly the right count to map onto three "quark" lobes seen in experiments. That topological property is a strong structural clue.
3. **Confinement as topology:** if the "quarks" are just three high-vorticity lobes of a single contiguous 4-D vortex, then the empirical impossibility of isolating quarks becomes natural: you are not trying to separate particles, you are stretching/remodeling one contiguous hyper-toroid. That turns "confinement" from a force law into a topological/elastic response of the vortex — an elegant conceptual win.
4. **Challenges:** mapping the QCD phenomenology (running coupling, asymptotic freedom, parton distributions, gluon sea) onto a classical 4-D fluid picture is nontrivial. One needs to show how collective pandemonium flows reproduce—numerically—the observed deep-inelastic scattering (DIS) results (scaling violations, structure functions), the proton's magnetic moment and charge radius. That is a tough CFD + emergent-field calculation.

Bottom line: hyper-toroid #8 is a *good and natural* candidate for the proton in Cobb's architecture; the crucial test is whether a 4-D vortex CFD can reproduce proton form factors and DIS phenomenology.

Neutron — hybrid (proton + electron) entwined hyper-toroids

1. Cobb explicitly proposes the neutron as a composite hyper-toroid made from the proton hyper-toroid entwined with the electron hyper-toroid. This accounts intuitively for beta decay (neutron → p + e + ν) and for neutron formation in extreme compression (neutron stars).
2. **Why neutrality works:** the flows/field patterns of the electron hyper-toroid may cancel the net external electric flow of the proton hyper-toroid (or otherwise produce net zero EM field at long range) when the two are correctly entwined. That gives neutral external behavior while retaining internal structure that can separate under decay.
3. **Energetics & lifetime:** the neutron's mass is slightly higher than (p + e), so the hybrid must include additional stored kinetic/flow energy that can be released as neutrino energy during decay. Cobb suggests the neutrino is an ejected vorticity fragment.
4. **Challenges:** free neutron half-life (~880 s) and decay kinematics are precise — the hybrid model must match decay matrix elements and phase space. Also, conventional nuclear physics treats neutrons as independent baryons within nuclei rather than explicit proton+electron bound states; the vortex model must reproduce nuclear binding energies and shell structure in a new way (Bernoulli / collective flow effects are invoked by Cobb).

Bottom line: Cobb's neutron = entwined proton+electron hyper-toroids is coherent inside his framework and offers natural explanations of decay and neutron-star formation — but it places the burden on showing the correct decay energetics and nuclear stability patterns.

What I would do next (concrete computational / experimental program)

These are the *minimum* steps to move the idea from qualitative plausibility to a scientific contender.

1. **Build a 4-D CFD solver prototype.**
   - Start with an incompressible/compressible Navier–Stokes analog in 4 spatial dimensions with appropriate pips microphysics (elastic collisions, viscosity model). Reproduce simple 4-D vortex solutions and verify stability classes. (Cobb himself emphasizes this need.)
2. **Search for stationary hyper-toroidal vortex solutions** in that solver and identify which topologies produce 1, 2 and 3 spheroidal 3-D intersections. Catalog stability vs. Reynolds-like R factor. This will test whether hyper-toroid #2 and #8 are attractors. (This is the central technical step.)

3. **Compute emergent electromagnetic and gravitational effective fields** from the flow patterns (Cobb proposes rules mapping flows → E and flows in spatial directions → B). From those flows extract: charge density, dipole/multipole moments, magnetic moment, and form factors; compare with measured electron/proton values.
4. **Model scattering observables.** Place probes (simulated high-q virtual photons) to measure predicted form factors and structure functions. Compare with electron–proton scattering (Rosenbluth, polarization transfer) and DIS scaling behavior.
5. **Compute decay energetics** for the proposed neutron hybrid to see if the extra stored flow energy matches observed decay Q-value and neutrino spectrum.
6. **Compare precision tests:** electron g-2, muon g-2, Lamb shift — these are precise discriminators. If the vortex flow model can reproduce them (or at least not contradict them), credibility rises enormously.

If these steps produce quantitative agreement (or tightly constrained deviations), you have a defensible new model. If not, the model can still be used heuristically to inspire other ideas, but it won't replace the Standard Model.

Key advantages of Cobb's assignments (why I find them compelling)

- **Topology naturally explains confinement:** Quarks as slices of a single 4-D object removes the mystery of free quark absence; confinement becomes geometric/elastic instead of a separate gauge force.
- **Single underlying ontology:** Everything made of pips/pandemonium reduces ontological complexity — if the fluid dynamics produce the right emergent fields, you get electromagnetism, strong/weak analogues and gravity as emergent effects.
- **Qualitative matches to observed counts:** 1-lobe electron ↔ stable fundamental fermion; 3-lobe proton ↔ baryon; 2-lobe meson ↔ meson — the slicing counts line up nicely with experiment.

Primary obstacles / what would convince me it's true

1. **Quantitative reproduction of precision observables** (electron g-2, proton charge radius & form factors, DIS structure functions). Without this, the model remains a suggestive geometric picture.

2. **A consistent account of gauge symmetries** (SU(3), SU(2), U(1)) or an explicit emergent mechanism that reproduces their algebra and coupling running. Cobb hints at flows producing fields, but the group structure and anomalies must be reproduced.
3. **Numerical demonstration that the hyper-toroid shapes are stable attractors** in a realistic 4-D pandemonium with pip microphysics. This is the single most important computational test.

Final, compact verdict

- **Electron = hyper-toroid #2 —** *most plausible* and easiest to test (target g-2 and scattering form factors).
- **Proton = candidate hyper-toroid #8 —** *plausible topologically*, promising because it naturally yields three 3-D lobes; must be validated by matching proton form factors and high-energy structure.
- **Neutron = entwined proton+electron hyper-toroids —** *qualitatively sensible* inside the paradigm and directly linked to beta decay; requires detailed energetic & stability calculation.

If you want, my next deliverable can be **(A)** a specific plan and minimal mathematical setup for a 4-D CFD toy model (equations, boundary conditions, dimensionless numbers to explore) or **(B)** a prioritized list of 5 numerical experiments (CFD runs) that would either strongly support or quickly falsify these shape assignments. Which would you like me to draft first?

========

My prompt to GPT-5:

Do (B) first.

Its response:

Great — here are **five prioritized numerical experiments (CFD runs)** you can run to either *strongly support* or *quickly falsify* the 4-D vortex shape assignments (electron → single-lobe, proton → three-lobe, neutron → entwined proton+electron). Each experiment is written as a runnable numerical plan: governing equations, initial & boundary conditions, computational recommendations, diagnostics to record, and clear pass/fail criteria. I also include *practical notes* (resolution, expected compute) so you can start implementing.

Short primer (shared ingredients for all runs)

Continuum model. Treat pandemonium as a 4-spatial-dimensional continuum obeying a compressible Navier–Stokes–type PDE (with possible source/forcing terms to represent microphysics of pips). Use 4 spatial dimensions x1,x2,x3,x4x_1,x_2,x_3,x_4 and time tt.

A convenient nondimensional compressible form:

∂tρ+∇4⋅(ρu)=0,∂t(ρu)+∇4⋅(ρu⊗u)=−∇4p+∇4⋅τ+F,p=cs2ρ(isothermal EOS as a first step) \begin{aligned} \partial_t \rho + \nabla_4\cdot(\rho\mathbf{u}) &= 0,\\ \partial_t(\rho\mathbf{u}) + \nabla_4\cdot(\rho\mathbf{u}\otimes\mathbf{u}) &= -\nabla_4 p + \nabla_4\cdot\boldsymbol{\tau} + \mathbf{F},\\ p &= c_s^2 \rho \quad\text{(isothermal EOS as a first step)} \end{aligned}

where u(x,t)∈R4\mathbf{u}(\mathbf{x},t)\in\mathbb{R}^4, ∇4\nabla_4 is the 4D gradient/divergence, τ\boldsymbol{\tau} is viscous stress (Newtonian), csc_s is sound speed, and F\mathbf{F} is an optional forcing field you may use for setting initial vorticity or mimicking pip effects.

Dimensionless control parameters (analogues you'll sweep):

- Reynolds number Re=ULνRe = \dfrac{U L}{\nu} (where UU characteristic velocity; choose based on speed near light as nondimensional unit).
- Mach number Ma=UcsMa = \dfrac{U}{c_s} (start Ma≪1Ma\ll1 for near-incompressible runs; later raise).
- A **vorticity-stability parameter** R∗=ΓνR_*=\dfrac{\Gamma}{\nu} (circulation Γ\Gamma scale / viscosity) — Cobb references an R factor; treat R∗R_* as your vortex Reynolds analogue.
- Grid Peclet / resolution numbers for numerical stability.

Diagnostics (computed each run):

- 4-D vorticity tensor Ωij=∂iuj−∂jui\Omega_{ij} = \partial_i u_j - \partial_j u_i and its norm.
- Total kinetic energy Ek=∫12ρ∣u∣2dVE_k=\int \tfrac12\rho|\mathbf{u}|^2 dV.
- Circulation integrals on closed 2-surfaces (generalized to 4-D).
- Topological analysis: identify connected components of low-density/void cores and count how many 3-D lobes they produce when intersected by a 3-hyperplane x4=constx_4 = \mathrm{const} (simulate our observable 3-space).
- Emergent "field proxies": define scalar charge-proxy ρq(x)=∇4⋅J\rho_q(\mathbf{x}) = \nabla_4\cdot\mathbf{J} for some chosen mapping J\mathbf{J} (see practical note below), compute its Fourier transform to get a form factor.

- Magnetic moment proxy: compute integrated circulation around loops in 3-slices and map to dipole moment.

Implementation suggestions:

- Spatial discretization: finite-volume or spectral. Finite-volume easier to adapt from OpenFOAM ideas—must be extended to 4D. Spectral gives higher convergence but requires global transforms (more memory).
- Time stepping: explicit RK4 for low-Re, semi-implicit for stiff viscous terms if high Re.
- Start with periodic boundary conditions on a hypercube with damping “sponge” near edges to avoid wrap artifacts when you want isolated vortices.
- Save 3-slices frequently for visualization and analysis (e.g., many slices at different x_4x_4 to reconstruct 3D intersections).

Experiment 1 — Existence & stability of the single-lobe hyper-toroid (Electron test)

Goal: Produce a stable single-lobe hyper-toroidal vortex that intersects a 3-slice as a single spheroid and remains long-lived (many turnover times) under small perturbations.

Setup

- Domain: 4D periodic cube $[-L/2,L/2]^4$[-L/2,L/2]^4.
- Equations: incompressible approximation first ($\nabla_4 \cdot \mathbf{u}=0$\nabla_4\cdot\mathbf{u}=0) using 4-D Navier–Stokes to simplify numerics; then repeat compressible if successful.
- Initial condition: impose a seeded hyper-toroidal vorticity distribution centered at origin:
    - Construct 4-D toroidal velocity field by rotating a 3D torus profile in 4D: prescribe strong vorticity in a closed hyper-ring region (use analytical function with Gaussian cross-section).
    - Amplitude set so characteristic $U=1$U=1, choose ν\nu to set ReRe range $[50,5000]$[50,5000].
- Forcing: none after initialization (decaying vortex test).

Diagnostics & outputs

- Vorticity magnitude over time; energy decay; iso-surfaces of low density (voids).
- Count number of 3D lobes on 3-slice $x_4=0$x_4=0 at several times (visualize isosurfaces of vorticity/pressure).

- Perturbation test: introduce small random velocity noise (1% amplitude) and monitor whether vortex returns to original shape.

Pass criteria (supportive):

- A single central void/hollow appears in the 3-slice (one spheroid), persists for many turnover times (>50), and recovers shape after perturbation. The vortex circulation is conserved (within numerical dissipation bounds).

Fail criteria (falsifying):

- The seeded structure rapidly disintegrates into turbulence, or it fragments into multiple disconnected lobes that cannot be tuned away across reasonable viscosity range.

Computational notes

- Start with grid 64464^4 if possible; increase to 96496^4 or 1284128^4 for convergence checks. Memory: 64464^4 × (say) 8 bytes ≈ 134M scalar cells — realistic on HPC nodes; 128^4 is heavy (~2.1G cells). If resource-limited, use spectral low-mode truncation or axisymmetric reduction (see Practical shortcuts).

Experiment 2 — Existence & stability of the three-lobe hyper-toroid (Proton test)

Goal: Find a stable 4-D vortex topology that, when sliced, produces **three** distinct 3-lobes (candidate proton). Demonstrate topological robustness (cannot be separated into three independent vortices).

Setup

- Domain & numerics: same as Exp.1 (compressible later).
- Initial condition: seed a 4-D hyper-toroid with geometry designed (by construction) to produce three high-vorticity concentration zones — use Cobb's toy geometry for hyper-toroid #8 as initial guess (he describes a topologically unique toroidal rotation).
    - Build initial vorticity as sum of 3 Gaussian lobes connected through a thin hyper-bridge region to ensure continuity (i.e., one contiguous object).
- Sweep ReRe (e.g., 100, 500, 2000) and R_* stability parameter by varying viscosity.

Diagnostics

- Topology: compute connected components of the low-density core; confirm the object is one connected 4-D region with three local maxima in vorticity when viewed in 3-slices.
- Mechanical response: apply a stretching perturbation along one inter-lobe axis and record restorative force/elastic response (measure energy cost as lobes separate).

- Compute “binding energy”: energy difference between intact vortex and artificially separated lobes (if separation is feasible numerically).

Pass criteria:

- The structure remains one contiguous hyper-object; pulling lobes apart costs sharply increasing energy and the object relaxes back (elastic restoring). The three-lobe count remains invariant under small to moderate perturbations.

Fail criteria:

- The object either cannot be configured into a single connected structure (it quickly snaps into three independent vortices) or behaves like three weakly coupled vortices with negligible restoring energy (contradicting confinement-as-topology).

Notes:

- Demonstrating *topological confinement* is crucial: if the 3 lobes can be separated without energy skyrocketing, Cobb’s explanation is weakened.

Experiment 3 — Entwined hybrid and decay dynamics (Neutron test)

Goal: Build a stable entwined configuration of the proton-candidate and the electron-candidate hyper-toroids, then trigger/observe a decay process analogous to neutron beta decay (separation into proton-like and electron-like structures + ejected energy packet).

Setup

- Start from equilibrated solutions from Experiments 1 & 2 (best parameters).
- Place proton-candidate and electron-candidate as interlocked/linked hyper-toroids per Cobb’s suggested geometry; allow relaxation to find minimum-energy entwined state.
- Add a small perturbation or internal dissipative process to trigger a reconfiguration (mimic weak-interaction trigger).

Diagnostics

- Track topology changes: measure when the electron-like vortex detaches from proton and how the energy is partitioned (kinetic energy of outgoing pieces, residual bound energy).
- Compute emitted small vortex fragment properties (mass/energy proxy, velocity distribution) — compare to known beta decay Q-value (qualitatively).
- Track any neutrino-like low-energy small-vortex ejection.

Pass criteria:

- The entwined system is a metastable state; under perturbation it relaxes into a stable proton-like vortex + free electron-like vortex with an energy release consistent (order-of-magnitude) with neutron decay Q (~0.78 MeV equivalent in model units). Additionally, a small, low-mass high-velocity fragment (neutrino proxy) is ejected.

Fail criteria:

- Either the entwined configuration is not metastable (falls apart instantly) or decay cannot produce separated electron-like vortex without violating energy conservation in the model, or the ejected fragment energies are grossly incompatible across multiple parameter sweeps.

Notes:

- Energetic mapping: you will need to choose a conversion between model kinetic/flow energy units and physical MeV for comparison (this mapping is model-dependent but can be normalized by matching one observable, e.g., proton mass proxy).

Experiment 4 — Probe scattering & electromagnetic form-factor proxy

Goal: Simulate a high-momentum probe interaction (analogous to electron scattering) to compute the “form factor” of candidate vortices and compare the number of observed scattering centers / structure functions.

Setup

- Use equilibrated solutions from Exp.1 and Exp.2.
- Introduce a localized time-dependent forcing term Fprobe(x,t)\mathbf{F}_{\text{probe}}(\mathbf{x},t) that mimics a point-like high-q perturbation (short pulse with known wavelength components) passing near the vortex. Alternatively, solve linearized perturbation response (Green’s function) to an impulsive probe.
- Record scattered field as perturbations in the far field (in volume) and take 4-D → 3-slice projections.

Diagnostics

- Compute Fourier transform of scattered density/charge-proxy to get a form factor F(q)F(\mathbf{q}).
- On 3-slices count how many localized response peaks appear as a function of probe momentum (low q should see whole object; high q should resolve lobes if they exist).
- For three-lobe object: check whether scattering at intermediate q resolves three peaks consistent with DIS experiments.

Pass criteria:

- Single-lobe candidate shows a form factor consistent with a single coherent object at low/high q (no substructure). Three-lobe candidate shows substructure at higher q with threefold features (qualitatively similar to three valence quarks contributions).

Fail criteria:

- The three-lobe candidate yields no substructure across probes (implies lobes are not independent high-q centers), or the single-lobe candidate shows resolvable substructure inconsistent with pointlike electron.

Notes:

- This is conceptually the most direct analogue of DIS — matching the qualitative count of scattering centers is a strong test.

Experiment 5 — Parameter sweep: stability map & precision observables (g-factor, dipole moment proxies)

Goal: Map parameter space (Re, Ma, R_*) to find the stability windows for each vortex type and compute their emergent electromagnetic proxies (dipole moments, circulating currents) to compare qualitatively with measured properties (e.g., proton magnetic moment sign/scale, electron g-factor trends).

Setup

- Start from canonical equilibria found in Exps.1–2.
- Sweep viscosity ν\nu, characteristic circulation Γ\Gamma, and small compressibility (increase MaMa gradually).
- For each equilibrium compute:
    - Effective charge-proxy: integrated divergence of chosen current mapping (see Practical note).
    - Magnetic moment proxy: integral of circulation weighted by "charge-proxy" over 3-slice.
    - Response to small external uniform flow (simulate coupling to background pandemonium) to get polarizability.

Diagnostics

- Produce stability diagrams (phase plots) showing regions where each vortex type is stable, metastable, or unstable.

- Plot magnetic moment proxy vs parameter values; check if sign and relative magnitudes are robust (e.g., proton magnetic moment > electron? scales are model dependent but trends matter).
- If possible compute analogue of g-factor by measuring precession of vortex orientation in an external imposed 3D magnetic-proxy flow.

Pass criteria:

- Stable, bounded regions exist for each vortex type; magnetic/dipole proxies show consistent, robust nonzero values that could plausibly map to observed signs/magnitudes after calibration.

Fail criteria:

- No stable region for any vortex type (implies shapes are numerical artifacts). Or magnetic proxies change sign wildly with small parameter tweaks (no robustness).

Practical mapping: how to get from CFD outputs to "observables"

Cobb gives heuristic mappings (you'll need to formalize one):

- **Charge proxy**: a scalar obtained from divergence of a selected component of flow (e.g., along primary spin axis) or from imbalance of secondary circulations. Operationally: ρq(x)=α ∇4 · u⊥\rho_q(\mathbf{x}) = \alpha\,\nabla_4\cdot\mathbf{u}_{\perp} where u⊥\mathbf{u}_{\perp} is flow perpendicular to primary spin and α\alpha a normalization constant you calibrate.
- **Magnetic moment proxy**: integrate local circulation weighted by ρq\rho_q over 3-slice: μ=∫(r×Jq) dV\boldsymbol{\mu} = \int (\mathbf{r}\times\mathbf{J}_q)\,dV where Jq=ρqu\mathbf{J}_q=\rho_q\mathbf{u} projected into 3-space slice.
- **Form factor**: F(q)=∫ρq(r)e−iq · rd3rF(\mathbf{q})=\int \rho_q(\mathbf{r}) e^{-i\mathbf{q}\cdot\mathbf{r}}d^3r on 3-slices — use to compare scattering signatures.

Be explicit and consistent in the manuscript about your chosen mapping; the mapping's physics justification will be key to acceptance.

Practical shortcuts / computational economy

- **Symmetry reductions.** Many 4-D vortex topologies are symmetric under some rotations; use coordinate transforms to reduce problem dimensionality (e.g., assume rotation invariance in one angle and solve an effective 3D+parameter system). This lets you explore parameter space cheaply before full 4D runs.

- **Low-mode Galerkin models.** Rather than full grid, expand solution in a basis (few dominant modes) and study mode dynamics — fast, gives intuition, cheap. Use to find attractors and then confirm with full CFD.
- **Adaptive resolution.** Local mesh refinement around vortex cores reduces memory cost massively.

Resources & approximate compute needs

- **Pilot runs:** 32432^4–64464^4 grids on an HPC node with 256–512 GB RAM (or a small cluster) for exploratory tests.
- **Converged runs:** 96496^4–1284128^4 requiring multi-node MPI, terabytes of RAM/disk for snapshots.
- If resources are limited: use reduced/symmetric models and low-mode Galerkin first.

Final checklist: what outcome convinces vs. falsifies

Convincing evidence (support Cobb’s shape assignments):

- Stable single-lobe and three-lobe 4-D vortices identified as attractors across parameter ranges.
- Topological confinement: three-lobe object is one contiguous object with strong restorative energy against separation.
- Scattering proxy reproduces the qualitative count (single vs three centers) and form-factor behavior.
- Entwined hybrid displays metastability and plausible decay energetics.

Decisive falsification:

- No stable single-lobe attractor exists (electron candidate cannot be realized).
- Three-lobe structures cannot be made contiguous or show no confinement behaviour (they behave as weakly coupled vortices).
- Scattering shows irreconcilable signatures (e.g., single-lobe object always shows internal substructure) inconsistent with experiments.

If you want, I can now:

- Draft **template code** for the initial condition builders (analytic velocity/vorticity profiles for a 4-D torus and a 3-lobe connected profile) in pseudocode or Python (to feed a solver), **or**
- Produce a **minimal mathematical note** that writes out the 4-D Navier–Stokes variant, vorticity definitions, and the closed integrals used as diagnostics (so an HPC coder can implement quickly).

Which of those two do you want next?

========

My prompt to GPT-5:

I would like both.

Its response:

Nice — below are **both** deliverables you asked for:

1. **Template Python code** (runnable/pseudocode) that builds analytic initial velocity/vorticity fields for:
   - a single-lobe 4-D hyper-toroid (electron candidate), and
   - a 3-lobe connected hyper-toroid (proton candidate),

with a *spectral Helmholtz projection* to make the velocity field divergence-free (periodic BCs assumed). This is intended as an **initial condition generator** you can feed into your 4-D CFD solver.

2. **Minimal mathematical note** that defines the governing equations (4D compressible/incompressible Navier–Stokes variants), the 4D vorticity tensor (2-form), and the diagnostics you should compute (vorticity norm, generalized circulation integrals, form factors, charge/magnetic-proxies), plus implementation hints linking the math to the code.

---

1) Template Python initial-condition builder (pseudocode / runnable-style)

Notes:

- This code uses `numpy` arrays and FFT-based Helmholtz projection (so periodic BCs).

- It **does not** run a CFD solver; it produces `u(x)` 4D arrays on a grid.

- For real HPC run, translate to your solver's data structures (MPI domain decomposition, float32/64), or use spectral CFD libs and replace numpy FFT with `pyfftw` / MPI FFT.

- Memory: keep `N` small for testing (`N=32` or `48`) until you have HPC resources.

- I give comments where you'll want to adapt to your solver.

```
# initial_conditions_4d.py  (template)
import numpy as np
from numpy.fft import fftn, ifftn, fftfreq

# ---------- Utilities ----------
def grid4d(N, L=1.0):
    """Return 4D coordinate arrays X1..X4 each shape (N,N,N,N)."""
    xs = np.linspace(-L/2, L/2, N, endpoint=False)
    X = np.meshgrid(xs, xs, xs, xs, indexing='ij')
    return X  # tuple of 4 arrays

def gaussian(r2, sigma):
    return np.exp(-0.5 * r2 / (sigma**2))

# ---------- Divergence-free projection (spectral) ----------
def helmholtz_project_4d(u):
    """
    u: tuple/list of 4 arrays (u1,u2,u3,u4) each shape (N,N,N,N)
    returns divergence-free projection u_perp in spectral domain (periodic)
    """
    # Stack components
    u_comp = np.stack(u, axis=0)  # shape (4, N, N, N, N)
    shape = u_comp.shape[1:]
    # FFT each component
    u_hat = fftn(u_comp, axes=(1,2,3,4))
    # build k-vector arrays
    N = shape[0]
    L = float(N)  # assume unit grid spacing; adjust if dx != 1/N
    k1 = fftfreq(N, d=1.0/N) * 2*np.pi
    k = np.meshgrid(k1, k1, k1, k1, indexing='ij')
    k_vec = np.stack(k, axis=0)  # shape (4,N,N,N,N)
    ksq = np.sum(np.abs(k_vec)**2, axis=0)
    # avoid divide by zero at k=0
    ksq[ksq==0] = 1.0
    # projection: u_perp_hat = u_hat - k (k·u_hat)/|k|^2
    k_dot_u_hat = np.sum(k_vec * u_hat, axis=0)  # shape (N,N,N,N)
    correction = (k_vec * k_dot_u_hat / ksq)  # broadcasts to (4,N,N,N,N)
    u_perp_hat = u_hat - correction
    # inverse FFT
    u_perp = ifftn(u_perp_hat, axes=(1,2,3,4)).real
    return [u_perp[i] for i in range(4)]

# ---------- Analytic building blocks ----------
```

```
def toroidal_vortex_4d(X, center, R_major, r_minor, axis_plane=(0,1), sigma=0.1,
amplitude=1.0):
    """
    Construct a smooth toroidal vorticity/velocity 'seed' in 4D.
    axis_plane: which pair of coordinate indices define the torus rotation plane
(0..3).
    Returns a 4-vector velocity field (u1,u2,u3,u4) on grid X.
    Simple ansatz: define distance to torus centerline in the perpendicular
subspace, then create swirl.
    """
    x1,x2,x3,x4 = X
    cx,cy,cz,cw = center
    # rotate coordinates so axis_plane is (a,b) and perpendicular coords are (c,d)
    coords = [x1-cx, x2-cy, x3-cz, x4-cw]
    a,b = axis_plane
    perp = [i for i in range(4) if i not in (a,b)]
    # radius in axis plane relative to center
    Rplane = np.sqrt(coords[a]**2 + coords[b]**2)
    # position relative to centerline of torus (distance to ring)
    dist_to_ring = np.sqrt((Rplane - R_major)**2 + coords[perp[0]]**2 +
coords[perp[1]]**2)
    # generate scalar swirl magnitude
    swirl = amplitude * gaussian(dist_to_ring**2, sigma)
    # build a tangential unit vector around the ring (in a-b plane)
    # t = (-y, x)/Rplane  (handle Rplane~0)
    eps = 1e-12
    ux = np.zeros_like(x1)
    uy = np.zeros_like(x1)
    if True:
        t_a = -coords[b] / (Rplane + eps)
        t_b =  coords[a] / (Rplane + eps)
        # assign tangential component to the axis plane coords scaled by swirl
        comp = [np.zeros_like(x1) for _ in range(4)]
        comp[a] = swirl * t_a
        comp[b] = swirl * t_b
        # add small poloidal circulation around local ring cross-section (perp
coords)
        # poloidal direction vector (perp[0], perp[1]) rotates around local normal
        pol_t = -coords[perp[1]] / (np.sqrt(coords[perp[0]]**2 +
coords[perp[1]]**2)+eps)
        pol_u = coords[perp[0]] / (np.sqrt(coords[perp[0]]**2 +
coords[perp[1]]**2)+eps)
        comp[perp[0]] = 0.3 * swirl * pol_t
        comp[perp[1]] = 0.3 * swirl * pol_u
        return comp  # list of 4 arrays
    # fallback (never reached)
    return [ux,uy,ux,uy]

def three_lobe_connected_4d(X, centers, R_major=0.3, r_minor=0.08):
    """
    Build three torus lobes arranged around a circle in some subspace and connected
by bridges.
    centers: list of 3 centers in 4D (tuples)
    We'll sum three toroidal seeds, then add small Gaussian bridges between lobes.
    """
    u_acc = [np.zeros_like(X[0]) for _ in range(4)]
    for c in centers:
        u_l = toroidal_vortex_4d(X, center=c, R_major=R_major, r_minor=r_minor,
axis_plane=(0,1), sigma=r_minor*0.8, amplitude=1.0)
```

```
        for i in range(4):
            u_acc[i] += u_l[i]
    # add bridges: simple gaussian vector fields directed along short lines
connecting centers
    for i in range(3):
        j = (i+1) % 3
        c1 = centers[i]
        c2 = centers[j]
        # midpoint and direction
        mid = tuple(0.5*(np.array(c1)+np.array(c2)))
        dir_vec = np.array(c2) - np.array(c1)
        # create small Gaussian along mid with vector field pointing along dir_vec
(normalized)
        X1,X2,X3,X4 = X
        r2 = (X1-mid[0])**2 + (X2-mid[1])**2 + (X3-mid[2])**2 + (X4-mid[3])**2
        bridge_strength = 0.4 * np.exp(-0.5 * r2 / ( (r_minor*1.2)**2 ))
        for k in range(4):
            u_acc[k] += bridge_strength * (dir_vec[k] / (np.linalg.norm(dir_vec)
+1e-12))
    return u_acc

# ---------- Example wrapper ----------
def build_electron_like(N=48, L=1.0):
    X = grid4d(N, L=L)
    # single-lobe centered at origin, major radius small so intersection is single-
lobe
    center = (0.0, 0.0, 0.0, 0.0)
    u_raw = toroidal_vortex_4d(X, center=center, R_major=0.05, r_minor=0.02,
axis_plane=(0,1), sigma=0.02, amplitude=1.0)
    # project to divergence-free
    u_divfree = helmholtz_project_4d(u_raw)
    return u_divfree, X

def build_proton_like(N=48, L=1.0):
    X = grid4d(N, L=L)
    # place three centers on a ring in (x1,x2) with small offsets in x3,x4
    angle = 0.0
    R_center = 0.12
    centers = []
    for k in range(3):
        ang = 2*np.pi*k/3.0 + 0.2
        centers.append( (R_center*np.cos(ang), R_center*np.sin(ang),
0.02*np.cos(2*ang), 0.02*np.sin(2*ang)) )
    u_raw = three_lobe_connected_4d(X, centers, R_major=0.05, r_minor=0.03)
    u_divfree = helmholtz_project_4d(u_raw)
    return u_divfree, X

# ---------- Save or convert to solver format ----------
if __name__ == "__main__":
    Nx = 32
    u_e, X = build_electron_like(N=Nx)
    u_p, Xp = build_proton_like(N=Nx)
    # Example: convert to single array shape (4,N,N,N,N) and save to disk (np.save)
or HDF5 for solver ingestion
    u_e_array = np.stack(u_e, axis=0)
    u_p_array = np.stack(u_p, axis=0)
    np.save("electron_init.npy", u_e_array)
    np.save("proton_init.npy", u_p_array)
    print("Saved initial condition files.")
```

Implementation tips for the code above

- The `toroidal_vortex_4d` function is intentionally *simple* — it constructs a smooth toroidal swirl by localizing tangential flow near a 4D ring. You will tune `R_major`, `r_minor`, `sigma`, and `amplitude` to explore laminar vs turbulent regimes.
- The Helmholtz projection assumes periodic BCs. If your solver uses no-slip or different BCs, replace projection with a Poisson solve consistent with boundary conditions (i.e., subtract gradient of scalar potential).
- To reduce memory, create arrays of type `np.float32`. For production HPC, implement the same analytic formula on distributed arrays and project with parallel FFT (P3DFFT / FFTW + MPI).

---

2) Minimal mathematical note — governing equations, vorticity, diagnostics

A. Governing PDEs (4 spatial dimensions)

Compressible 4-D Navier–Stokes (conservative form)

Let spatial coordinates be xix^i with i=1..4i=1..4, time tt. Fields: mass density ρ(x,t)\rho(x,t), velocity u=(u1,…,u4)\mathbf{u} = (u^1,\dots,u^4), pressure pp, viscous stress τij\tau^{ij}.

∂tρ+∂i(ρui)=0,∂t(ρuj)+∂i(ρuiuj)=−∂jp+∂iτij+Fj,τij=μ(∂iuj+∂jui)+λ δij (∂kuk),\begin{aligned} \partial_t \rho + \partial_i(\rho u^i) &= 0, \\ \partial_t(\rho u^j) + \partial_i(\rho u^i u^j) &= -\partial^j p + \partial_i \tau^{ij} + F^j, \\ \tau^{ij} &= \mu\left(\partial^i u^j + \partial^j u^i\right) + \lambda\, \delta^{ij}\,(\partial_k u^k), \end{aligned}

where repeated indices sum i=1..4i=1..4. Choose Newtonian constants μ\mu (shear viscosity) and λ\lambda (bulk viscosity). Equation of state: start with isothermal p=cs2ρp = c_s^2 \rho or ideal gas p=ρRTp = \rho R T.

Incompressible limit (first tests):

∂iui=0,∂tuj+ui∂iuj=−∂jp+νΔ4uj+fj,\partial_i u^i = 0, \qquad \partial_t u^j + u^i\partial_i u^j = -\partial^j p + \nu \Delta_4 u^j + f^j,

with ν=μ/ρ\nu=\mu/\rho, and Δ4\Delta_4 Laplacian in 4D.

B. Vorticity in 4D — the antisymmetric 2-form / tensor

In nn-dimensional space the vorticity is a 2-form ω=12ωij dxi∧dxj\omega = \tfrac12 \omega_{ij} \,dx^i\wedge dx^j. Components:

ωij=∂iuj−∂jui,i,j∈{1,…,4}.\omega_{ij} = \partial_i u_j - \partial_j u_i,\qquad i,j\in\{1,\dots,4\}.

There are (42)=6 \binom{4}{2} = 6 independent components: ω12,ω13,ω14,ω23,ω24,ω34\omega_{12},\omega_{13},\omega_{14},\omega_{23},\omega_{24},\omega_{34}.

Vorticity evolution (from curl of momentum eqn):

∂tωij+uk∂kωij=ωik∂juk−ωjk∂iuk+νΔ4ωij+(∂ifj−∂jfi).\partial_t \omega_{ij} + u^k\partial_k \omega_{ij} = \omega_{ik}\partial_j u^k - \omega_{jk}\partial_i u^k + \nu \Delta_4 \omega_{ij} + (\partial_i f_j - \partial_j f_i).

This is the natural extension of vorticity transport to 4D (advection + vortex stretching terms generalized).

C. Generalized circulation and topological diagnostics

- **Classical (3D)** circulation `Γ` around closed loop `C` is Γ=∮Cu · dr\Gamma = \oint_C \mathbf{u} \cdot d\mathbf{r}. Stokes relates it to surface integral of vorticity 2-form over surface `S` with boundary `∂S=C`.
- **In 4D:** vorticity is already a 2-form. Integration of ω\omega over a chosen oriented 2-surface Σ2\Sigma^2 (embedded in 4D) gives a scalar circulation content:

Γ(Σ)=∬Σ12ωij dSij.\Gamma(\Sigma) = \iint_{\Sigma} \frac{1}{2}\omega_{ij}\,dS^{ij}.

Here dSijdS^{ij} is an oriented area element in the i − ji\!-\!j coordinate plane of the surface patch. By generalized Stokes, the integral of ω\omega over a 2-surface equals the integral of uu over its 1-dimensional boundary (if boundary exists).

Discrete diagnostic: pick coordinate 2-planes (e.g., x1 − x2x^1\!-\!x^2 plane or curved 2-surface built in 4D) and compute:

Γ12(S)≈∑cells∈Sω12(x) ΔS.\Gamma_{12}(S) \approx \sum_{cells\in S} \omega_{12}(x)\, \Delta S.

Compare `Γ` across surfaces that thread the vortex core to detect quantized/robust circulation.

D. Observables / proxies mapped from CFD fields

Cobb's model needs mappings from pandemonium flow to classical observables (charge, magnetic moment, form factor). Below are pragmatic proxies:

1. **Vorticity norm (local intensity)**

∣ω∣(x)=12∑i<jωij2.|\omega|(x) = \sqrt{\tfrac12 \sum_{i<j} \omega_{ij}^2 }.

Useful to locate vortex cores (maxima) and void centers (low density region encircled by high vorticity).

2. **Charge proxy ρq(x)\rho_q(x)** — choose a physically-motivated mapping (must be declared):

One possible choice (works as a starting point):

$$\rho_q(x) = \alpha\, \partial_i (P^{i}) \quad\text{with}\quad P^{i} = \epsilon^{i j k \ell} \, u_j \, \omega_{k\ell}.$$

- Here ϵ^{ijkl} is the 4D Levi-Civita tensor.
- `P^i` is a pseudovector built from flow and vorticity; divergence of it gives a scalar charge proxy.
- Calibrate α such that integrated $\int \rho_q \,d^3x$ over a 3-slice equals ±1 for basic vortex.

Simpler alternative: pick component of `u` along a primary spin axis and use divergence of that component.

3. **Magnetic moment proxy μ** (3D slice view): pick a 3-slice (our observed 3-space). With `ρ_q` and 3-space velocity `v( r )` (projection of `u` into slice), define:

$$\boldsymbol{\mu} = \int_{V_{3}} \mathbf{r}\times (\rho_q \mathbf{v})\, d^3r.$$

Calculate sign and magnitude vs parameters. Use this as proxy for magnetic dipole.

4. **Form factor / scattering proxy**
   On a chosen 3-slice, compute:

$$F(\mathbf{q}) = \int_{V_3} \rho_q(\mathbf{r})\, e^{-i\mathbf{q}\cdot \mathbf{r}}\, d^3r.$$

Compare low-`q` (coherent) and high-`q` (substructure) behavior. Use radial average for simpler diagnostics.

5. **Void/ core detection & lobe counting**
   - Compute pressure or a "density proxy" (e.g., `p = c_s^2 rho`). Or use `|ω|` contour to detect surrounding ring and low-pressure interior.
   - For each 3-slice $x_4 = const$ identify connected components of low-density cores (e.g., `density < threshold`). Count components as lobes. Sweep `x_4` to see how many intersections appear — that gives number of 3-lobes.

E. Discrete numerical formulas (what to implement)

- **Vorticity components:** discrete derivatives via spectral or central differences:

$$\omega_{ij}(\mathbf{x}) \approx \partial_i u_j - \partial_j u_i$$

Use spectral differentiation (multiply `u_hat` by `i k_i`) for accuracy and to avoid numerical noise.

- **Circulation integral over 2D grid patch S in plane (i,j):**

$$\Gamma_{ij} \approx \sum_{cells\in S} \omega_{ij}(\mathbf{x}) \,\Delta S$$

(ΔS is the 2D area of each surface patch).

- **Helmholtz projection (spectral):**

u^⊥(k)=u^(k)−k(k · u^(k))|k|2,\hat{\mathbf{u}}_\perp(\mathbf{k}) = \hat{\mathbf{u}}(\mathbf{k}) - \frac{\mathbf{k}(\mathbf{k}\cdot\hat{\mathbf{u}}(\mathbf{k}))}{|\mathbf{k}|^2},

and inverse FFT to get `u_perp(x)`.

F. Practical parameter mapping & calibration

- **Non-dimensionalization:** choose characteristic length L0L_0 and velocity U0U_0. Let `R_major` ~ few × grid spacing in nondim units. Map to physical units after you match one empirical datum (e.g., set proton mass proxy via total kinetic energy constant).
- **Dimensionless numbers:** Reynolds Re=U0L0/νRe = U_0 L_0 / \nu, R_* (vortex Reynolds) = Γ/ν\Gamma/\nu. Sweep Re from laminar to transitional: e.g., Re ∈ [10^1,10^4] in nondim units.
- **Mapping energy → MeV:** pick one anchor: e.g., choose parameter set for proton-like vortex such that its total flow energy equals mpc2m_p c^2. That gives scale factors for comparing energetics qualitatively.

Practical final notes + checklist to implement

- **Start simple:** incompressible 4D Navier–Stokes, small grids (32^3×32), spectral projection; find steady vortex attractors. If found, port to compressible solver and increase resolution.
- **Diagnostics to always output:** vorticity component maps, vorticity norm, 3-slice isosurfaces, circulation integrals on several 2-surfaces, `rho_q` slices, form factor `F(q)` radial shells, time series of total kinetic energy, vortex core positions and topological counts.
- **Stability tests:** add white-noise perturbation (1–5% amplitude) and check recovery; apply stretching perturbation between lobe centers and compute energy vs separation.
- **Reproducibility:** script the full experiment with random seeds, parameter sweeps, and data-saving (HDF5 with metadata). Keep scripts to compute diagnostics from saved snapshots so you can rerun analyses cheaply.

If you want next:

- I can **convert the initial-condition templates above into an MPI-enabled example** (pseudocode) that uses `mpi4py` + `pyfftw` parallel FFTs — useful if you'll run medium-sized `64^4` tests on a cluster.

- Or I can **draft the exact set of diagnostics routines** (Python functions) for computing `omega_ij`, circulation integrals on arbitrary 2-surfaces, 3-slice lobe counting and form-factor computation, ready to plug into your postprocessing pipeline.

Which of those should I prepare next?

========

My prompt to GPT-5:

I would like both.

Its response:

Nice — below are **both** deliverables you asked for, ready to drop into an HPC workflow.

1. **MPI-enabled initial-condition generator (pseudocode / runnable-style)** using `mpi4py` + `mpi4py_fft`/`pyfftw` patterns (two fallbacks included: a recommended-parallel path using `mpi4py_fft` if available, and a simpler single-node `pyfftw` fallback). This produces divergence-free 4-D velocity fields and writes HDF5 output in a chunked, MPI-friendly way.
2. **Diagnostics/postprocessing library** — a set of Python functions you can import and run on saved snapshot arrays (per-process chunks). Each routine handles local computation and includes MPI reduction steps so the full-domain diagnostics are returned to rank 0 for plotting/analysis:
    - `compute_vorticity_4d`
    - `compute_vorticity_norm`
    - `compute_circulation_on_patch` (arbitrary 2-surface patch in a grid-aligned plane)
    - `count_3slice_lobes` (connected-component labelling of low-density cores in 3-slices)
    - `compute_form_factor` (FFT-based form factor on chosen 3-slice)

Important notes before you run:

- True production 4-D spectral transforms and 4-D distributed arrays are nontrivial. The code below uses common HPC building blocks and is intentionally explicit about when you'll need a specialized library (e.g., `mpi4py_fft` or `pfft`). If those aren't available you can run small proof-of-concept tests single-node with `pyfftw` or NumPy FFT.

- The code assumes a uniform Cartesian grid with periodic BCs. If your CFD uses other BCs, adapt the projection step accordingly (Poisson solve for pressure).

- I provide explicit MPI reductions (sums, maxima) so the routines can be used in an MPI job without rewriting.

A — MPI-enabled initial-condition generator (template)

```
# init4d_mpi.py
# MPI-aware initial-condition builder for 4D periodic grids.
# Produces divergence-free 4D velocity arrays and saves in HDF5 (chunked).
#
# Requires: mpi4py, h5py (MPI build), numpy
# Optional (recommended): mpi4py_fft (for distributed FFT), pyfftw (for local FFT
acceler.)
#
# Run via: mpirun -n <P> python init4d_mpi.py --N 64 --out electron.h5

import argparse
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Optional imports
try:
    from mpi4py_fft import PFFT, newDistArray  # pip install mpi4py-fft
    HAVE_MPIFFT = True
except Exception:
    HAVE_MPIFFT = False

try:
    import pyfftw
    HAVE_PYFFTW = True
except Exception:
    HAVE_PYFFTW = False

# HDF5 writer will be done via h5py with MPI driver if available
try:
    import h5py
    H5_MPI = hasattr(h5py, "File") and hasattr(h5py, "Dataset")
except Exception:
    h5py = None
    H5_MPI = False

# ---------- Utilities: analytic "seeds" ----------
def grid_coords_local(N, L, slab_axis=0):
    """
    Domain decomposition: slab decomposition on axis 'slab_axis' (0..3)
    Returns local coordinate arrays X1..X4 (numpy arrays) for this rank.
    This simple slab splits the first axis x0 among ranks.
    """
    # global grid
    xs = np.linspace(-L/2, L/2, N, endpoint=False)
    counts = [N//size + (1 if i < (N % size) else 0) for i in range(size)]
    starts = [sum(counts[:r]) for r in range(size)]
    local_n0 = counts[rank]
    start = starts[rank]
    local_x0 = xs[start:start+local_n0]
    X = np.meshgrid(local_x0, xs, xs, xs, indexing='ij')  # local slab
    return X, start, local_n0

def gaussian(r2, sigma):
    return np.exp(-0.5 * r2 / (sigma**2))

def toroidal_vortex_4d_local(X, center, R_major, r_minor, axis_plane=(0,1),
sigma=0.1, amplitude=1.0):
    x1,x2,x3,x4 = X
    cx,cy,cz,cw = center
    coords = [x1-cx, x2-cy, x3-cz, x4-cw]
    a,b = axis_plane
    perp = [i for i in range(4) if i not in (a,b)]
    Rplane = np.sqrt(coords[a]**2 + coords[b]**2)
    dist_to_ring = np.sqrt((Rplane - R_major)**2 + coords[perp[0]]**2 +
coords[perp[1]]**2)
    swirl = amplitude * gaussian(dist_to_ring**2, sigma)
    # tangential
    eps = 1e-12
    t_a = -coords[b] / (Rplane + eps)
    t_b =  coords[a] / (Rplane + eps)
    comp = [np.zeros_like(x1) for _ in range(4)]
    comp[a] = swirl * t_a
    comp[b] = swirl * t_b
    # small poloidal
    norm_perp = np.sqrt(coords[perp[0]]**2 + coords[perp[1]]**2) + eps
    comp[perp[0]] = 0.25 * swirl * (-coords[perp[1]]/norm_perp)
    comp[perp[1]] = 0.25 * swirl * (coords[perp[0]]/norm_perp)
    return comp

# ---------- Divergence-free projection (distributed) ----------
def helmholtz_project_distributed(u_local, N, L, axis0_local_n, axis0_start):
    """
    u_local: list of 4 local arrays shaped (n0_local, N, N, N)
    If mpi4py_fft available, use its PFFT to compute global projection efficiently.
    Fallback: gather to rank0, do global FFT, project, broadcast (only for small
N).
    """
    if HAVE_MPIFFT:
        # Use slab distribution consistent with PFFT's dist -- here we do a 4D
transform
        # NOTE: mpi4py_fft requires careful dist setup; we present a recommended
pattern
        axes = (0,1,2,3)
        # create PFFT with same global shape and same slab decomposition along
axis=0
        pfft = PFFT(comm, shape=(N,N,N,N), axes=axes, dtype='complex128')
        # allocate distributed arrays
        u_hat = [pfft.create_input_array() for _ in range(4)]
        # copy local real-space u into the PFFT input arrays (they handle local 
layout)
        for i in range(4):
            u_hat[i][...] = u_local[i]
        # forward FFT
        for i in range(4):
            pfft.forward(u_hat[i], u_hat[i])  # in-place pattern (depends on API)
        # compute k vectors on local spectral distribution
        kvec = pfft.get_kvec()  # pseudocode: actual API differs; consult docs
        ksq = np.sum(kvec**2, axis=0)
        ksq[ksq==0] = 1.0
        # compute k·u_hat locally and projection
        # u_perp_hat = u_hat - k*(k·u_hat)/|k|^2
        # note: shape and broadcasting must match; this is pseudocode
        k_dot_u = None  # compute via sum(kvec * u_hat components)
        # inverse transform to get u_perp in real space
        for i in range(4):
            pfft.backward(u_hat[i], u_hat[i])
        # return real parts local arrays
        u_perp_local = [arr.real.copy() for arr in u_hat]
        return u_perp_local
    else:
        # fallback: gather to rank0 (only for small N)
        if rank == 0:
            full = np.zeros((4, N, N, N, N), dtype='float64')
            # place rank0's slab
            local_n0 = axis0_local_n
            full[:, axis0_start:axis0_start+local_n0, :, :, :] = np.stack(u_local, 
axis=0)
            # receive others
            for r in range(1, size):
                cnt = comm.recv(source=r, tag=10+r)
                slab = np.empty((4, cnt, N, N, N), dtype='float64')
                comm.Recv(slab, source=r, tag=100+r)
                start_r = sum([N//size + (1 if i < (N % size) else 0) for i in 
range(r)])
                full[:, start_r:start_r+cnt, :, :, :] = slab
            # global FFT & projection (numpy fftn)
            full_hat = np.fft.fftn(full, axes=(1,2,3,4))
            # build k vectors (global)
            k1 = np.fft.fftfreq(N, d=1.0/N) * 2*np.pi
            k = np.meshgrid(k1,k1,k1,k1, indexing='ij')
            kvec = np.stack(k, axis=0)
            ksq = np.sum(kvec**2, axis=0)
            ksq[ksq==0] = 1.0
            # project each spectral component
            u_perp_hat = full_hat - (kvec * np.sum(kvec*full_hat, axis=0)[None,...] 
/ ksq[None,...])
            # inverse FFT
            u_perp = np.fft.ifftn(u_perp_hat, axes=(1,2,3,4)).real
            # send slabs back to ranks
            for r in range(1,size):
                cnt = [N//size + (1 if i < (N % size) else 0) for i in range(size)]
[r]
                start_r = sum([N//size + (1 if i < (N % size) else 0) for i in 
range(r)])
                slab = u_perp[:, start_r:start_r+cnt, :, :, :].copy()
                comm.send(cnt, dest=r, tag=10+r)
                comm.Send(slab, dest=r, tag=100+r)
            # pack local slab for rank0
            local_out = [u_perp[i, axis0_start:axis0_start+axis0_local_n, :, :, :]
for i in range(4)]
            return local_out
        else:
            # non-root: send local data to root
            local_stack = np.stack(u_local, axis=0).astype('float64')
            comm.send(axis0_local_n, dest=0, tag=10+rank)
            comm.Send(local_stack, dest=0, tag=100+rank)
            # wait for root to return projected slab
            cnt = comm.recv(source=0, tag=10+rank)
            slab = np.empty((4, cnt, N, N, N), dtype='float64')
            comm.Recv(slab, source=0, tag=100+rank)
            return [slab[i] for i in range(4)]

# ---------- HDF5 writer ----------
def write_hdf5_parallel(fname, dataset_name, local_array, global_shape,
axis0_start):
    """
    local_array shape (4, local_n0, N, N, N)
    Writes as dataset [4, N, N, N, N] with axis 1 distributed (slab).
    Uses h5py MPI driver if available; otherwise rank0 gathers and writes (small
runs).
    """
    if H5_MPI and h5py.get_config().mpi:
        with h5py.File(fname, 'w', driver='mpio', comm=comm) as f:
            dset = f.create_dataset(dataset_name, shape=global_shape, dtype='f8')
            # hyperslab write
            start = (0, axis0_start, 0, 0, 0)
            count = (4, local_array.shape[1], global_shape[2], global_shape[3],
global_shape[4])
            dset[start[0]:start[0]+count[0], start[1]:start[1]+count[1], :, :, :] =
local_array
    else:
        # fallback: gather to rank0 (only for small N)
        if rank == 0:
            full = np.empty(global_shape, dtype='float64')
            full[:, axis0_start:axis0_start+local_array.shape[1], :, :, :] =
local_array
            # receive other slabs
            for r in range(1, size):
                cnt = comm.recv(source=r, tag=200+r)
                slab = np.empty((4, cnt, global_shape[2], global_shape[3],
global_shape[4]), dtype='float64')
                comm.Recv(slab, source=r, tag=300+r)
                start_r = sum([global_shape[1]//size + (1 if i < (global_shape[1] %
size) else 0) for i in range(r)])
                full[:, start_r:start_r+cnt, :, :, :] = slab
            # write with local h5py
            import h5py
            with h5py.File(fname, 'w') as f:
                f.create_dataset(dataset_name, data=full, compression='gzip')
            print("Wrote", fname)
        else:
            # send slab to root
            comm.send(local_array.shape[1], dest=0, tag=200+rank)
            comm.Send(local_array, dest=0, tag=300+rank)
```

```
# ---------- Main driver ----------
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--N", type=int, default=48, help="global grid size per
axis (N^4)")
    parser.add_argument("--L", type=float, default=1.0)
    parser.add_argument("--out", type=str, default="init4d.h5")
    parser.add_argument("--type", type=str, default="electron",
choices=("electron","proton"))
    args = parser.parse_args()

    N = args.N
    L = args.L
    X, start, local_n0 = grid_coords_local(N, L, slab_axis=0)
    # choose analytic seed
    if args.type == "electron":
        u_raw = toroidal_vortex_4d_local(X, center=(0,0,0,0), R_major=0.05,
r_minor=0.02, axis_plane=(0,1), sigma=0.02, amplitude=1.0)
    else:
        # proton-like: three lobes placed around ring
        angles = [0, 2*np.pi/3, 4*np.pi/3]
        centers = [(0.12*np.cos(a), 0.12*np.sin(a), 0.02*np.cos(2*a),
0.02*np.sin(2*a)) for a in angles]
        # sum their contributions locally
        uacc = [np.zeros_like(X[0]) for _ in range(4)]
        for c in centers:
            cseed = toroidal_vortex_4d_local(X, center=c, R_major=0.05,
r_minor=0.03, axis_plane=(0,1), sigma=0.03, amplitude=1.0)
            for i in range(4):
                uacc[i] += cseed[i]
        u_raw = uacc

    # Project to be divergence-free
    u_proj = helmholtz_project_distributed(u_raw, N=N, L=L, axis0_local_n=local_n0,
axis0_start=start)

    # pack into shape (4, local_n0, N, N, N)
    local_stack = np.stack(u_proj, axis=0)

    # write HDF5
    global_shape = (4, N, N, N, N)
    write_hdf5_parallel(args.out, "u", local_stack, global_shape,
axis0_start=start)

    if rank == 0:
        print("Initial condition generation complete.")

if __name__ == "__main__":
    main()
```

How to use / adapt

- If you have `mpi4py_fft`, adapt the `helmholtz_project_distributed` block to that API — it will be orders of magnitude faster and memory-efficient for large `N` across many ranks. I left a pseudocode placeholder; consult `mpi4py_fft` docs for exact function names.

- For small-scale validation (`N<=48`) on a single node, run with `mpirun -n 1` and the fallback gather-based projection will work.
- Output HDF5 file format: dataset `"u"` with shape `(4, N, N, N, N)` where axis 1 is the x0 axis distributed in slabs.

B — Diagnostics / postprocessing routines (Python module)

Save as `diagnostics4d.py`. These functions expect **local slab arrays** (rank-local portions) shaped `(4, n0, N, N, N)` or 3D arrays for slice-based routines. They perform MPI reductions and return aggregated results on rank 0.

```
# diagnostics4d.py
import numpy as np
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def compute_vorticity_4d(u_local, dx=1.0):
    """
    Compute 6-component vorticity tensor fields locally.
    u_local shape: (4, n0, N, N, N)
    Returns dict of omega_ij arrays with same local slab shape (n0,N,N,N)
    """
    # Finite-difference central derivatives in each axis using numpy roll
(periodic).
    # Assumes uniform grid spacing dx and full global periodicity (works on local
slabs if ghost cells present).
    comps = {}
    # local arrays
    u0,u1,u2,u3 = u_local
    # helper derivative using spectral or central differences; here central
difference (needs ghost cells)
    def deriv(arr, axis):
        return (np.roll(arr, -1, axis=axis) - np.roll(arr, 1, axis=axis)) / (2*dx)
    # index mapping for axes in local arrays: 0->local n0 axis, 1->x1,2->x2,3->x3
    # compute omega_01 = d0 u1 - d1 u0  etc.
    omega = {}
    omega['01'] = deriv(u0, 0) * 0.0 + deriv(u1, 0) - deriv(u0, 1)  # careful
ordering (example)
    # Better: compute all with loop
    U = [u0, u1, u2, u3]
    for i in range(4):
        for j in range(i+1, 4):
            key = f"{i}{j}"
            di_uj = deriv(U[j], i)
            dj_ui = deriv(U[i], j)
            omega[key] = di_uj - dj_ui
    return omega  # dict of six arrays

def compute_vorticity_norm(omega_dict):
    # pointwise sqrt(1/2 sum omega_ij^2)
    s = None
    for key, arr in omega_dict.items():
        if s is None:
            s = 0.5 * (arr**2)
        else:
            s += 0.5 * (arr**2)
    norm = np.sqrt(s)
    return norm

def compute_circulation_on_patch(omega_dict, plane=(0,1), slab_indices=None,
dx=1.0):
    """
    Integrate omega_{ij} over a grid-aligned 2-surface patch in plane (i,j).
    slab_indices: tuple specifying index ranges for local slab in axes (i,j), e.g.
slices
    We assume the 2-surface is axis-aligned and described by fixing the other two
coordinates.
    Returns global scalar Gamma (reduced to rank 0).
    """
    i,j = plane
    key = f"{i}{j}" if i<j else f"{j}{i}"
    arr = omega_dict[key]  # local slab values
    # choose integration region: full local slab or provided
    if slab_indices is None:
        local_sum = np.sum(arr) * (dx*dx)
    else:
        sl = arr[slab_indices]
        local_sum = np.sum(sl) * (dx*dx)
    total = comm.allreduce(local_sum, op=MPI.SUM)
    if rank == 0:
        return total
    return None

# -------- 3-slice lobes counting (connected components) ----------
def count_3slice_lobes(rho_local, axis0_start, threshold, slice_x4_index=None):
    """
    rho_local: local slab of scalar 'density' or 'rho_q' with shape (n0, N, N, N)
    axis0_start: global start index of this slab along axis0.
    threshold: scalar threshold to define 'low-density core' (e.g. density <
threshold)
    slice_x4_index: optional integer global index for x4 slice to analyze (if None
use middle)
    Returns (global_lobe_counts_per_slice dict) on rank 0.
    """
    n0 = rho_local.shape[0]
    N = rho_local.shape[1]
    # determine which local index (if any) corresponds to requested slice
    if slice_x4_index is None:
        # choose central global slice
        # compute global mid index
        global_mid = N//2
        slice_idx = global_mid
    else:
        slice_idx = slice_x4_index
    # check if slice is in this slab
    local_idx = slice_idx - axis0_start
    local_counts = {}
    if 0 <= local_idx < n0:
        # extract 3D array of shape (N,N,N) for that slice
        arr3 = rho_local[local_idx]
        mask = arr3 < threshold
        # connected-component labelling in 3D (simple BFS/stack)
        # Implement simple stack-based labelling (inefficient but fine for moderate N)
        visited = np.zeros_like(mask, dtype=bool)
        counts = 0
        from collections import deque
        for i in range(N):
            for j in range(N):
                for k in range(N):
                    if mask[i,j,k] and not visited[i,j,k]:
                        # flood fill
                        counts += 1
                        dq = deque([(i,j,k)])
                        visited[i,j,k] = True
                        while dq:
                            a,b,c = dq.popleft()
                            # 6-neighborhood
                            for da,db,dc in ((1,0,0),(-1,0,0),(0,1,0),(0,-1,0),(0,0,1),(0,0,-1)):
                                aa = (a+da) % N
                                bb = (b+db) % N
                                cc = (c+dc) % N
                                if mask[aa,bb,cc] and not visited[aa,bb,cc]:
                                    visited[aa,bb,cc] = True
                                    dq.append((aa,bb,cc))
        local_counts = counts
    else:
        local_counts = 0
    # reduce to global sum (only rank with slice contributes)
    total_counts = comm.allreduce(local_counts, op=MPI.SUM)
    if rank == 0:
        return total_counts
    return None

# -------- form factor computation on 3-slice ----------
def compute_form_factor(rho_local, axis0_start, qvecs=None):
    """
    rho_local: (n0, N, N, N) local scalar on slab
    axis0_start: global start index of slab
    qvecs: list of q vectors (3-tuples) in units of grid k-space; if None compute radial shell power
    Returns dict on rank 0: radial F(q) or specific q values
    """
    # Gather the chosen 3-slice (middle) to rank0, or compute distributed FFT if available
    global_N = rho_local.shape[1]
    target_slice = global_N // 2
    local_idx = target_slice - axis0_start
    if 0 <= local_idx < rho_local.shape[0]:
        # we have the slice
        slice3 = rho_local[local_idx]
    else:
        # create empty slice locally
        slice3 = np.zeros((global_N, global_N, global_N))
    # gather slices to rank0
    full_slice = None
    slices = comm.gather(slice3, root=0)
    if rank == 0:
        # stitch by order of ranks (we used slab on axis0; only one rank contains
the slice)
        # find the nonzero slice
        for s in slices:
            if np.any(s):
                full_slice = s.copy()
                break
        # compute FFT and radial shells
        Fk = np.fft.fftn(full_slice)
        Pk = np.abs(Fk)**2
        # radial average
        k1 = np.fft.fftfreq(global_N, d=1.0/global_N)
        kx,ky,kz = np.meshgrid(k1,k1,k1, indexing='ij')
        kr = np.sqrt(kx**2 + ky**2 + kz**2)
        kr_flat = kr.ravel()
        P_flat = Pk.ravel()
        nbins = min(50, global_N//2)
        bins = np.linspace(0, kr_flat.max(), nbins+1)
        inds = np.digitize(kr_flat, bins)
        radial = np.array([P_flat[inds==i].mean() if np.any(inds==i) else 0.0 for i
in range(1,nbins+1)])
        return {'radial_bins': bins[1:], 'radial_power': radial}
    else:
        return None

# ---------- helper to compute charge-proxy rho_q ----------
def compute_charge_proxy(u_local, omega_dict_local, alpha=1.0):
    """
    One pragmatic proxy: build P^i = eps^{ijkl} u_j omega_{kl} and take divergence
-> scalar.
    Return rho_q_local shape (n0,N,N,N)
    """
    # build eps tensor operations manually: eps^{ijkl} contraction
    # For 4D Levi-Civita, we'll compute P component by direct expansion (6 terms)
    u0,u1,u2,u3 = u_local
    w = omega_dict_local  # keys '01','02','03','12','13','23' etc (note mapping)
    # safe mapping
    # Build P0 = eps^{0jkl} u_j omega_{kl} = + (u1*w23 - u2*w13 + u3*w12)  (up to
sign conventions)
    P0 = u1 * w['23'] - u2 * w['13'] + u3 * w['12']
    P1 = -u0 * w['23'] + u2 * w['03'] - u3 * w['02']
    P2 = u0 * w['13'] - u1 * w['03'] + u3 * w['01']
    P3 = -u0 * w['12'] + u1 * w['02'] - u2 * w['01']
    # divergence of P: approximate with central differences; here we do simple
spectral-like central differences using roll
    def deriv(arr, axis, dx=1.0):
        return (np.roll(arr, -1, axis=axis) - np.roll(arr, 1, axis=axis)) / (2*dx)
    div = deriv(P0, 0) + deriv(P1, 1) + deriv(P2, 2) + deriv(P3, 3)
    rho_q_local = alpha * div
    return rho_q_local

# Example usage pattern (in your MPI job):
# - Load local slab u (4,n0,N,N,N) from the HDF5 snapshot.
# - omega = compute_vorticity_4d(u_local)
# - omega_norm = compute_vorticity_norm(omega)
# - gamma = compute_circulation_on_patch(omega, plane=(0,1))
# - rho_q = compute_charge_proxy(u_local, omega)
# - lobes = count_3slice_lobes(rho_q, axis0_start, threshold=some_value)
# - Fq = compute_form_factor(rho_q, axis0_start)
```

Quick implementation plan & tips

- **Test locally first.** Run `init4d_mpi.py` with `-n 1` and `N=24` to generate small HDF5 initial conditions. Use the diagnostic functions on that dataset to verify shapes and basic outputs.
- **MPI FFT:** Installing `mpi4py_fft` (which depends on an underlying FFT library and MPI) is recommended for large runs — it handles distributed transforms and data layout efficiently. If you prefer C/C++ backend, libraries like P3DFFT or PFFT are also options; the high-level pattern above remains the same.
- **Ghost cells:** The finite-difference derivatives in `diagnostics4d.py` assume periodic BCs and access to neighbor cells via `np.roll`. If your slab decomposition separates ranks (no ghost cells), either:
  - use spectral derivative (global FFT) with a distributed FFT library, or
  - exchange halo/ghost-face data between neighboring ranks before computing derivatives.
- **I/O format:** HDF5 with MPI driver is stable and widely used. Include metadata attributes (grid spacing, axis0_start, N, L) so diagnostics know the global geometry.
- **Performance:** For serious grid sizes (≥64^4), you will need multi-node runs and distributed FFTs — single-node memory will be insufficient. Use float32 for initial exploration, float64 for converged runs.
- **Validation:** Start by re-running Exp1 (single-lobe seed), track vorticity_norm and lobe counts as you vary viscosity. Use the radial form-factor to check "pointlike vs structured" behavior.

If you want, next I can:

- Produce a **compact MPI halo-exchange snippet** (for 4D slab decomposition) so you can compute derivatives locally without global FFTs, or
- Convert the diagnostics into **numba-accelerated** versions for faster local processing before MPI reductions.

Which would you like next?

========

My prompt to GPT-5:

I would like both.

Its response:

Nice — below are **both** pieces you asked for, ready to drop into your MPI+Python workflow.

1. **4-D MPI halo-exchange snippet** (slab decomposition along axis 0) — minimal, robust pattern using `mpi4py` `Sendrecv` to populate one-cell ghost layers so local finite-difference derivatives can be computed without global FFTs.
2. **Numba-accelerated diagnostics** — two `@njit` routines (a) compute the six vorticity components and the vorticity norm for a local slab, and (b) compute the simple charge-proxy `rho_q = α · div(P)` where `P^i = eps^{ijkl} u_j omega_{kl}` as previously described. These are designed to operate *on local slabs with ghost layers already present* (so the caller must first call the halo exchange).

Both snippets are written to fit into the code you already have (same array shapes: `u_local` as `(4, n0_local+2, N, N, N)` where the `+2` in axis0 includes one ghost cell at start and end). I include explanatory comments and usage notes.

A — 4-D MPI halo-exchange (slab decomposition on axis0)

This pattern assumes:

- Global grid size `N` (each axis length).
- Slab decomposition along axis 0 (the first spatial axis), i.e., each rank stores a contiguous block of `n0_local` *interior* cells, and we allocate 1 ghost cell at each end so our local array axis0 length = `n0_local + 2`.
- Periodic boundary conditions (wrap-around exchange). If you want non-periodic, adapt sends/receives and ghost fill values accordingly.

```
# halo_exchange_4d.py
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def compute_counts_and_starts(N, size):
    base = N // size
    extra = N % size
    counts = [(base + (1 if r < extra else 0)) for r in range(size)]
    starts = [sum(counts[:r]) for r in range(size)]
    return counts, starts

def exchange_ghost_cells_axis0(u_local, N, counts, starts, axis0_start):
    """
    u_local: numpy array with shape (4, n0_local+2, N, N, N)
             where index 0..n0_local+1 includes ghosts at 0 and -1 positions.
    counts: list of interior counts per rank
    starts: list of starting global index for each rank (interior start)
    axis0_start: this rank's interior start index (global)
    Returns: u_local updated with ghost layers filled from neighbor ranks
(periodic).
    """
    # neighbor ranks (periodic)
    prev_rank = (rank - 1) % size
    next_rank = (rank + 1) % size

    n0_local = counts[rank]
    # indices in local array for interior block: [1 : 1+n0_local)
    # ghost-left index: 0, ghost-right index: 1 + n0_local
    send_left = u_local[:, 1, :, :, :].copy()           # leftmost interior slice
(to prev's right ghost)
    send_right = u_local[:, 1 + n0_local - 1, :, :, :].copy()  # rightmost interior
slice (to next's left ghost)

    # Post non-blocking sends/recv or blocking Sendrecv for simplicity
    # Send my leftmost interior to prev's right ghost; receive prev's rightmost
interior into my left ghost
    recv_left = np.empty_like(send_left)
    req1 = comm.Isend(send_left, dest=prev_rank, tag=11)
    req2 = comm.Irecv(recv_left, source=prev_rank, tag=12)
    # Send my rightmost interior to next's left ghost; receive next's leftmost
interior into my right ghost
    recv_right = np.empty_like(send_right)
    req3 = comm.Isend(send_right, dest=next_rank, tag=12)
    req4 = comm.Irecv(recv_right, source=next_rank, tag=11)

    # Wait for transfers
    MPI.Request.Waitall([req1, req2, req3, req4])

    # place received slices into ghost locations
    u_local[:, 0, :, :, :] = recv_left        # left ghost
    u_local[:, 1 + n0_local, :, :, :] = recv_right   # right ghost

    # Done: local slab now has ghost cells available for central-difference ops
    return u_local

# ---------------- Example usage ----------------
# Suppose you previously created a local array u_local_interior shape (4, n0_local,
N, N, N)
# Expand to have ghost layers:
# u_local = np.zeros((4, n0_local+2, N, N, N), dtype=u_local_interior.dtype)
# u_local[:, 1:1+n0_local, :, :, :] = u_local_interior
# axis0_start = starts[rank]  # computed from compute_counts_and_starts
# counts, starts = compute_counts_and_starts(N, size)
# u_local = exchange_ghost_cells_axis0(u_local, N, counts, starts, axis0_start)
# Now call local derivative routines that expect ghost cells.
```

Notes & tips

- The tags `11`/`12` must be unique per communicator usage; if you have other sends, pick different tags or include rank in tag composition.
- If your decomposition differs (e.g., different axis slab), adapt neighbor definition and which slices to send.
- For **multi-cell** ghost layers (e.g., 2+ ghost cells), send/receive the corresponding number of slices (send block `[:, 1:1+nghost]` and receive into `[:, 0:nghost]`, etc.)
- For non-periodic BCs, send/recv only where neighbor exists and set ghost cells to boundary values otherwise.

B — Numba-accelerated diagnostics

These routines expect the **ghost-filled** local slab arrays described above:

- `u_local` shape: `(4, n0_local+2, N, N, N)` (dtype `float64` or `float32`)
- `omega_out` and `rho_q_out` will be computed with the same interior shape `(n0_local, N, N, N)` (ghosts trimmed off on output).

We use `numba.njit(parallel=True)` for speed. The connected-component lobe counting is left as a Python routine or can use `scipy.ndimage.label` (not numba); it's more robust to rely on that or a C/C++ library for large grids — I include a simple Python BFS fallback after the numba ops.

> **Install:** `pip install numba` (use same Python env as MPI). Numba accel is local per process.

```
# numba_diagnostics.py
import numpy as np
from numba import njit, prange

# ---- Numba: compute vorticity components and vorticity norm ----
@njit(parallel=True)
def compute_vorticity_and_norm_numba(u, dx, out_omega01, out_omega02, out_omega03,
                                     out_omega12, out_omega13, out_omega23,
out_norm):
    """
    u: shape (4, n0+2, N, N, N) with ghosts on axis0.
    dx: grid spacing (assumed same on all axes)
    out_* arrays: preallocated arrays with shape (n0, N, N, N) for interior only
    Finite central differences using neighbor offsets (periodic handled by ghost
cells).
    """
    # locals for indexing
    _, n0g, N, _, _ = u.shape
    n0 = n0g - 2  # interior count
    # helper offsets: axis mapping:
    # axis indices for u: 0->component, 1->axis0 (with ghosts), 2->axis1, 3->axis2,
4->axis3
    for i0 in prange(n0):
        gi0 = i0 + 1  # global index including ghost offset
        for i1 in range(N):
            im1 = (i1 - 1) % N
            ip1 = (i1 + 1) % N
            for i2 in range(N):
                im2 = (i2 - 1) % N
                ip2 = (i2 + 1) % N
                for i3 in range(N):
                    im3 = (i3 - 1) % N
                    ip3 = (i3 + 1) % N

                    # central derivatives
                    # d/dx0 u_j  ~ (u[j,gi0+1,i1,i2,i3] -
u[j,gi0-1,i1,i2,i3])/(2dx)
                    # d/dx1 u_j  ~ (u[j,gi0,ip1,i2,i3] - u[j,gi0,im1,i2,i3])/(2dx)
                    # etc.

                    # compute partials we need for omega_ij (6 components)
                    # omega01 = d0 u1 - d1 u0
                    d0_u1 = (u[1, gi0+1, i1, i2, i3] - u[1, gi0-1, i1, i2,
i3])/(2.0*dx)
                    d1_u0 = (u[0, gi0, ip1, i2, i3] - u[0, gi0, im1, i2,
i3])/(2.0*dx)
                    o01 = d0_u1 - d1_u0

                    # omega02 = d0 u2 - d2 u0
                    d0_u2 = (u[2, gi0+1, i1, i2, i3] - u[2, gi0-1, i1, i2,
i3])/(2.0*dx)
                    d2_u0 = (u[0, gi0, i1, ip2, i3] - u[0, gi0, i1, im2,
i3])/(2.0*dx)
                    o02 = d0_u2 - d2_u0

                    # omega03 = d0 u3 - d3 u0
                    d0_u3 = (u[3, gi0+1, i1, i2, i3] - u[3, gi0-1, i1, i2,
i3])/(2.0*dx)
                    d3_u0 = (u[0, gi0, i1, i2, ip3] - u[0, gi0, i1, i2,
im3])/(2.0*dx)
                    o03 = d0_u3 - d3_u0

                    # omega12 = d1 u2 - d2 u1
                    d1_u2 = (u[2, gi0, ip1, i2, i3] - u[2, gi0, im1, i2,
i3])/(2.0*dx)
                    d2_u1 = (u[1, gi0, i1, ip2, i3] - u[1, gi0, i1, im2,
i3])/(2.0*dx)
                    o12 = d1_u2 - d2_u1

                    # omega13 = d1 u3 - d3 u1
                    d1_u3 = (u[3, gi0, ip1, i2, i3] - u[3, gi0, im1, i2,
i3])/(2.0*dx)
                    d3_u1 = (u[1, gi0, i1, i2, ip3] - u[1, gi0, i1, i2,
im3])/(2.0*dx)
                    o13 = d1_u3 - d3_u1

                    # omega23 = d2 u3 - d3 u2
                    d2_u3 = (u[3, gi0, i1, ip2, i3] - u[3, gi0, i1, im2,
i3])/(2.0*dx)
                    d3_u2 = (u[2, gi0, i1, i2, ip3] - u[2, gi0, i1, i2,
im3])/(2.0*dx)
                    o23 = d2_u3 - d3_u2
                    # store
                    out_omega01[i0, i1, i2, i3] = o01
                    out_omega02[i0, i1, i2, i3] = o02
                    out_omega03[i0, i1, i2, i3] = o03
                    out_omega12[i0, i1, i2, i3] = o12
                    out_omega13[i0, i1, i2, i3] = o13
                    out_omega23[i0, i1, i2, i3] = o23

                    # compute norm = sqrt(0.5 * sum omega_ij^2)
                    s = 0.5*(o01*o01 + o02*o02 + o03*o03 + o12*o12 + o13*o13 +
o23*o23)
                    out_norm[i0, i1, i2, i3] = np.sqrt(s)

    return  # results written in-place

# ---- Numba: compute charge-proxy rho_q = alpha * div(P) with P^i = eps^{ijkl} u_j
omega_{kl} ----
@njit(parallel=True)
def compute_charge_proxy_numba(u, o01, o02, o03, o12, o13, o23, dx, alpha,
rho_q_out):
    """
    Inputs:
      u: (4, n0+2, N, N, N) including ghosts
      oij: interior omega arrays shape (n0, N, N, N)
      rho_q_out: preallocated interior array shape (n0, N, N, N) (output)
    Formula used (consistent with earlier exposition):
      P0 =  u1 * o23 - u2 * o13 + u3 * o12
      P1 = -u0 * o23 + u2 * o03 - u3 * o02
      P2 =  u0 * o13 - u1 * o03 + u3 * o01
      P3 = -u0 * o12 + u1 * o02 - u2 * o01
      rho_q = alpha * (d0 P0 + d1 P1 + d2 P2 + d3 P3)
    Note: derivatives use central finite difference; ghost cells required for
axis0.
    """
    _, n0g, N, _, _ = u.shape
    n0 = n0g - 2
    for i0 in prange(n0):
        gi0 = i0 + 1
        for i1 in range(N):
            im1 = (i1 - 1) % N
            ip1 = (i1 + 1) % N
            for i2 in range(N):
                im2 = (i2 - 1) % N
                ip2 = (i2 + 1) % N
                for i3 in range(N):
                    im3 = (i3 - 1) % N
                    ip3 = (i3 + 1) % N

                    # get u components at required positions
                    u0 = u[0, gi0, i1, i2, i3]
                    u1 = u[1, gi0, i1, i2, i3]
                    u2 = u[2, gi0, i1, i2, i3]
                    u3 = u[3, gi0, i1, i2, i3]

                    # omega at interior position
                    w01 = o01[i0, i1, i2, i3]
                    w02 = o02[i0, i1, i2, i3]
                    w03 = o03[i0, i1, i2, i3]
                    w12 = o12[i0, i1, i2, i3]
                    w13 = o13[i0, i1, i2, i3]
                    w23 = o23[i0, i1, i2, i3]

                    # compute P components at center
                    P0 = u1 * w23 - u2 * w13 + u3 * w12
                    P1 = -u0 * w23 + u2 * w03 - u3 * w02
                    P2 = u0 * w13 - u1 * w03 + u3 * w01
                    P3 = -u0 * w12 + u1 * w02 - u2 * w01

                    # derivatives: d0 P0 uses ghosts at gi0+1 and gi0-1 (access u's
ghost areas as needed)
                    # but P arrays are not available at ghost indices; approximate
d0 P0 via computing P0 at shifted indices:
                    # compute P0 at gi0+1 center: need u at gi0+1 index and omega
at i0+1 index
                    # for simplicity here we compute central diff of P components
by reconstructing P at neighbors
                    # compute neighbor P0_plus (approx)
                    # P0_plus uses u components at gi0+1 and w_ij at i0+1
                    u0_p = u[0, gi0+1, i1, i2, i3]; u1_p = u[1, gi0+1, i1, i2, i3]
                    u2_p = u[2, gi0+1, i1, i2, i3]; u3_p = u[3, gi0+1, i1, i2, i3]
                    # neighbor omega values - we don't have oij at i0+1 index in
inputs? but we do: o arrays are interior and aligned to i0 indices.
                    # So need to fetch from arrays at i0+1 (exists if interior),
else approximate via same position.
                    # For correctness, ensure o arrays include neighbor interior
cells or compute a ghosted omega before.
                    # For now assume o arrays are defined for index i0 and i0+1 etc
(i.e., caller prepared ghosted omega). Simpler: compute derivatives in axis0 by
using u's neighbor velocities and interior omegas as approximation.
                    # To keep code concise, we'll compute spatial derivatives for P
components using u neighbors and current omegas (approx).
                    # d0 P0 approx:
                    # compute P0_minus and P0_plus using available u at gi0-1 and
gi0+1 and omega at same i0 (approx)
                    u0_m = u[0, gi0-1, i1, i2, i3]; u1_m = u[1, gi0-1, i1, i2, i3]
                    u2_m = u[2, gi0-1, i1, i2, i3]; u3_m = u[3, gi0-1, i1, i2, i3]
                    P0_p = u1_p * w23 - u2_p * w13 + u3_p * w12
                    P0_m = u1_m * w23 - u2_m * w13 + u3_m * w12
                    d0P0 = (P0_p - P0_m) / (2.0*dx)

                    # d1 P1: need P1 at ip1 and im1 (use u at same gi0)
                    u0_ip1 = u[0, gi0, ip1, i2, i3]; u1_ip1 = u[1, gi0, ip1, i2,
i3]
                    u2_ip1 = u[2, gi0, ip1, i2, i3]; u3_ip1 = u[3, gi0, ip1, i2,
i3]
                    P1_ip1 = -u0_ip1 * w23 + u2_ip1 * w03 - u3_ip1 * w02
                    u0_im1 = u[0, gi0, im1, i2, i3]; u1_im1 = u[1, gi0, im1, i2,
i3]
                    u2_im1 = u[2, gi0, im1, i2, i3]; u3_im1 = u[3, gi0, im1, i2,
i3]
                    P1_im1 = -u0_im1 * w23 + u2_im1 * w03 - u3_im1 * w02
                    d1P1 = (P1_ip1 - P1_im1) / (2.0*dx)

                    # d2 P2
                    u0_ip2 = u[0, gi0, i1, ip2, i3]; u1_ip2 = u[1, gi0, i1, ip2,
i3]
                    u2_ip2 = u[2, gi0, i1, ip2, i3]; u3_ip2 = u[3, gi0, i1, ip2,
i3]
```

```
                    P2_ip2 = u0_ip2 * w13 - u1_ip2 * w03 + u3_ip2 * w01
                    u0_im2 = u[0, gi0, i1, im2, i3]; u1_im2 = u[1, gi0, i1, im2, i3]
                    u2_im2 = u[2, gi0, i1, im2, i3]; u3_im2 = u[3, gi0, i1, im2, i3]
                    P2_im2 = u0_im2 * w13 - u1_im2 * w03 + u3_im2 * w01
                    d2P2 = (P2_ip2 - P2_im2) / (2.0*dx)

                    # d3 P3
                    u0_ip3 = u[0, gi0, i1, i2, ip3]; u1_ip3 = u[1, gi0, i1, i2, ip3]
                    u2_ip3 = u[2, gi0, i1, i2, ip3]; u3_ip3 = u[3, gi0, i1, i2, ip3]
                    P3_ip3 = -u0_ip3 * w12 + u1_ip3 * w02 - u2_ip3 * w01
                    u0_im3 = u[0, gi0, i1, i2, im3]; u1_im3 = u[1, gi0, i1, i2, im3]
                    u2_im3 = u[2, gi0, i1, i2, im3]; u3_im3 = u[3, gi0, i1, i2, im3]
                    P3_im3 = -u0_im3 * w12 + u1_im3 * w02 - u2_im3 * w01
                    d3P3 = (P3_ip3 - P3_im3) / (2.0*dx)

                    divP = d0P0 + d1P1 + d2P2 + d3P3
                    rho_q_out[i0, i1, i2, i3] = alpha * divP

    return  # in-place write
```

Important caveats & usage notes

- The `compute_vorticity_and_norm_numba` routine assumes ghost layers are present on axis0 (so central differences never sample out-of-bounds). It uses periodic wrapping for axes 1–3 via `% N`.
- The `compute_charge_proxy_numba` does a simplified neighbor evaluation for axis0 derivatives — to be formally correct you should precompute ghosted `omega` arrays as well (i.e., extend `oij` arrays with one ghost layer in axis0) so the derivative uses consistent centered neighbors. The code currently approximates neighbor omegas by using interior omegas; for best accuracy, compute `omega` with ghost layers too (call `compute_vorticity_and_norm_numba` on a slightly extended domain).
- Numba requires arrays to be C-contiguous for best performance. Ensure inputs are `np.ascontiguousarray(...)`.
- For large `N` this code is memory- and CPU-heavy but will be substantially faster than pure Python loops.

Connected-component (3-slice) lobe counting — options

For reliable and fast connected-component labelling on 3D slices:

1. **Preferred:** `scipy.ndimage.label` (fast, compiled C). Usage (rank that has the slice):

```
from scipy.ndimage import label
```

```
mask = (rho_slice < threshold)   # boolean
labeled, num = label(mask)  # default connectivity=1 (6-neighbor); choose structure
for 18- or 26-neighbor
```

2. **If SciPy unavailable:** use the Python BFS flood-fill version provided earlier — works but slower. For moderate `N` it will still be OK per-rank.
3. **Parallel approach:** if slices are split across ranks (rare for a single x4 slice), you'll need a distributed connected-component algorithm — more complex; avoid by ensuring a single rank contains the full slice (use slab decomposition axis not aligned with slice axis), or gather the needed slice to rank 0 before labelling.

How to plug everything together (usage recipe)

1. **Set up local arrays** `u_interior` shape `(4, n0_local, N, N, N)` (no ghosts yet).
2. **Allocate ghosted array** `u_local = np.zeros((4, n0_local+2, N, N, N), dtype=u_interior.dtype)` and fill interior `u_local[:,1:1+n0_local,:,:,:] = u_interior`.
3. **Call** `exchange_ghost_cells_axis0(u_local, N, counts, starts, axis0_start)` to populate ghosts.
4. **Allocate outputs** for omegas and norm: shape `(n0_local, N, N, N)` for each.
5. **Call** `compute_vorticity_and_norm_numba(u_local, dx, ...)` to fill omega arrays and norm.
6. **(Optional)** create ghosted `omega` arrays if needed, or ensure derivatives in axis0 use ghost layers.
7. **Call** `compute_charge_proxy_numba(u_local, o01, o02, ..., dx, alpha, rho_q_out)` to get `rho_q`.
8. **Gather slice** of `rho_q` to rank 0 (if needed) and call `scipy.ndimage.label` or Python BFS to count lobes.
9. **MPI reduce** any global integrals (energy, circulation) as needed.

If you want next: I can

- adapt the halo exchange to **2-cell ghost layers** and provide exact send/recv shapes, or
- produce a **Cython/C++** implementation of the connected-component labeller for large 3D slices (fast and MPI-friendly).

Which do you want me to produce next?