Scaling your science with parallel computing

Shao-Ching Huang

2025-10-17

Contents

Workshop series:

1. Foundations for sustainable research software (October 9)
2. **Scaling your science with parallel computing (October 16)**
3. Accelerating your code with GPUs (October 30)

Resources:

- Workshop materials [link]
- UCLA Office of Advanced Research Computing [link]
- Hoffman2 Cluster [link]

Introuction

Recap from workshop 1

- Structure research code with **functions** and **classes** that follow single-responsibility principles, use clear signatures, and handle errors deliberately.
- Lean on object-oriented patterns (instance, class, and static methods, inheritance and abstraction) to encapsulate scientific workflows.
- Use modern Python features like dataclasses, decorators, and type aliases to simplify programming, reduce boilerplate while keeping intent explicit.
- Package projects with a `src/` layout, `pyproject.toml`, semantic versioning, and tagged releases so collaborators can install the exact code you share.
- Lock down reproducibility by recording dependencies, capturing environments (conda, Docker, Apptainer), and looping through the write-test-release development cycle.

Goals for today

- Workshop 2 in the Advanced Python Series: shift your Python workflows from a single laptop to HPC clusters and supercomputers
- Today's focus: understanding when and why to parallelize, MPI fundamentals, hands-on practice with mpi4py, PETSc workflows, and real-world examples
  - Introduce distributed-memory parallel computing using MPI
  - Use `mpi4py` to access MPI functionality from Python
  - Introduce PETSc for scaling research applications onto HPC clusters (PDE solvers, climate models, MRI reconstruction, genome analytics, multi-omics pipelines, etc.)
  - Use `petsc4py` to access PETSc functionality from Python (built on `mpi4py`)
  - Walk through sample code demonstrating these methods in real applications

- Prerequisites: Python and MPI environment ready; familiarity with Python, NumPy, and basic MPI concepts is helpful

Why parallel computing matters

Parallel computing lets you tackle simulations and datasets that exceed single-machine memory or time limits (higher-resolution PDEs, larger ML models, whole-genome analyses). Distributing work across many cores or nodes reduces wall-clock time, enabling quicker iterations and more experiments. This makes possible ensemble runs, uncertainty quantification, parameter sweeps, and higher-fidelity models that reveal phenomena single-node runs cannot.

Using cluster resources effectively (many moderate-cost nodes) can be cheaper and more energy-efficient than a single oversized machine. Scalable pipelines and reproducible deployments on HPC clusters support multi-user workflows and reliable, automated processing in research environments.

Datasets and simulations now outgrow a single CPU core. Hardware roadmaps add cores, cache, and accelerators rather than higher clocks. Parallel workflows turn extra hardware into faster, reproducible results. Everyday laptops and desktops ship with multi-core CPUs, so modernizing code to exploit those cores is increasingly necessary.

Key Concepts

- Shared-memory vs distributed-memory (single workstation vs cluster).
- Embarrassingly parallel vs tightly coupled workloads.
- Speedup estimates
  - Overall speedup is limited by the fraction of the program that must run serially (Amdahl's Law): even if the parallel portion is sped up arbitrarily, the serial part bounds the maximum achievable speedup.

Embarassingly parallel example

Climate researchers run 1,000 weather ensemble members independently to bracket uncertainty. Each simulation uses different initial conditions, needs **zero communication**, and can saturate every core on a workstation or a cluster queue.

Tightly coupled example

The same research group runs a 3D Navier–Stokes solver where every timestep must exchange boundary values with neighboring subdomains. Latency and bandwidth now determine how fast those tightly coupled processes can march forward together.

Parallel Models

Shared memory

- threads share RAM on one machine
- Python tools: `threading`, `concurrent.futures`, `numpy`, `numba`.

Distributed memory

- processes own RAM and exchange messages
- Requires explicit communication among processes
- a.k.a. MPI-style parallelism
- Python tools: `mpi4py`, `petsc4py`, MPI-enabled `Dask`.

Hybrid model

- Shared-memory and distributed-memory models are not mutually exclusive — they are frequently combined in real applications.
- Use shared-memory parallelism (threads, OpenMP, or multiprocessing with shared arrays) to parallelize work within a single compute node (many cores), and
- Use distributed-memory parallelism (MPI, distributed Dask, PETSc) to scale across multiple nodes in an HPC cluster.
- Hybrid approaches (e.g., MPI between nodes + multithreading or vectorized libraries within a node) are common and often give the best performance on modern clusters.

We will focus on MPI-style parallelism in this workshop.

`mpi4py` is our Python's bridge to distributed-memory computing.

Examples of MPI applications

- 3D MRI/CT reconstruction pipelines that distribute FFTs and solvers.
- Cardiac and vascular simulations using PETSc domain decomposition.
- Large molecular dynamics workloads (NAMD, GROMACS) with MPI force exchange.
- Large-scale (fine resolution, large region) earthquake simulations.
- Cryo-EM processing pipelines (RELION, cryoSPARC) accelerating EM refinement.
- MPI-enabled sequence alignment and multi-omics analytics (HipMer, mpiBLAST).

And many other domain-specific, tightly coupled or data-parallel workflows.

Takeaways

Use the right mix: combine shared-memory parallelism inside a node with distributed-memory (MPI) across nodes when appropriate. Match the parallel strategy to where the data lives and how tasks communicate (data locality matters). Start with a simple model, profile early, and iterate. Profiling catches I/O and communication bottlenecks before large-scale runs.

Prefer high-level, well-supported tools: `numpy`, `numba`, or OpenMP for per-node performance; `mpi4py`, `petsc4py`, or Dask for scaling across nodes. Test and scale incrementally: verify correctness and performance on a single node before moving to multi-node runs.

Message Passing Interface (MPI)

What Is MPI?

- MPI stands for Message Passing Interface, a standard for processes to exchange data by sending messages.
- Each process runs its own program instance and shares results through explicit communication.
- Designed for high-performance clusters, MPI works on laptops, workstations, and supercomputers.
- Different languages (C, C++, Fortran, Python, R, Julia, etc.) follow the same communication rules.

What Is a Process?

- A process is an independent program instance with its own memory space and resources.
- MPI launches many processes that run the same code on different slices of the data. Processes communicate by passing messages instead of sharing memory directly.

What Is a Rank?

- A rank is the ID number MPI assigns to each process inside a communicator.
- Ranks let you address a specific process when sending messages.
- Rank 0 can act as coordinator or root, while other ranks (1, 2, …) handle parallel work, but this is not always the case.

What Is `MPI_COMM_WORLD`?

- This is the default communicator created at startup, containing every MPI process in the job.
- It provides `Get_rank()` and `Get_size()` for global coordination. This is a good starting point; create custom communicators later for subgroups.

Where MPI Came From

- In the early 1990s, the community unified competing libraries into MPI-1 (1994).
- MPI-2 (1997) added dynamic processes and parallel I/O.
- MPI-3 (2012) added nonblocking collectives, shared windows, and more.
- The standard API covers C, C++, and Fortran.
- Vendors implement the same calls for portability.
- Popular implementations: MPICH, Open MPI, MVAPICH2
- Python uses `mpi4py` to wrap the C bindings with a clean, NumPy-friendly layer.

Why MPI Scales Everywhere

- MPI provides a common language for distributed-memory systems.
- One specification runs portably from laptops to supercomputers.

- Well-written MPI code moves from laptop tests to supercomputer runs without rewrites.
- `mpi4py` keeps Python productive while tapping into the same MPI foundation.

Core Vocabulary

- Communicators define who can talk to whom; start with `MPI.COMM_WORLD`.
- Rank identifies each process; size reports how many are participating.
- Custom communicators carve out subgroups for focused collaboration.

Point-to-Point Communication

- One sender talks to one receiver; each "point" is a process (rank). Blocking `Send` and `Recv` complete only when buffers are safe to reuse.
- Nonblocking `Isend` and `Irecv` overlap communication with computation; finish via `Wait` or `Test`.
- Tags label message streams so receives can pull exactly what they expect.

Point-to-Point Examples

- Boundary exchange: rank 0 sends edge cells to rank 1 (`comm.send(boundary, dest=1)`), then receives updates back.
- Ping-pong timing: two ranks trade a token (`Send`/`Recv`) to measure latency.
- Task farm: workers `Recv` work units from rank 0, process them, then `Send` results home.

Collective Communication

- Broadcast: one root rank shares identical data with everyone.
- Scatter: root distributes distinct chunks to each rank.
- Gather: ranks collect results back to the root; `Allgather` returns a copy to every rank.
- Reduce and Allreduce: combine values (sum, max, etc.) across ranks, optionally sharing the result with all.
- Barrier: everyone waits until all ranks arrive before continuing.

Computing Environment Setup

What You Need

- **MPI implementation** (MPICH, Open MPI, Intel MPI, MVAPICH2) with compiler wrappers (`mpicc`, `mpiexec`). I will use MPICH in this workshop.
- **Python 3.9+** with virtual environments (`venv` or `conda`).
- **mpi4py** to access MPI from Python.
- **PETSc + petsc4py** for scalable linear algebra, nonlinear solvers.
- **SLEPc + slepc4py** for eigenvalue problems.
- **TAO** for optimization problems (part of PETSc).

- Recommended extras: BLAS/LAPACK (OpenBLAS, MKL), CMake, Git, compilers (gcc/gfortran or clang/flang).

Keep everything inside a virtual environment (`python -m venv .venv`) so upgrades do not disturb system Python.

Quick Workflow (All Platforms)

1. Install or load an MPI implementation.
2. Create and activate a Python virtual environment.
3. Upgrade `pip`, then install `mpi4py`.
4. Install PETSc/SLEPc (via package manager or source) and matching `petsc4py`/`slepc4py`.
5. Verify with short MPI and PETSc test programs.

The sections below give concrete commands for each operating system.

Installation

```
python -m venv .venv
source .venv/bin/activate
pip install -U pip
pip install mpi4py mpi4py-fft petsc petsc4py slepc slepc4py
```

Docker

Use Docker when you want a reproducible environment on any host with container support. You can run Docker on your laptop computer or in the cloud environment. You can convert your Docker "image" to Singularity/Apptainer for portability to run the HPC cluster.

Install Docker Engine or Docker Desktop first:

- Windows: https://docs.docker.com/desktop/install/windows-install/
- macOS: https://docs.docker.com/desktop/install/mac-install/
- Linux: https://docs.docker.com/engine/install/

```
# docker/Dockerfile
FROM ubuntu:24.04

ENV DEBIAN_FRONTEND=noninteractive \
    VENV_PATH=/opt/hpc-venv

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        python3 python3-venv python3-dev \
        build-essential gfortran cmake git \
        mpich libmpich-dev \
        libblas-dev liblapack-dev libopenblas-dev \
```

```
        petsc-dev slepc-dev && \
    python3 -m venv ${VENV_PATH} && \
    ${VENV_PATH}/bin/pip install --upgrade pip && \
    ${VENV_PATH}/bin/pip install mpi4py petsc4py slepc4py && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

ENV PATH="${VENV_PATH}/bin:${PATH}"
CMD ["bash"]
```

Build and launch the image from the repository root:

```
docker build -f docker/Dockerfile -t advanced-python-hpc .
docker run --rm -it --name hpc \
    --mount type=bind,source="$PWD",target=/workspace \
    advanced-python-hpc
```

Inside the container, your project appears at `/workspace`; the virtual environment is already active via `PATH`. MPI programs run with `mpiexec` as usual. Adjust `docker run` mount and port flags if you need to expose data directories or Jupyter notebooks.

Note: in the `--mount` flag, `source` points to the host directory (here your current working directory), while `target` is where that directory is mounted inside the container (`/workspace`).

Verification Commands

```
$ mpirun -n 2 python tests/test_mpi.py
Hello from rank 0 / 2
Hello from rank 1 / 2
```

```
$ mpirun -n 1 python tests/test_slepc.py
PETSc: (3, 24, 0)
SLEPc: 3.24.0
```

```
$ mpirun -n 1 python tests/test_tao.py
TAO: lmvm available
```

```
$ mpirun -n 4 python tests/test_petsc.py
Rank 0 sum = 8.0
Rank 1 sum = 8.0
Rank 2 sum = 8.0
Rank 3 sum = 8.0
```

Running simple MPI programs

Launching MPI Programs

Run scripts with `mpirun` or `mpiexec` to spin up multiple processes:

```
mpirun -n 4 python examples/01_mpi_hello_world.py
```

- Make sure the MPI runtime is on `PATH` and `LD_LIBRARY_PATH`.
- Activate the Python environment that has `mpi4py`.
- Reserve nodes and cores (scheduler directives) before launching on clusters.

Hands-On Flow

- Start with environment checks and practice launching scripts via `mpiexec` or `mpirun`.
- Watch for rank-tagged print statements to confirm parallel execution.

Writing `mpi4py` Programs

- Hello world: import `MPI`, grab `COMM_WORLD`, print rank, size, and hostname.
- Point-to-point: use `comm.send` and `comm.recv`
  - (Optional exercise) Switch to `comm.isend` and `comm.irecv` with `Wait` to overlap work.
- Collectives: `comm.bcast`, `comm.scatter`, `comm.gather`, `comm.reduce` map directly onto MPI theory.
- Keep data types consistent across ranks—prefer NumPy arrays or explicit dtype hints.

Hello World Walkthrough

- Explore `examples/01_mpi_hello_world.py` to map ranks to hosts.
- Print from both root and non-root processes to see the full communicator.
- There is no need to run `MPI_Init` and `MPI_Finalize` as in standard C/C++/Fortran MPI; `mpi4py` automatically handles them.

Send/Recv Patterns

- Step through `examples/02_mpi_send_recv.py` to show the handshake.
- Emphasize matching tags and datatypes for deterministic ordering.

Broadcast and Collectives

- Use `examples/03_mpi_bcast.py` to share data from root to every rank.
- Broadcast beats repeated point-to-point sends in performance.

Scatter and Gather

- Dive into `examples/04_mpi_scatter_gather.py` to partition work and collect results.
- Connect the pattern to chunking large arrays and aggregating statistics.

Debugging and Validation

- Prefix log messages with rank IDs to trace execution order.
- Drop `comm.Barrier()` between phases to expose ordering or race issues.
- Time sections with `MPI.Wtime()` to capture communication overhead.

MPI in Practice

- The MPI standard defines semantics, datatypes, and communication patterns; language bindings expose the calls.
- Implementations (MPICH, Open MPI, Intel MPI, MVAPICH2) follow the same spec, so the **same user code** can run with different MPI implementations on different parallel computers.
- MPI offers performance, scalability, and portability across mixed hardware generations.
- `mpi4py` maps MPI communicators and datatypes into Python objects, leaning on NumPy buffers or pickles instead of raw pointers.
- Exceptions propagate MPI errors in Python, giving friendlier feedback during development.

Parallel FFT

- FFT is widely used in scientific computing (spectral solvers, turbulence analysis, seismic imaging).
- Large-scale FFT computations are done in parallel with MPI.
- We use `mpi4py` to orchestrate data decomposition and local FFT kernels, connecting the workshop examples with production-ready Python toolchains.

Fast Fourier Transform Primer

The Fast Fourier Transform (FFT) is the workhorse algorithm that converts regularly sampled data from the time or spatial domain into its frequency components, doing the same job as a discrete Fourier transform but about $O(N \log N)$ fast instead of $O(N^2)$. That speedup turned spectral methods from a theoretical curiosity into a daily tool for scientists: we can isolate dominant wavelengths in turbulence snapshots, track seismic reflections buried in noisy sensor arrays, or reconstruct medical images from raw k-space samples within seconds.

For N complex sequence x_n, the discrete Fourier transform is

$$X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi i k n / N}, k = 0, ..., N-1.$$

The FFT is any algorithm that evaluates this sum for all k in $O(N \log N)$ operations (Cooley & Tukey, 1965).

FFT Applications

- **Turbulence & climate modeling (large data):** spectral solvers advance Navier–Stokes equations by hopping between velocity space and frequency space each timestep, and 3D grids at 2048^3 resolution only fit when the FFT is parallelized; MPI lets us decompose the grid and keep turnaround times compatible with daily forecast cycles.
- **Seismic & subsurface imaging (large data):** batched FFTs across thousands of geophones highlight stratigraphic boundaries that hint at oil, gas, or geothermal reservoirs; exploration crews expect overnight processing, so distributed FFT pipelines convert petabytes of field shots into actionable maps on schedule.

- **Biomedical imaging (MRI, CT, PET, ultrasound Doppler) (large data + latency pressure):** scanners record k-space or projection data in the frequency domain; fast inverse FFTs turn it into anatomy, while Doppler FFTs extract blood-flow velocities in real time—parallel transforms shave minutes off scans, enabling shorter breath-holds, reduced anesthesia time, and near-real-time interventional guidance.
- **Biomedical signals:** spectral decomposition exposes arrhythmias, seizure signatures, and sleep-stage bands, enabling clinicians to work directly in the frequency domain.
- **Spectroscopy & metabolomics:** FFTs transform time-domain free-induction decays and transient ion signals into molecular fingerprints.
- **Accelerator physics & plasma codes (large data):** FFT-based Poisson solvers update electric potentials across colossal meshes that span entire accelerator rings; high-throughput transforms keep pace with beam dynamics simulations tied to accelerator operations.
- **Signal processing & radio astronomy (large data):** correlating antenna feeds or filtering instrument noise uses FFTs to peel out astrophysical signals from static; instruments like the Square Kilometre Array ingest terabits per second, so parallel FFT stages are the only way to keep up with the data firehose.

Each example mirrors the cases we will explore: big multidimensional arrays, repeated forward/backward transforms, and the need to keep communications efficient as we scale to more ranks or GPUs.

References

- J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, 1965.
- M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, 2005.
- FFTW website: https://www.fftw.org/

Parallel strategy

- Data layout: slice a multidimensional array along one axis so each rank owns a contiguous block that fits in memory.
- Local work: execute `numpy.fft` (or `pyfftw.interfaces.numpy_fft`) on the assigned block; these kernels are already threaded and vectorized.
- Communication: exchange or gather transformed slabs so that boundary conditions or spectral post-processing can continue.
- Scaling step: for serious 3D FFTs, use pencil or slab decompositions plus all-to-all exchanges; libraries like `mpi4py-fft` and PETSc automate those patterns.

Sequential vs Parallel FFT

Sequential version (`examples/05_fft_sequential.py`)

The sequential example establishes a baseline by computing a 2D FFT on a single process with plain NumPy:

```
rows = 12
cols = 32

# set up data
x = np.linspace(0, 2 * np.pi, cols, endpoint=False)
base_signal = np.sin(3 * x) + 0.3 * np.sin(9 * x)
data = np.vstack([np.roll(base_signal, shift) for shift in range(rows)])

# compute FFT
fft_result = np.fft.fft(data, axis=1)
```

How it works:

1. Creates a 12×32 array where each row is a phase-shifted version of a composite sine wave (3x and 9x harmonics).
2. Applies `np.fft.fft` along axis=1 (columns), transforming each row independently into the frequency domain.
3. The result is a 12×32 complex array containing frequency coefficients.

This sequential approach is simple and efficient for small arrays but doesn't scale to large datasets that exceed single-node memory or require faster turnaround.

run: `python examples/05_fft_sequential.py`

Parallel Row-wise FFT (`examples/05_mpi_parallel_fft.py`)

The parallel example distributes the same workload across multiple MPI ranks using a **scatter-compute-gather** pattern:

```
# Rank 0 creates the data
if rank == root:
    x = np.linspace(0, 2 * np.pi, cols, endpoint=False)
    base_signal = np.sin(3 * x) + 0.3 * np.sin(9 * x)
    data = np.vstack([np.roll(base_signal, shift) for shift in range(rows)])
    chunks = np.array_split(data, size, axis=0)
else:
    chunks = None

# Scatter row blocks to all ranks
local_block = comm.scatter(chunks, root=root)

# Each rank computes FFT on its local rows
local_fft = np.fft.fft(local_block, axis=1)
```

```
# Gather results back to rank 0
gathered = comm.gather(local_fft, root=root)

# Rank 0 validates against sequential reference
if rank == root:
    fft_result = np.vstack(gathered)
    reference = np.fft.fft(data, axis=1)
    max_error = np.max(np.abs(fft_result - reference))
```

How it works: 1. **Data generation (rank 0):** Creates the same 12×32 signal array as the sequential version and splits it into `size` chunks along axis=0 (rows). With 4 ranks, each chunk contains 3 rows.

2. **Scatter:** `comm.scatter()` distributes one chunk to each rank. After scattering, rank 0 holds rows 0-2, rank 1 holds rows 3-5, rank 2 holds rows 6-8, and rank 3 holds rows 9-11.
3. **Local FFT:** Each rank independently computes `np.fft.fft(local_block, axis=1)`, transforming its assigned rows. Since FFT along axis=1 treats each row independently, no inter-rank communication is needed during this step.
4. **Gather:** `comm.gather()` collects all transformed blocks back to rank 0, which stacks them with `np.vstack()` to reconstruct the complete 12×32 result.
5. **Validation:** Rank 0 compares the parallel result against the sequential baseline to verify correctness (error should be near machine epsilon).

Key insight: Row-wise FFT is embarrassingly parallel when transforming along axis=1 because each row's transform is independent. Communication overhead is limited to the initial scatter and final gather.

Run: `mpiexec -n 4 python examples/05_mpi_parallel_fft.py`

NOTE: The scatter-compute-gather pattern is similar to Hadoop/Spark's map-reduce pattern. But they differ in implementation details and use cases.

MPI (scatter-compute-gather):

- Numerical simulations requiring low latency
- Problems where all ranks must synchronize frequently
- HPC clusters with fast interconnects
- Code needs fine-grained control over communication

MapReduce/Spark:

- Data analytics on petabyte datasets
- Embarrassingly parallel tasks with minimal inter-task communication
- Need fault tolerance for long-running jobs
- Prefer declarative APIs over manual message passing

Using mpi4py-fft

- The `mpi4py-fft` library provides high-level abstractions for parallel FFTs, handling domain decomposition, transposes, and communication automatically.
- This example demonstrates a true 2D FFT (transforming along both axes) rather than just row-wise transforms.

2D FFT (`examples/06_mpi4py_fft.py`)

```python
from mpi4py_fft import PFFT, newDistArray

global_shape = (12, 32)

# Plan a 2D complex FFT across all available ranks
fft = PFFT(comm, global_shape, axes=(0, 1), dtype=np.complex128)

# Create distributed arrays - each rank owns a portion of data
u = newDistArray(fft, forward_output=False)   # spatial domain
uh = newDistArray(fft, forward_output=True)   # frequency domain

# Fill local portion with reproducible random data
rng = np.random.default_rng(seed=13 + rank)
u[:] = rng.standard_normal(u.shape)
u_original = u.copy()

# Forward and backward transforms
fft.forward(u, uh)
fft.backward(uh, u)

# Validate reconstruction (accounting for unnormalized backward FFT)
total_points = np.prod(global_shape)
u /= total_points
local_error = np.max(np.abs(u - u_original))
global_error = comm.allreduce(local_error, op=MPI.MAX)
```

How it works:

1. **Planning:** `PFFT(comm, global_shape, axes=(0, 1))` creates an FFT plan for a 12×32 global array, transforming along both axes. The library automatically:
   - Decomposes the domain across available ranks (typically splitting along axis=0).
   - Plans the necessary local FFTs and all-to-all transposes.
   - Optimizes memory layouts and communication patterns.
2. **Distributed arrays:** `newDistArray()` allocates arrays that are partitioned across ranks. Each rank owns a contiguous slab:
   - With 4 ranks, rank 0 might own rows 0-2, rank 1 owns rows 3-5, etc.
   - The library tracks global vs. local indexing and manages ghost/halo regions automatically.

3. **Data initialization:** Each rank fills its local portion with random data seeded by `13 + rank`, ensuring reproducibility while keeping data decorrelated across ranks.

4. **Forward transform:** `fft.forward(u, uh)` performs a full 2D FFT:

   - First, local 1D FFTs along axis=1 (columns within each rank's slab).
   - Then, an MPI transpose redistributes data so each rank owns a different slab for axis=0 transforms.
   - Finally, local 1D FFTs along axis=0 complete the 2D transform.
   - The result `uh` is stored in a distributed frequency-domain array.

5. **Backward transform:** `fft.backward(uh, u)` reverses the process with inverse FFTs and transposes. Note that the backward FFT is unnormalized (standard FFTW convention), so we must divide by `total_points` to recover the original data.

6. **Validation:** Each rank computes its local max error, then `MPI.MAX` allreduce finds the global maximum. The error should be near machine epsilon ($\approx 10^{-15}$) for float64).

Key differences from manual parallelization:

Aspect	Manual (`05_mpi_parallel_fft.py`)	Library (`06_mpi4py_fft.py`)
Transform dimensionality	1D (row-wise only)	True 2D (both axes)
Decomposition	Explicit scatter/gather	Automatic domain partitioning
Communication	User manages	Hidden behind library API
Transpose handling	Not needed (1D transforms)	Automatic all-to-all exchanges
Scalability	Limited to embarrassingly parallel cases	Scales to 3D, pencil decompositions
Code complexity	~40 lines	~20 lines

When to use mpi4py-fft: - Multi-dimensional FFTs where transforms span multiple axes. - Large-scale 3D problems (turbulence, seismic imaging, cosmology) that require pencil decompositions. - Production workflows where you need battle-tested transpose algorithms and FFTW integration. - When you want to minimize communication code and focus on physics/algorithms.

Requirements: `python -m pip install mpi4py-fft` (FFTW libraries strongly recommended for performance).

Launch: `mpiexec -n 4 python examples/06_mpi4py_fft.py`

Notes

Use `mpi4py-fft` for turnkey pencil decompositions, transpose-heavy 3D plans, and collective-aware FFTW plans. Pair `mpi4py` with `pyFFTW` or MKL-accelerated NumPy when node-level performance is critical. PETSc (`petsc4py`) provides FFT-friendly distributed arrays (`DMDA`) that integrate with solvers and time integrators. Profile communication vs compute: large FFTs often become network-bound without topology-aware layouts.

MRI scaling takeaway: 128×128 phantoms fit on a laptop, but clinical slices exceed 1024×1024 and multi-coil stacks multiply the data volume. Parallel FFT shortens reconstruction latency. Streaming benefit: with MPI you can reconstruct subsets of k-space as they arrive, overlapping acquisition and reconstruction for near real-time imaging.

Discussion Prompts

Where do FFTs appear in your pipelines (spectral PDEs, filtering, convolution)? Which axis should you decompose first when memory grows faster than cores? How would you extend the row-wise pattern to a full 3D FFT with multiple transpose stages?

MRI Reconstruction with Parallel FFT

MRI overview

- Magnetic Resonance Imaging (MRI) reconstruction presents a canonical application of parallel FFT algorithms.
- Unlike conventional imaging where sensors directly capture spatial intensity, MRI scanners measure **k-space**—the 2D or 3D Fourier transform of the spin density.
- Reconstruction requires applying an inverse FFT to convert frequency-domain measurements into interpretable anatomical images.

K-space and the Fourier Encoding

- In MRI physics, spatial information is encoded through magnetic field gradients that create position-dependent resonance frequencies.
- The raw signal received by RF coils populates k-space, where:
- **k-space coordinates** (k_x, k_y, k_z) represent spatial frequencies, not physical positions.
- The **center of k-space** captures low spatial frequencies (coarse anatomy, contrast).
- The **periphery of k-space** captures high spatial frequencies (edges, fine detail).

Each k-space trajectory (typically line-by-line in Cartesian acquisition) is related to the image I(x,y) by the 2D Fourier transform:

$$S(k_x, k_y) = \int \int I(x, y) exp(-2\pi i(k_x \cdot x + k_y \cdot y))dxdy$$

Reconstruction inverts this relationship:

```
I(x,y) = IFFT2D[S(k_x, k_y)]
```

Computational Challenge

The computational burden scales with resolution and hardware complexity:

- **Clinical scans:** 256×256 complex-valued arrays (131K points per slice, multiple slices per volume).
- **Research protocols:** 512×512 or 1024×1024 grids (262K to 1M points).
- **Phased array coils:** 8–32 independent receiver channels, each requiring separate reconstruction and coil combination.
- **Real-time constraints:** Interventional imaging (MR-guided surgery, cardiac cine) demands reconstruction latencies under 50 ms.

Parallel FFT infrastructure directly addresses these demands: distributing k-space rows or employing slab/pencil decompositions reduces wall-clock time, enables higher throughput, and makes real-time applications feasible.

Test Data: Shepp-Logan Phantom

To demonstrate reconstruction algorithms without requiring real scanner data, we use the **Shepp-Logan phantom**—a standard test image in medical imaging since 1974. This synthetic phantom consists of overlapping ellipses with intensities representing different tissue types (gray matter, white matter, CSF, bone).

The phantom provides several advantages for algorithm development: - **Ground truth:** Known analytical form allows precise error quantification. - **Reproducibility:** Deterministic, no patient privacy concerns, consistent across studies. - **Controlled complexity:** Captures essential features (piecewise constant regions, smooth boundaries) without MRI physics complications.

Our examples use `_phantoms.shepp_logan()` to generate a 128×128 phantom, then simulate k-space acquisition via forward FFT. This lets us validate reconstruction correctness by comparing the reconstructed image against the original phantom.

Implementation Examples

Sequential Baseline (`examples/07_mri_fft_sequential.py`)

The sequential implementation establishes a reference reconstruction using NumPy's optimized FFT routines:

```
from _phantoms import shepp_logan

shape = (128, 128)
phantom = shepp_logan(shape)          # Generate test image (128×128 real-valued)
kspace = np.fft.fft2(phantom)         # Simulate k-space: forward FFT
recon = np.fft.ifft2(kspace)          # Reconstruct: inverse FFT

max_err = np.max(np.abs(recon.real - phantom))
```

Pipeline:

1. **Phantom generation:** `shepp_logan(shape)` returns a 128×128 array with intensity values $\in [0, 1]$ representing tissue contrast (CSF ≈ 0, bone ≈ 1, soft tissue intermediate).
2. **K-space simulation:** `np.fft.fft2(phantom)` computes the 2D DFT, yielding a 128×128 complex array. In real acquisitions, this k-space data comes directly from the scanner; here

we generate it synthetically to isolate reconstruction logic from acquisition physics.

3. **Reconstruction:** `np.fft.ifft2(kspace)` inverts the transform. Since the forward FFT was applied to real data, the result is Hermitian-symmetric; we extract `recon.real` for comparison.
4. **Validation:** The max absolute error should be O(ε_machine) $\approx 10^{-16}$ for float64, confirming that FFT $\rightarrow$ IFFT is numerically stable.

Observed output:

```
Phantom shape: (128, 128)
Sequential recon max error: 3.331e-16
```

This establishes the baseline for parallel correctness checks.

Launch: `python examples/07_mri_fft_sequential.py`

Parallel Row-Decomposition (`examples/07_mpi_fft_parallel.py`)

The parallel implementation uses **row-wise domain decomposition** to distribute the 2D IFFT across ranks. Since the 2D transform factors as a sequence of 1D transforms along each axis, we can parallelize stages that operate independently within row blocks:

```python
# Rank 0: Generate data and partition
if rank == root:
    phantom = shepp_logan(shape)
    kspace = np.fft.fft2(phantom)
    reference = np.fft.ifft2(kspace)  # Sequential baseline
    chunks = np.array_split(kspace, size, axis=0)
else:
    chunks = None

# Distribute k-space rows
local_kspace = comm.scatter(chunks, root=root)

# Stage 1: Independent column transforms (axis=1)
stage_one = np.fft.ifft(local_kspace, axis=1)

# Gather partial results
partials = comm.gather(stage_one, root=root)

# Stage 2: Complete row transforms (axis=0) on rank 0
if rank == root:
    assembled = np.vstack(partials)
    recon = np.fft.ifft(assembled, axis=0)
    max_err = np.max(np.abs(recon - reference))
```

Algorithm breakdown:

1. **Data partitioning (rank 0):** The 128×128 k-space is split along axis=0 into `size` contiguous row blocks. With 4 ranks, each receives a 32×128 chunk.

2. **Scatter phase:** MPI scatter distributes one chunk to each rank. Post-scatter, the k-space is decomposed as:
    - Rank 0: rows 0–31
    - Rank 1: rows 32–63
    - Rank 2: rows 64–95
    - Rank 3: rows 96–127
3. **Stage 1—Parallel column transforms:** Each rank applies 1D IFFT along axis=1 (across columns, within its local rows). Since each row's column transform is independent of other rows, this stage exhibits perfect parallelism with **zero inter-rank communication**.
4. **Gather phase:** Partial results are collected back on rank 0 and reassembled with `vstack`.
5. **Stage 2—Sequential row transform:** Rank 0 completes the 2D IFFT by applying 1D IFFT along axis=0 (across rows, down columns). This stage is sequential because the row transform couples all rows—data from all ranks is required.
6. **Validation:** Two error checks confirm correctness:
    - `|recon - reference|` (parallel vs. sequential) $\approx 10^{-16}$
    - `|recon.real - phantom|` (reconstructed vs. ground truth) $\approx 10^{-16}$

Performance characteristics:

The 2D IFFT factors as:

```
IFFT2D(S) = IFFT_axis0(IFFT_axis1(S))
```

For an N×N grid: - Stage 1 (axis=1): N × O(N log N) = O(N^2 log N) distributed across P ranks → O(N^2/P log N) per rank - Stage 2 (axis=0): N × O(N log N) = O(N^2 log N) sequential on rank 0

Work distribution: - Each rank handles 1/P of stage 1 (ideal speedup). - Stage 2 remains sequential (Amdahl bottleneck). - Communication: 2 collective operations (scatter + gather), O(N^2/P) data per rank.

For the 128×128 case with 4 ranks, approximately 75% of the FFT work is parallelized.

Observed output:

```
[Rank 1] Processed rows: 32
[Rank 2] Processed rows: 32
[Rank 3] Processed rows: 32
[Rank 0] Parallel MRI recon max error vs sequential: 5.204e-16
[Rank 0] Parallel MRI recon max error vs phantom: 5.204e-16
```

Limitations: - Stage 2 is sequential (limits scalability for large P). - All-to-all transpose would enable full parallelization but increases code complexity. - For production 3D FFTs, libraries like `mpi4py-fft` employ pencil decompositions that parallelize all stages.

Launch: `mpiexec -n 4 python examples/07_mri_fft_parallel.py`

Reconstruction Pipeline Summary

The complete MRI acquisition and reconstruction workflow:

Common initial steps: - Scanner acquires raw signal from tissue - K-space data populated (frequency domain): S(k_x, k_y) as complex 2D grid

Sequential reconstruction path: - Apply IFFT2D on single node - Output: I(x,y) reconstructed image

Parallel reconstruction path: - Partition k-space by rows across MPI ranks - Distribution: Rank 0, 1, 2, 3, etc. each holds a row subset - Stage 1 (parallel): - Each rank applies 1D IFFT along axis=1 (column transforms) - No inter-rank communication required - Communication phase: - Gather partial results to rank 0 - Stage 2 (sequential): - Rank 0 applies 1D IFFT along axis=0 (row transforms) - Output: I(x,y) reconstructed image

Clinical Impact and Performance

Throughput and Latency

Parallel reconstruction directly improves clinical workflow metrics:

Resolution scaling: - 256×256 clinical protocols: Reconstruction latency reduced from ~1 second (sequential) to ~200 ms (4-rank parallel). - 512×512 research protocols: Wall-clock time drops from ~5 seconds to ~1 second with 4-8 ranks. - 1024×1024 high-field imaging: Minutes → seconds, making high-resolution viable for routine use.

Multi-coil parallelism: Modern phased-array systems employ 8–32 receiver coils. Each coil requires independent reconstruction followed by optimal combination (e.g., sum-of-squares or SENSE). Parallel FFT enables: - Coil-level work distribution: Assign coils to rank groups. - Concurrent reconstruction: All coils processed simultaneously. - Aggregate speedup: 10–30× for 32-coil systems compared to sequential coil-by-coil processing.

Real-Time Applications

Interventional MRI (MR-guided surgery): - **Requirement:** Sub-200 ms latency for surgical navigation feedback. - **Challenge:** Continuous acquisition + reconstruction + display pipeline must operate at 5–10 fps. - **Solution:** Parallel FFT infrastructure + pipelined stages (while slice N reconstructs, slice N+1 acquires).

Cardiac cine imaging: - **Requirement:** Reconstruct cardiac phases within one R-R interval (~50 ms @ 60 bpm). - **Challenge:** Real-time imaging to visualize valve motion, wall dynamics during stress tests. - **Solution:** Distribute temporal frames across ranks; streaming reconstruction keeps pace with acquisition.

Streaming and Pipelining

Parallel infrastructure enables sophisticated pipelining strategies: - **Slice-level concurrency:** While ranks 0–3 reconstruct slice A, ranks 4–7 handle slice B. - **Temporal streaming:** Dynamic studies (perfusion, functional MRI) pipeline frames through reconstruction stages. - **Overlapped I/O:** Reconstruction begins as soon as k-space lines arrive from the scanner, rather than waiting for complete acquisition.

Scaling Beyond Row Decomposition

The two-stage scatter-gather approach demonstrated here is pedagogically clear but has limitations for large-scale production use:

Bottlenecks: - Stage 2 (row-wise IFFT) remains sequential, limiting strong scalability. - Rank 0 memory becomes a bottleneck when reassembling large datasets. - Gather/scatter communication patterns are less efficient than all-to-all transposes for balanced workloads.

Production alternatives: For 3D FFTs on 512^3 grids or higher-dimensional problems (4D time series, 5D diffusion), production libraries employ: - **Pencil decompositions:** Data partitioned along multiple axes, enabling parallel transforms in all dimensions. - **All-to-all transposes:** MPI `Alltoall` operations redistribute data between transform stages with optimal communication patterns. - **Hybrid parallelism:** Combine MPI (inter-node) with OpenMP/CUDA (intra-node) for hierarchical machines.

Libraries like `mpi4py-fft` automate these strategies.

PETSc

What is PETSc

The **Portable, Extensible Toolkit for Scientific Computation (PETSc)** is a comprehensive software library for the parallel numerical solution of partial differential equations (PDEs) and related problems in scientific computing. Developed at Argonne National Laboratory, PETSc provides a unified framework that abstracts away the complexities of distributed-memory parallelism while exposing high-performance numerical algorithms.

From MPI to PETSc: Why the Abstraction Matters

With plain MPI, implementing a parallel PDE solver requires manually managing:

1. Domain decomposition and ghost cells:

```
# Plain MPI: Manual halo exchange for a 2D Laplacian stencil
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# Determine local subdomain boundaries
local_nx, local_ny = compute_local_size(rank, size)
ghost_cells = allocate_with_halo(local_nx, local_ny, halo_width=1)

# Exchange boundary data with 4 neighbors (N, S, E, W)
send_north = ghost_cells[1, :]
send_south = ghost_cells[-2, :]
comm.Sendrecv(send_north, dest=rank_north, recvbuf=recv_south, source=rank_south)
comm.Sendrecv(send_south, dest=rank_south, recvbuf=recv_north, source=rank_north)
# ... repeat for east/west boundaries ...
```

With PETSc DMDA (Distributed Multi-dimensional Array):

```python
# PETSc: Automatic halo exchange
from petsc4py import PETSc

da = PETSc.DMDA().create(dim=2, sizes=(nx, ny), stencil_width=1, comm=comm)
da.setFromOptions()

# One-line halo exchange:
da.globalToLocal(global_vec, local_vec)  # All neighbor communication handled internally
```

2. Sparse matrix assembly with correct global indexing:

```python
# Plain MPI: Track global indices, communicate off-processor entries
local_rows = rows_for_rank(rank, size)
A_local = scipy.sparse.lil_matrix((len(local_rows), global_N))

for i in local_rows:
    # Compute stencil entries for row i
    for j in stencil_neighbors(i):
        A_local[local_to_global(i), j] = compute_coefficient(i, j)

# Convert to parallel format, communicate off-processor data...
# (Requires custom CSR distribution + MPI_Alltoallv communication)
```

With PETSc Mat:

```python
# PETSc: Distributed sparse matrix with automatic communication
A = PETSc.Mat().createAIJ([global_N, global_N], comm=comm)
A.setFromOptions()
A.setUp()

istart, iend = A.getOwnershipRange()  # My row range
for i in range(istart, iend):
    # Set values; PETSc handles off-processor communication
    A.setValues(i, neighbor_indices, neighbor_coefficients)

A.assemblyBegin()  # Start communication
A.assemblyEnd()    # Complete assembly (MPI under the hood)
```

3. Iterative solver implementation:

```python
# Plain MPI: Implement CG from scratch with parallel dot products
r = b - A @ x
p = r.copy()
for iteration in range(max_iter):
    alpha = comm.allreduce(np.dot(r, r), op=MPI.SUM) / comm.allreduce(np.dot(p, A @ p), op=
    x += alpha * p
    r_new = r - alpha * (A @ p)
    beta = comm.allreduce(np.dot(r_new, r_new), op=MPI.SUM) / comm.allreduce(np.dot(r, r),
    p = r_new + beta * p
    r = r_new
```

```
    # Missing: preconditioning, convergence checks, restarts...
```

With PETSc KSP:

```
# PETSc: Production solver with one setup call
ksp = PETSc.KSP().create(comm=comm)
ksp.setOperators(A)
ksp.setType('cg')                          # Or gmres, bicgstab, etc.
ksp.getPC().setType('gamg')                # Algebraic multigrid preconditioner
ksp.setFromOptions()                       # Override via -ksp_type, -pc_type at runtime

ksp.solve(b, x)                            # Solves Ax = b in parallel
```

Key Abstractions and Their Benefits

PETSc Component	Abstraction	Plain MPI Equivalent	Complexity Reduction
Vec	Distributed vector with ghost cells	Manual partitioning + halo exchange code	~100 lines → 5 lines
Mat	Distributed sparse matrix	Custom CSR distribution + communication	~200 lines → 10 lines
DM (DMDA/DMPlex)	Structured/unstructured grid topology	Manual neighbor tracking, ghost regions	~300 lines → 15 lines
KSP	Krylov subspace solvers + preconditioners	CG/GMRES from scratch + parallel dot products	~500 lines → 20 lines
SNES	Nonlinear solvers (Newton, quasi-Newton)	Custom Newton with line search, Jacobian assembly	~800 lines → 30 lines
TAO	Optimization (bound-constrained, PDE-constrained)	Custom gradient descent/L-BFGS + parallel reductions	~1000 lines → 40 lines

Algorithmic Richness

PETSc ships with production-quality implementations of algorithms that would take months to develop from MPI primitives:

Linear solvers (KSP):

- Direct: LU, Cholesky (via external packages: MUMPS, SuperLU_DIST)
- Krylov: CG, GMRES, BiCGStab, MINRES, TFQMR, Richardson
- Specialized: QMR for complex symmetric, LGMRES with restart

Preconditioners (PC):

- Algebraic multigrid: GAMG (PETSc-native), BoomerAMG (HYPRE)

- Domain decomposition: Block Jacobi, additive/multiplicative Schwarz
- Incomplete factorizations: ILU(k), ICC(k)
- Physics-based: Fieldsplit (segregated fluid-structure, Stokes saddle point)

Nonlinear solvers (SNES):

- Newton methods with line search or trust region
- Quasi-Newton: Broyden, L-BFGS
- Nonlinear CG, Anderson mixing (for fixed-point problems)

Optimization (TAO):

- Unconstrained: Newton, L-BFGS, conjugate gradient
- Bound-constrained: BLMVM, TRON (trust region Newton)
- PDE-constrained: Reduced-space methods, adjoint-driven optimization

Time integration (TS):

- Explicit: Forward Euler, Runge-Kutta (2-5 stages)
- Implicit: Backward Euler, BDF, Theta methods
- IMEX: Implicit-explicit splittings for stiff-nonstiff systems

Runtime Configurability

Unlike custom MPI code where algorithm choices are hard-coded, PETSc allows runtime tuning without recompilation:

```
# Try different solver combinations from command line:
mpiexec -n 16 python my_solver.py -ksp_type gmres -pc_type gamg
mpiexec -n 16 python my_solver.py -ksp_type bcgs -pc_type ilu
mpiexec -n 16 python my_solver.py -ksp_type preonly -pc_type lu -pc_factor_mat_solver_type

# Monitor convergence:
mpiexec -n 16 python my_solver.py -ksp_monitor -ksp_view

# Profile performance:
mpiexec -n 16 python my_solver.py -log_view
```

This design philosophy—compose algorithms from command-line options—accelerates experimentation and makes production codes adaptable to different hardware (CPUs, GPUs) and problem scales without source changes.

The Complete Scientific Computing Stack

PETSc provides an **end-to-end HPC toolkit** that covers the full spectrum from problem setup to solution:

Data structures and topology:

- `Vec` / `Mat`: Distributed linear algebra primitives
- `DM` (DMDA for structured grids, DMPlex for unstructured meshes): Automatic domain decomposition, ghost cell management, and global-to-local mappings

Solvers and algorithms:

- `KSP`: 40+ iterative linear solvers with 30+ preconditioners
- `SNES`: Nonlinear equation systems (Newton, quasi-Newton, nonlinear CG)
- `TS`: Time integration for ODEs and time-dependent PDEs
- `TAO`: Optimization (unconstrained, bound-constrained, PDE-constrained)
- `SLEPc` (extension): Eigenvalue problems for modal analysis

Scalability:

The MPI-native design means codes written with `petsc4py` scale seamlessly from **laptop prototyping** (single process) to **leadership-class supercomputers** (100,000+ cores) without modification. The same Python script that solves a 1000-element problem on your workstation can tackle 100-million-element problems on a production cluster by simply changing `mpiexec -n`.

Real-world impact:

- **Climate models:** DMDA manages structured atmospheric grids; KSP solves pressure Poisson equations.
- **Subsurface flow & geomechanics:** DMPlex handles unstructured tetrahedral meshes for reservoir simulation.
- **Cardiac electrophysiology:** DMPlex + SNES for reaction-diffusion systems on anatomical heart geometries.
- **Fusion plasma (MHD):** SNES solves coupled magnetohydrodynamic equations in tokamak simulations.
- **Aircraft structural analysis:** SLEPc computes flutter modes and eigenfrequencies.
- **Medical imaging:** TAO performs MRI parameter estimation and image reconstruction.
- **Power grid optimization:** TAO handles optimal power flow with tens of thousands of constraints.

Quick Start

Install MPI, PETSc, SLEPc, and their Python bindings. Initialize once: `from petsc4py import PETSc`; `PETSc.Sys.Initialize()` (auto via import in new versions). Core layers to remember: `Vec`/`Mat`, `KSP`, `SNES`, `TAO`, `DM`.

Core Patterns

Assemble distributed matrices and vectors collaboratively (e.g., structural stiffness). Select solvers and preconditioners per workload (AMG for groundwater, block PC for Navier–Stokes). Tune behavior with runtime flags (`-ksp_type`, `-pc_type`, `-snes_type`, `-tao_type`), no recompiles. `petsc4py` mirrors the C API while speaking NumPy buffers and `mpi4py` communicators. Hands-on: `examples/08_petsc_tao_t2_fit.py` uses TAO to recover MRI T2 decay parameters.

Practical Guidance

Prototype small meshes before full-scale (coarse cardiac grid to full anatomy). Log convergence and timings every run. Swap solver pipelines via options without recompiling.

Discussion

What domain decompositions fit your models (DMDA vs DMPlex)? What are your favorite PETSc examples or docs to bookmark? When do you promote from `KSP` to `SNES` or `TAO` (MRI inverse problems, coupled climate)?

Architecture Overview

`Vec` and `Mat`: distributed linear algebra for CFD, seismic inversion. `KSP`: Krylov methods with preconditioners (GAMG, block PC) for combustion, ocean flow. `SNES`: nonlinear engines for phase-field materials, biomechanics elasticity. `SLEPc`: eigenvalues for aircraft modes, photonic bands. `TAO`: optimization for medical imaging, climate assimilation, power grids. `DM`: DMDA (structured) and DMPlex (unstructured) handle grids; MPI backbone manages halo exchange.

Parameter Estimation using PETSc/Tao

Problem Overview

This case study demonstrates how PETSc's **Toolkit for Advanced Optimization (TAO)** solves a canonical parameter estimation problem: **fitting exponential decay curves to noisy measurements**. While our motivating application is MRI tissue characterization (T_2 mapping), the mathematical framework applies broadly to:

- **Pharmacokinetics:** Drug concentration decay in blood plasma.
- **Nuclear physics:** Radioactive decay rate determination.
- **Chemical kinetics:** Reaction rate constant estimation.
- **Geophysics:** Electrical conductivity relaxation in subsurface materials.
- **Battery science:** Charge-discharge capacity fade modeling.

The computational challenge: **millions of independent small-scale nonlinear least-squares problems** that must be solved efficiently. TAO's parallel optimization infrastructure addresses this need.

The Physical Process: MRI T_2 Relaxation

In magnetic resonance imaging, hydrogen nuclei (protons) in tissue are first **excited** by a radiofrequency pulse, causing their magnetic moments to precess coherently. After excitation stops, this coherence gradually **decays** due to local field inhomogeneities—a process called **transverse relaxation**.

What is T_2?

T_2 (pronounced "T-two") is a **tissue-specific time constant** that characterizes how quickly the MRI signal decays after excitation. Physically, it measures how long proton spins remain synchronized:

- **Large T_2 values** (100–300 ms): Spins stay coherent longer, producing sustained signal. This occurs in tissues with free water molecules (edema, CSF, cysts) where protons experience relatively uniform local magnetic environments.

- **Small T_2 values** (10–50 ms): Spins rapidly lose synchronization, causing fast signal decay. This occurs in structured tissues (bone, tendons, fibrotic regions) where protons are tightly bound and experience heterogeneous magnetic fields.

Mathematically, T_2 is the time constant in the exponential decay function: after time T_2, the signal intensity falls to approximately 37% (1/e) of its initial value. After $3\times T_2$, only ~5% of the original signal remains.

Why T_2 varies across tissues: The local magnetic environment surrounding each proton depends on molecular mobility, chemical composition, and tissue microstructure. Different tissues have characteristic T_2 values:

Tissue Type	Typical T_2 (ms)	Interpretation
Bone, calcification	1–10	Very fast decay (tightly bound protons)
Muscle	40–60	Moderate decay
Gray matter (brain)	80–100	Slower decay
Edema, inflammation	150–300	Very slow decay (excess free water)

Clinical value: T_2 mapping converts qualitative MRI "brightness" into quantitative tissue properties, enabling objective diagnosis of cartilage degeneration, tumor characterization, and treatment response monitoring.

Acquisition: The scanner measures signal intensity at multiple **echo times** (time delays after excitation), producing a decay curve at each spatial location (voxel). Our task: **invert** these measurements to recover the underlying decay parameters.

Mathematical Model

The observed MRI signal at a single voxel follows an exponential decay:

$$S(t) = A \exp\left(-\frac{t}{T_2}\right) + \varepsilon(t)$$

where: - $\mathbf{S(t)}$: Measured signal intensity at echo time t (observable). - $\mathbf{A}$: Initial signal amplitude, proportional to proton density (unknown parameter). - T_2: Transverse relaxation time constant in milliseconds (unknown parameter, our primary target). - $\boldsymbol{\varepsilon(t)}$: Measurement noise from thermal fluctuations, system imperfections, patient motion.

Given a set of measurements $\{(t_i, S_i)\}_{i=1}^N$ acquired at echo times $t_1, t_2, \ldots, t_N$ (typically 8–15 samples), we seek the parameters $\theta = (A, T_2)$ that best explain the data.

The Inverse Problem: Least-Squares Estimation

Formulation

We cast parameter estimation as a **nonlinear least-squares problem**: find $\theta = (A, T_2)$ that minimizes the sum of squared residuals:

$$f(A, T_2) = \frac{1}{2} \sum_{i=1}^{N} \left[A \exp\left(-\frac{t_i}{T_2}\right) - S_i \right]^2$$

Intuition: For any candidate parameter pair (A, T_2), we: 1. **Predict** signal values at each echo time: $\hat{S}_i = A \exp(-t_i/T_2)$. 2. **Compute residuals**: $r_i = \hat{S}_i - S_i$ (prediction error). 3. **Penalize** via squared norm: $\|r\|^2 = \sum r_i^2$.

The optimizer iteratively adjusts (A, T_2) to minimize this penalty, converging to the **maximum likelihood estimate** under Gaussian noise assumptions.

Why This is Challenging

Nonlinearity: The exponential term couples A and T_2 in a nonlinear fashion. Unlike linear least squares (where parameters appear linearly), we cannot solve this in closed form via matrix inversion. The problem requires **iterative** optimization with gradient-based methods.

Parameter scaling: A and T_2 live on different scales (A $\approx$ 0.1–2.0, $T_2 \approx$ 10–300 ms). Naive gradient descent often struggles without proper preconditioning or quasi-Newton methods.

Scale: A 256×256×128 MRI volume contains ~8 million voxels. Even with fast convergence (5–10 iterations per voxel), sequential processing is prohibitively slow. **Parallel execution** across voxels is essential.

Gradient Computation

Gradient-based optimizers (L-BFGS, TRON, etc.) require the derivative of f with respect to each parameter:

$$\frac{\partial f}{\partial A} = \sum_{i=1}^{N} r_i \exp\left(-\frac{t_i}{T_2}\right)$$

$$\frac{\partial f}{\partial T_2} = \sum_{i=1}^{N} r_i \cdot A \exp\left(-\frac{t_i}{T_2}\right) \cdot \frac{t_i}{T_2^2}$$

where $r_i = A \exp(-t_i/T_2) - S_i$ is the residual at echo time t_i.

Why provide gradients? While TAO can approximate gradients via finite differences, **analytical gradients** are: - **More accurate:** No truncation error from ε-perturbations. - **Faster:** One function evaluation instead of 2N (for N parameters). - **Numerically stable:** Avoids catastrophic cancellation in difference quotients.

PETSc TAO Implementation

Algorithm Selection

PETSc TAO provides multiple optimization algorithms. For our bounded, smooth, nonlinear least-squares problem, we use **LMVM** (Limited-Memory Variable Metric), a quasi-Newton method that:

- **Approximates the Hessian** using gradient history (L-BFGS approach).

- **Requires only first derivatives** (no need to code second derivatives).
- **Handles thousands of parameters efficiently** (though we have only 2 per voxel).
- **Supports bound constraints** (e.g., (T_2 > 0), (A > 0)).

Alternative TAO solvers include: - `blmvm`: Bound-constrained L-BFGS (enforces (T_2)). - `tron`: Trust-region Newton (faster for well-conditioned problems). - `nls`: Nonlinear least-squares specialized (explicitly exploits residual structure).

Code Structure

The implementation (`examples/08_petsc_tao_t2_fit.py`) consists of three components:

1. Objective and gradient callback (T2Objective class):

```python
class T2Objective:
    """Least-squares objective for T2 decay fitting."""

    def __init__(self, times: np.ndarray, signal: np.ndarray):
        self.times = times      # Echo times [t1, t2, ..., tN]
        self.signal = signal    # Measured intensities [S1, S2, ..., SN]

    def __call__(self, tao: PETSc.TAO, x: PETSc.Vec, g: PETSc.Vec) -> float:
        # Extract parameters from distributed Vec
        vs, seq = PETSc.Scatter.toAll(x)
        vs.scatter(x, seq, PETSc.InsertMode.INSERT, PETSc.ScatterMode.FORWARD)
        params = seq.getArray(readonly=True).copy()

        amp, t2 = params
        t2 = max(t2, 1e-6)  # Numerical safeguard

        # Forward model: predict signal at each echo time
        exp_term = np.exp(-self.times / t2)
        pred = amp * exp_term
        residual = pred - self.signal

        # Objective: f = 0.5 * ||residual||^2
        f = 0.5 * np.dot(residual, residual)

        # Gradient: df/dA, df/dT2
        full_grad = np.empty_like(params)
        full_grad[0] = np.dot(residual, exp_term)
        full_grad[1] = np.dot(residual, amp * exp_term * self.times / (t2 * t2))

        # Write gradient into PETSc Vec
        start, end = x.getOwnershipRange()
        g.getArray()[:] = full_grad[start:end]

        return f
```

Key design choice:

The callback computes **both** objective and gradient in one call, avoiding redundant exponential evaluations.

2. Synthetic data generation:

```
# Echo times (ms) - typical clinical sequence
times_ms = np.array([10, 20, 40, 60, 80, 120, 160, 200, 240, 280])

# Ground truth parameters
true_amp = 1.0
true_t2 = 85.0  # Typical gray matter value

# Generate noisy measurements
signal = true_amp * np.exp(-times_ms / true_t2) + noise
```

3. TAO solver setup and execution:

```
# Create distributed parameter vector (2 parameters)
x = PETSc.Vec().createMPI(2, comm=comm)
x.setValues(range(2), [0.8, 50.0])  # Initial guess
x.assemble()

# Configure TAO optimizer
tao = PETSc.TAO().create(comm=comm)
tao.setType("lmvm")                          # Limited-memory quasi-Newton
tao.setObjectiveGradient(objective, None)    # Register callback
tao.setFromOptions()                         # Allow runtime override
tao.solve(x)                                 # Run optimization

# Extract solution
solution = retrieve_global_solution(x)
amp_est, t2_est = solution
```

Runtime and Results

Launch the example:

```
mpiexec -n 2 python examples/08_petsc_tao_t2_fit.py
```

Expected output:

```
--- PETSc TAO MRI T2 Fit ---
True parameters: amplitude = 1.000, T2 = 85.0 ms
Estimated parameters: amplitude = 0.998, T2 = 84.6 ms
Final objective value: 1.234e-03
```

Convergence analysis:

- Typical iteration count: 8–12 iterations for convergence.
- Final objective ~10 ^{3}: consistent with noise variance ($^{2} \approx 0.02^2 \times 10$ points).

- Parameter errors < 1%: demonstrates robust estimation despite 2% noise.

Why PETSc + Python for This Problem?

Performance: PETSc's C/MPI core handles:

- Line search algorithms (backtracking, quadratic interpolation).
- Convergence monitoring (gradient norm, function decrease).
- Parallel vector operations (dot products, norms) via MPI collectives.

Productivity: Python enables:

- Concise problem specification (decay model in ~10 lines).
- NumPy integration for vectorized exponential/residual computation.
- Rapid prototyping (test with synthetic data before real MRI I/O).

Scalability: The same code scales from:

- **Development (1 rank):** Debug on synthetic 1-voxel problems.
- **Validation (4 ranks):** Test on small 64×64 regions.
- **Production (1000+ ranks):** Process 512×512×300 whole-brain volumes in parallel, distributing voxels across nodes.

Workflow Summary

Data acquisition and preprocessing:

- MRI scanner acquires raw k-space data
- Reconstruct images using inverse FFT (see 36-mri.md)
- Extract signal decay curves at each voxel location

Parallel parameter estimation (distributed across MPI ranks):

- For each voxel independently:
  - Input data:
    - Echo times: t1, t2, …, tN (typically 10 samples)
    - Measured signals: S1, S2, …, SN
  - Initial parameter guess: A0 = 0.8, T2_0 = 50 ms
  - Optimization setup:
    - Define objective function: f(A, T2) = sum of squared residuals
    - Compute analytical gradient: grad_f(A, T2)
  - TAO LMVM solver execution:
    - Iteratively refine parameters (8-12 iterations)
    - Terminate when gradient norm < tolerance
  - Output: Converged parameters A, *T2* for this voxel

Post-processing:

- Gather results from all ranks
- Assemble complete T2 map (e.g., 256x256x128 volume)
- Clinical interpretation:
  - Identify tissue abnormalities (edema, tumors, degeneration)

- Quantify treatment response over time
- Generate diagnostic reports for radiologists

Extending the Example

1. Add Bound Constraints

Real parameters have physical limits: A > 0, $T_2 > 0$. Enforce these with TAO's built-in bound constraints:

```
# Set lower bounds (no upper bounds)
lower = PETSc.Vec().createMPI(2, comm=comm)
lower.setValues([0, 1], [1e-6, 1e-3])  # A <=10 , T_2 <= 1 ms
lower.assemble()

tao.setVariableBounds(lower, None)
tao.setType("blmvm")  # Bounded L-BFGS
```

2. Multi-Exponential Models

Tissues with multiple compartments (intracellular/extracellular water) exhibit multi-exponential decay:

$$S(t) = A_1 \exp(-t/T_{2,1}) + A_2 \exp(-t/T_{2,2}) + \varepsilon(t)$$

Extend the parameter vector to 4D: ((A_1, T_{2,1}, A_2, T_{2,2})), update the objective/gradient, and TAO handles the increased dimensionality automatically.

3. Spatial Regularization

Anatomical structures have smooth T_2 variations. Add a spatial penalty to couple neighboring voxels:

[f_{reg}( ) = *{voxels}* f{data}(*v) + {edges}* | _u - _v|^2]

This requires switching to PETSc **SNES** (nonlinear solvers with PDE coupling) or TAO's composite formulations with distributed Hessian approximations.

4. Real MRI Data Integration

Replace synthetic data with actual scanner outputs:

```
import nibabel as nib  # NIfTI reader

# Load 4D time-series (X × Y × Z × Nechoes)
img = nib.load('multi_echo_scan.nii.gz')
data = img.get_fdata()

# Distribute voxels across ranks
local_voxels = partition_voxels(data.shape[:3], comm)
```

```
# Fit each voxel's decay curve in parallel
for vox in local_voxels:
    signal = data[vox[0], vox[1], vox[2], :]
    objective = T2Objective(echo_times, signal)
    # ... TAO solve ...
```

With MPI-parallel I/O (HDF5, MPI-IO), ranks read/write their assigned voxels directly, eliminating root-node bottlenecks.

Discussion Questions

- **Algorithmic choice:** When would you prefer `tron` (trust region) over `lmvm` (quasi-Newton)?
- **Jacobian sparsity:** For multi-voxel coupled problems, how would you exploit sparsity in the Hessian?
- **Uncertainty quantification:** Can TAO provide parameter confidence intervals (bootstrap, Hessian-based covariance)?
- **GPU acceleration:** PETSc supports CUDA/HIP backends—how would you adapt the callback for GPU execution?

Vibrating Bridge Mode (Eigenvalue) using PETsc/SLEPc

Eigenvalue Problems

The eigenvalue problem seeks non-zero (x) and scalar ( ) satisfying

$$Ax = \lambda x$$

where ( ) is an eigenvalue (scaling factor) and (x) the eigenvector (direction). In structural dynamics, eigenvalues and eigenvectors predict vibration modes. For bridge dynamics, ( ) gives the squared vibration frequency while (x) describes the deflection shape for that mode. The vector (x) carries one entry per degree of freedom, so its length equals the number of mesh nodes or DOFs. Realistic meshes often exceed millions of unknowns, requiring eigenproblems to be distributed across MPI ranks. The lowest eigenpair approximates the fundamental frequency and shape, which matters for safety analysis of bridges, towers, and aircraft. PETSc with SLEPc provides scalable eigensolvers, accessible in Python through `petsc4py` and `slepc4py`.

Problem Setup

We model the bridge deck as a rectangular plate with fixed edges. Finite differences discretize the Laplacian, assembling a stiffness matrix (K), while the cell areas fill a lumped mass matrix (M). This yields the generalized eigenproblem

$$K\,x = \lambda M\,x, \quad f = \frac{\sqrt{\lambda}}{2\pi}.$$

Here (K) and (M) form a generalized mass–spring system where ( ) corresponds to squared angular frequency (^2); taking $\sqrt{\lambda}$ recovers ω. Converting angular frequency to Hertz via $f = \omega/(2\pi)$ gives $\sqrt{\lambda}/(2\pi)$ as an approximation of the bridge's vibration frequency.

The script `examples/09_petsc_eigen_bridge.py` follows these steps: 1. Partition interior grid points across MPI ranks. 2. Assemble `Mat` objects for (K) (5-point stencil) and (B) (mass). 3. Configure SLEPc `EPS` to request the smallest real eigenpair. 4. Gather the eigenvector, reshape back to the 2D grid, and print a cross-section of the mode shape.

Key SLEPc Steps

```
from petsc4py import PETSc
from slepc4py import SLEPc

A = PETSc.Mat().createAIJ(size=n, nnz=5)      # stiffness
B = PETSc.Mat().createAIJ(size=n, nnz=5)      # mass
# ... assemble finite-difference Laplacian with boundary conditions ...

eps = SLEPc.EPS().create()
eps.setOperators(A, B)
eps.setProblemType(SLEPc.EPS.ProblemType.GHEP)
eps.setWhichEigenpairs(SLEPc.EPS.Which.SMALLEST_REAL)
eps.solve()

nev = eps.getConverged()
vr, vi = A.createVecs()
eps.getEigenpair(0, vr, vi)
```

The eigenvector `vr` can be reshaped into the 2D grid to visualize the bridge deck deflection. Larger grids scale across MPI ranks automatically.

Requirements

You need PETSc compiled with SLEPc support (`--with-slepc=1` or via package managers that bundle SLEPc). Install `slepc4py` alongside `petsc4py`. Run the demo with `mpirun -n 4 python examples/09_petsc_eigen_bridge.py`. Optionally, add `matplotlib` or `pyvista` to visualize the mode surface.

Installation Notes

For setup steps, see **25-environment.md** for combined MPI/PETSc/SLEPc installation instructions, platform specifics, and verification commands. Use the same PETSc/SLEPc build across all nodes (or rely on cluster modules) to avoid mismatched libraries.

Extending the Example

Request multiple eigenpairs with `eps.setDimensions(nevs=5)`. Explore damping via quadratic eigenvalue problems. Swap finite differences for PETSc DMPlex finite-element assembly.

Solving nonlinear system of equations with PETSc

Background

- Lithium-ion battery packs power laptops, phones, grid storage, and electric vehicles, making safety and reliability important.
- A **cell** is the smallest electrochemical unit: anode, cathode, separator, and electrolyte sealed in a can or pouch delivering a few volts.
- A battery pack groups hundreds to thousands of cells, their electrical connections, thermal pathways, and battery-management electronics into one enclosure.
- Lithium-ion chemistry stores energy by shuttling lithium ions between electrode materials.
- Malfunctioning cells can trigger exothermic reactions.
- When those reactions heat the pack faster than thermal systems can remove energy, **thermal runaway** begins: each temperature rise accelerates more reactions, releasing even more heat, potentially leading to venting or fire.
- Predicting the onset requires solving a temperature field coupled with reaction kinetics, involving nonlinear terms that grow exponentially with temperature. Real packs involve thousands of cells and detailed geometries, leading to millions of unknowns and requiring MPI-parallel nonlinear solvers.

Governing Equations

The heat balance with Arrhenius source (Bratu-type model) is

$$-\nabla \cdot (k \nabla T) = Q_0 \exp\left(\frac{E_a}{R(T + T_{\text{ref}})}\right)$$

where (T) is temperature rise over ambient, (k) is effective thermal conductivity, (Q_0) is reaction rate prefactor, (E_a) activation energy, and (R) gas constant. Boundary conditions include Dirichlet on housing and insulated symmetry planes. Discretization (finite differences or finite elements) leads to a nonlinear residual

$$F_i(T) = -k \sum_{j \in \mathcal{N}(i)} w_{ij}(T_j - T_i) - Q_0 \exp\left(\frac{E_a}{R(T_i + T_{\text{ref}})}\right).$$

This system is strongly nonlinear (exponential source) and globally coupled. A Bratu problem is the canonical PDE (- u - e^{u} = 0), capturing diffusion balanced against exponential heat release, mirroring thermal runaway physics.

PETSc SNES

- PETSc's **SNES** (Scalable Nonlinear Equations Solvers) provides Newton, line-search, and trust-region methods with flexible Jacobians and preconditioners.
- DMDA (distributed arrays) manages structured grids across MPI ranks, while DMPlex extends to unstructured packs. Combining SNES with scalable Krylov solvers (KSP) and algebraic multigrid preconditioners (PC) handles more than 10^7 unknowns, fitting real packs.

petsc4py Implementation Outline

The script `examples/10_petsc_snes_bratu.py` (Bratu as thermal runaway analogue) demonstrates how to: 1. Create a 2D DMDA grid distributed over MPI ranks. 2. Define the nonlinear residual (F(T)) and its Jacobian for the exponential source. 3. Configure SNES (Newton with line search) and solve for the steady-state temperature field. 4. Monitor convergence and gather the solution for analysis or visualization.

The script works in dimensionless variables (Bratu form (- u - e^{u} = 0)) but the scaffolding matches full thermo-chemical models.

Key PETSc Calls

```
from petsc4py import PETSc

dm = PETSc.DMDA().create([nx, ny], dof=1, stencil_width=1)
snes = PETSc.SNES().create()
snes.setDM(dm)
snes.setFunction(residual, None)
snes.setJacobian(jacobian, None)
snes.setFromOptions()
snes.solve(None, dm.createGlobalVec())
```

The residual and Jacobian callbacks leverage DMDA local vectors to access stencil values. Customize runtime with options like `-snes_monitor -ksp_type cg -pc_type gamg`.

Scaling to Real Systems

Increase DMDA dimensions (3D pack) or switch to DMPlex for CAD-derived meshes. Couple thermal field to electrochemical state equations within SNES or via PETSc's multiphysics interfaces. Deploy on clusters with `mpirun -n 256 python examples/10_petsc_snes_bratu.py` to match realistic resolution.

Stabilizing a power grid with PETSc

Background

- Modern towns and cities rely on electrical grids. When a feeder line fails or demand spikes, operators must reroute power.
- Grid engineers solve large sparse systems that model how voltages distribute across the network.
- As grids grow to millions of nodes (smart meters, EV chargers, rooftop solar), analyzing the network in real time requires MPI-scale solvers.

Network Model

- We treat the grid as a graph: nodes represent buses or substations, edges represent transmission lines with conductance (g_{ij}).

- The unknowns are the voltage (electrical pressure) at every node, measured in per-unit.
- Solving the equations tells us how the grid redistributes power after disturbances.
- Disturbances include feeder outages, lightning strikes, surges in EV charging demand, or generators tripping offline—events that force the network to reroute power safely.

Kirchhoff's current law for each node (i) is

$$\sum_{j \in \mathcal{N}(i)} g_{ij}(v_i - v_j) = p_i,$$

where (v) is node voltage (per-unit) and (p_i) is the net injection (generation positive, load negative). This assembles into the graph Laplacian system

$$L\,v = p,$$

with $L_{ii} = \sum_j g_{ij}$ and $L_{ij} = -g_{ij}$ for $i \neq j$. We fix one or more slack buses (reference voltages) to make the system well-posed.

PETSc for Graph Systems

- Graph Laplacians are sparse and often ill-conditioned.
- PETSc's `KSP` solvers with algebraic multigrid or incomplete factorization preconditioners scale to very large networks.
- MPI distribution lets you partition the graph and solve voltages for continental-scale grids in near real time.
- The `petsc4py` interface lets domain engineers script prototypes quickly while delegating heavy computation to PETSc.

Demo Script

The script `examples/11_petsc_graph_laplacian.py` builds a synthetic city grid (streets × avenues), injects power at generation nodes, and sinks power at neighborhoods. The steps are: 1. Generate the Laplacian (L) from the grid graph. 2. Impose a reference voltage at the main substation and a ground at the city perimeter. 3. Solve (L v = p) using PETSc's conjugate gradient with GAMG. 4. Report voltages and line currents, showing how power flows around a failed line.

The same scaffolding extends to real utility data (e.g., MATPOWER cases, CIM models).

Scaling Notes

- Increase grid resolution or import GIS-based line data to reach tens of millions of nodes.
- Couple with dynamic simulations (faults, renewables) by repeatedly solving updated Laplacians.
- Command-line options `--nx` and `--ny` describe how many "streets" (east-west lines) and "avenues" (north-south lines) we discretize, offering a convenient way to picture the synthetic city grid layout.
- For example, a run with `mpirun -n 64 python examples/11_petsc_graph_laplacian.py --nx 200 --ny 200 --load 0.8` would require a HPC cluster.

- Real-world grids are unstructured: lines weave irregularly through geography.
- Swap the synthetic mesh for your actual line list or GIS/CIM data when assembling (L); the PETSc solver workflow stays identical.
- GIS (geographic information system) layers capture where lines, substations, and loads sit in the city.
- CIM (Common Information Model) is the utility data standard that stores the same equipment and electrical parameters.
- Both GIS and CIM data can be fed directly into the Laplacian assembly.

Earthquake Ground Motion with PETSc

Background

- Earthquake early-warning systems and structural engineers need fast forecasts of how seismic waves will shake cities seconds after a fault ruptures.
- Full-resolution crust models contain millions of cells, making wave propagation simulation on laptops infeasible.
- Clusters with MPI and scalable solvers are standard.
- PETSc's distributed data structures and time integrators let us march the wave equation over large domains quickly, even with complex material maps.

Physics Model

We use a scalar acoustic approximation of seismic waves:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u + s(\mathbf{x}, t),$$

where $u(\mathbf{x}, t)$ is ground displacement, c is wave speed, and s is the source (fault slip). In this 2D acoustic demo, u represents a single out-of-plane (vertical) displacement. Full elastic solvers track three displacement components but follow the same parallel workflow. We rewrite this as leapfrog time stepping:

$$u^{n+1} = 2u^n - u^{n-1} + (c\Delta t)^2 \nabla^2 u^n + (\Delta t)^2 s^n.$$

Absorbing boundary layers mimic the Earth soaking up energy so reflections do not pollute the solution.

PETSc Implementation

The script `examples/12_petsc_wave_propagation.py` uses a PETSc DMDA to distribute a 2D grid (tens of thousands of points) across MPI ranks. Each step: 1. Convert the global field to a local array with halos. 2. Compute a finite-difference Laplacian. 3. Update displacement using the leapfrog scheme plus a tapered source term. 4. Apply damping near the boundaries to absorb outgoing waves.

- PETSc vectors store the wavefield, and MPI handles halo exchanges automatically.

- Real hazard models use unstructured meshes conforming to true topography and bedrock.
- Swap the structured DMDA for a DMPlex or imported mesh and the solver flow is unchanged.

Scaling Highlights

- Increase `--nx` and `--ny` to cover whole metropolitan regions at 5–10 m resolution.
- Coupled problems (elasticity, attenuation) simply extend the state vectors.
- A run with `mpirun -n 256 python examples/12_petsc_wave_propagation.py --nx 1024 --ny 1024 --steps 3000` would require a HPC cluster.

Training a neural network with PETSc

Background

- Hospitals, finance teams, and emergency services often need instant predictions (e.g., sepsis risk, fraud alerts) from large data feeds. Training models on millions of records requires splitting data across nodes while keeping model parameters synchronized, which is classic **data parallel** deep learning.
- Frameworks such as PyTorch hide the MPI details. PETSc lets you build the same scalable pattern from first principles, useful for custom workflows or HPC integration.
- Training a neural network is a numerical optimization problem, so we will use TAO (for optimization) with PETSc.

NOTE: This example demonstrates how PETSc can be used for distributed machine learning workflows, illustrating the underlying MPI communication patterns that power modern frameworks. In production, you would typically use PyTorch, TensorFlow, or JAX, which provide optimized implementations and GPU support. This workshop example serves as a pedagogical tool to understand the data-parallel training pattern and how MPI enables it.

Model and Objective

We train a logistic regression (single-layer neural net) with parameters $w \in \mathbb{R}^d$:

$$\hat{y} = \sigma(w^\top x), \qquad \sigma(z) = \frac{1}{1+e^{-z}}.$$

We minimize the average binary cross-entropy over samples $\{(x_i, y_i)\}_{i=1}^N$:

$$L(w) = -\frac{1}{N}\sum_{i=1}^{N} \left[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i) \right].$$

The gradient for each sample is $\nabla_w L_i = (\hat{y}_i - y_i)x_i$. The goal: each MPI rank owns a shard of the dataset, computes its local gradient, and all ranks **average gradients** to update the shared weight vector.

TAO + Deeper Network (examples/14_petsc_deep.py)

Neural Network as an Optimization Problem

Training a neural network is fundamentally an **unconstrained optimization problem**: given a loss function $L(\theta)$ parameterized by weights θ, find $\theta^* = \arg\min_\theta L(\theta)$. This is exactly what PETSc's TAO (Toolkit for Advanced Optimization) is designed to solve. By framing deep learning as optimization, we can leverage sophisticated numerical methods (L-BFGS, Newton, trust-region) instead of just stochastic gradient descent.

Architecture and Forward Pass

The example trains a three-layer binary classifier:

first hidden layer, $16 \to 32$

$$h_1 = \tanh(xW_1 + b_1)$$

second hidden layer, $32 \to 16$)

$$h_2 = \tanh(h_1 W_2 + b_2)$$

output layer, $16 \to 1$

$$\hat{y} = \sigma(h_2 W_3 + b_3)$$

where $\sigma(z) = 1/(1 + e^{-z})$ is the sigmoid function. The loss is binary cross-entropy:

$$L(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right].$$

Key Implementation Details

1. Parameter Packing

TAO requires a single flat vector for optimization. We pack all weights and biases into one NumPy array:

```
theta = [W1.ravel(), b1.ravel(), W2.ravel(), b2.ravel(), W3.ravel(), b3.ravel()]
```

The `unpack_params` function reverses this, reshaping the flat vector back into separate weight matrices for the forward pass. This is analogous to how PyTorch's optimizers work internally.

2. Backpropagation (lines 89-126)

The `forward_backward` function computes both the loss and its gradient using the chain rule: - **Forward pass:** compute activations a_1, a_2 and predictions $\hat{y}$ - **Backward pass:** compute gradients using δ (error terms) propagated from output to input: - Output layer: $\delta_3 = \hat{y} - y$ - Hidden layer 2: $\delta_2 = (\delta_3 W_3^T) \odot (1 - a_2^2)$ (tanh derivative) - Hidden layer 1: $\delta_1 = (\delta_2 W_2^T) \odot (1 - a_1^2)$

This is standard backpropagation, implemented in NumPy without automatic differentiation frameworks.

3. Data-Parallel Training via MPI

Each MPI rank owns a shard of the dataset (10,000 samples split across ranks). The `objective_gradient` callback: 1. **Local computation:** Each rank computes loss and gradients on its local data shard using mini-batches 2. **Global reduction:** `comm.allreduce` sums losses and `comm.Allreduce` sums gradients across all ranks 3. **Averaging:** Divide by total sample count to get the global gradient

This is the **data-parallel pattern**: replicate the model on all ranks, partition the data, and aggregate gradients. This is how PyTorch's `DistributedDataParallel` works under the hood.

4. TAO Solver Setup

```
tao = PETSc.TAO().create(comm=PETSc.COMM_SELF)  # Sequential TAO on each rank
tao.setType(args.tao_type)  # L-BFGS-VM by default
tao.setObjectiveGradient(objective_gradient, grad)
tao.solve(w)
```

Key points: - TAO is created with `COMM_SELF` (no TAO-level parallelism), but MPI parallelism happens inside `objective_gradient` - Each rank maintains a full copy of weights (redundant storage), but data is partitioned - L-BFGS-VM (`lmvm`) is a quasi-Newton method that approximates the Hessian, typically converging faster than vanilla gradient descent - Alternative solvers: `tron` (trust-region Newton), `nls` (Newton line search), `cg` (conjugate gradient)

5. Monitoring Convergence (lines 204-208)

The monitor callback displays iteration number, loss, and gradient norm:

```
its, f, res, cnorm, step, reason = tao.getSolutionStatus()
```

The gradient norm (`res`) should decrease as the optimizer converges. A norm below tolerance (e.g., 1e-5) indicates a local minimum.

Why This Matters for HPC

This example demonstrates: 1. **MPI communication patterns:** `Allreduce` for gradient aggregation is the bottleneck in data-parallel training 2. **Optimization perspective:** Neural networks aren't magic—they're just high-dimensional nonlinear least squares 3. **Solver flexibility:** By using TAO, you can experiment with advanced optimizers (trust-region, line search) that deep learning frameworks rarely expose 4. **Scalability:** The same code scales to hundreds of ranks by just changing `mpirun -n <ranks>`

Running at Scale

```
# Single node (debugging)
python examples/14_petsc_deep.py --samples 10000 --epochs 50

# Multi-node (production)
mpirun -n 16 python examples/14_petsc_deep.py --samples 200000 --features 64 --hidden1 256

# Try different optimizers
python examples/14_petsc_deep.py --tao_type tron  # Trust-region Newton
python examples/14_petsc_deep.py --tao_type nls   # Newton line search
```

Expected output: Loss should decrease from ~0.69 (random initialization) to ~0.40-0.45, with training accuracy ~75-80% on the synthetic data. Convergence reason `-2` means maximum iterations reached; `-3` to `-8` indicate convergence to tolerance.

Comparison to Production Frameworks

Feature	This Example	PyTorch DDP
Gradient computation	Manual backprop	Autograd
Parallelism	Explicit MPI `Allreduce`	Hidden in `DistributedDataParallel`
Optimizer	TAO (L-BFGS, trust-region)	SGD, Adam, etc.
Performance	Educational (CPU-only)	Production (GPU-optimized)

This example strips away the abstractions to show *how* data-parallel training works at the MPI level—knowledge directly applicable to designing custom HPC workflows.

Genome-Wide Association Screening with PETSc

Background

- Imagine you have genetic data from 50,000 patients (some healthy, some sick) and want to test 1 million genetic positions to find which ones correlate with disease.
- This is essentially testing 1 million hypotheses—computing a statistical score for each position and ranking them. The data is too large for one machine, so we distribute the computation across nodes using MPI.
- Biomedical labs routinely run genome-wide association studies (GWAS) to find variants linked to diseases (e.g., cancer risk, pharmacogenomics).
- Modern cohorts contain millions of genetic variants across hundreds of thousands of patients, which is too large for a single machine.
- MPI lets us split variants (or samples) across nodes while still producing global statistics such as allele frequencies or association scores.
- This is a classic **embarrassingly parallel** problem: each genetic variant can be tested independently, making it ideal for distributed computing.

Statistical Model

What We're Computing:

- For each genetic position, we have a value of 0, 1, or 2 for each patient (representing how many copies of a variant they have).
- We also have a binary label (0=healthy, 1=sick).
- The question: does this genetic position appear more frequently in sick patients than healthy ones?

For each variant j with genotypes $g_{ij} \in \{0, 1, 2\}$ (minor-allele counts) and binary phenotype $y_i \in \{0, 1\}$, we compute a chi-square test: 1. Count minor and major alleles in cases vs controls. 2. Form a 2×2 contingency table of allele counts. 3. Compute the Pearson chi-square statistic: $\chi^2 = \sum \frac{(O-E)^2}{E}$ where O is observed counts and E is expected counts under independence.

MPI Strategy: Each rank owns disjoint sets of variants (columns of the genotype matrix), but all ranks need access to the full phenotype vector (which is broadcasted). PETSc distributed vectors hold the chi-square scores so we can locate the top associated variants globally.

Demo Script (examples/15_petsc_gwas.py)

The script simulates genotypes with random minor-allele frequencies and a binary phenotype. Each MPI rank works on a contiguous block of variants, computes local allele statistics, and stores chi-square values in a PETSc distributed vector.

How the Code Works

1. Data Distribution (lines 20-26, 110-111)

The `compute_local_range` function divides variants across ranks using a balanced strategy that handles remainders:

```
start_idx, end_idx = compute_local_range(args.variants, size, rank)
local_variants = end_idx - start_idx
```

Example: 50,000 variants across 4 ranks → each rank owns ~12,500 variants (rank 0: 0-12499, rank 1: 12500-24999, etc.)

2. Phenotype Broadcasting (lines 106-108)

All ranks need the full phenotype vector (case/control status for all patients) since each rank's variants must be tested against all patients:

```
phenotype = phen_rng.binomial(1, 0.5, size=args.samples)  # Rank 0 generates
comm.Bcast(phenotype, root=0)  # Broadcast to all ranks
```

This is a one-time $O(N)$ communication where N is the number of patients.

3. Local Genotype Generation (lines 29-41)

Each rank independently generates its assigned variants using offset seeds to ensure reproducibility:

```
rng = np.random.default_rng(seed + start_idx * 97)  # Unique seed per rank
genotypes[:, j] = rng.binomial(2, maf, size=num_samples)  # 0, 1, or 2 copies
```

This avoids storing or transferring the entire genotype matrix—each rank generates only what it needs.

4. Chi-Square Computation (lines 44-92)

This is pure local computation with no MPI communication. For each variant owned by the rank: - Split patients into cases and controls using the phenotype vector - Count minor/major alleles in each group: $minor_{cases} = \sum_{i \in cases} g_{ij}$ - Build a 2×2 contingency table:

	Minor Allele	Major Allele
Cases	O_{11}	O_{12}
Controls	O_{21}	O_{22}

- Compute expected counts under independence: $E_{ij} = \frac{(\text{row total}) \times (\text{col total})}{\text{grand total}}$
- Calculate chi-square: $\chi^2 = \sum_{i,j} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$

Higher χ^2 indicates stronger association between the variant and disease status.

5. Distributed Vector Assembly (lines 123-133)

Each rank stores its local chi-square results in a PETSc distributed vector:

```
chi2_vec = PETSc.Vec().createMPI(args.variants, bsize=local_variants, comm=PETSc.COMM_WORLD
chi2_arr = chi2_vec.getArray()  # Get local portion
chi2_arr[:] = chi2_local  # Fill with computed values
chi2_vec.assemblyBegin()
chi2_vec.assemblyEnd()  # Finalize parallel assembly
```

The vector is distributed: rank 0 owns entries [0:12500), rank 1 owns [12500:25000), etc.

6. Global Maximum Finding (lines 135-144)

PETSc's `Vec.max()` performs an `MPI_Allreduce` to find the global maximum across all ranks:

```
max_val, max_idx = chi2_vec.max()  # Returns (value, global index)
```

Then we determine which rank owns this index and retrieve its minor allele frequency using ownership ranges and broadcasting.

7. Top-K Reporting (lines 151-161)

Each rank sorts its local variants and sends top-K to rank 0:

```
local_pairs.sort(key=lambda x: x[1], reverse=True)  # Sort by chi-square
gathered = comm.gather(top_local, root=0)  # Gather to rank 0
```

Rank 0 merges and re-sorts to find the global top-K variants.

Key MPI Patterns Demonstrated

1. **Broadcast:** Phenotype vector shared across all ranks (one-to-all communication)
2. **Embarrassingly parallel:** Chi-square computation is fully independent per variant
3. **Distributed data structures:** PETSc vectors provide natural abstraction for distributed results
4. **Collective reductions:** `Vec.max()` and `comm.gather()` aggregate results across ranks
5. **Ownership-based access:** Each rank queries which indices it owns to retrieve metadata

This demonstrates the core pattern used in production GWAS pipelines that scale to millions of variants across clusters.

Scaling Notes

Swap the synthetic generator for real VCF or PLINK readers; the PETSc/MPI communication remains unchanged. Add covariates or logistic regression by extending the local gradient to TAO. Run on clusters with `mpirun -n 32 python examples/15_petsc_gwas.py --samples 50000 --variants 1000000`.

Summary of Examples

Heat Equation: Domain Decomposition

We split the simulation grid across ranks with halo and ghost exchanges. Synchronizing boundary conditions after each timestep is necessary. Scaling considerations include load balance and communication cost. See `05_heat_equation_mpi.py` as a template for diffusion problems.

Deep Learning: Data Parallelism

We partition the dataset per rank for independent training passes. Synchronizing model updates through collective operations like allreduce is necessary. Handling randomness and reproducibility across processes matters. Reference `06_deep_learning_data_parallel.py` for an end-to-end example.

Discussion Prompts

Compare domain and data parallel strategies. Explore opportunities for hybrid model and data parallel approaches. Define success metrics such as speedup and accuracy retention.

PETSc Optimization: MRI T2 Fitting

Calibrate T2 decay parameters from synthetic MRI spin-echo data using PETSc TAO. This illustrates mapping a least-squares objective onto distributed vectors and shows how `petsc4py` exposes TAO quasi-Newton optimizers with minimal boilerplate. Try `mpirun -n 2 python examples/08_petsc_tao_t2_fit.py` and compare recovered vs true parameters.

PETSc/SLEPc Eigenmodes: Bridge Dynamics

Build stiffness and mass matrices for a rectangular bridge deck with finite differences. Use SLEPc to extract the lowest vibration mode and approximate natural frequency. This demonstrates scalable eigenvalue solves in `petsc4py` and `slepc4py`. Run `mpirun -n 4 python examples/09_petsc_eigen_bridge.py` and inspect the mode profile.

PETSc SNES: Thermal Runaway (Bratu)

Solve a Bratu-style nonlinear heat equation mimicking battery thermal runaway. Uses SNES with DMDA to distribute the grid and assemble residual and Jacobian in parallel. Try `mpirun -n 4 python examples/10_petsc_snes_bratu.py -nx 128 -ny 64 -lambda 6.0`.

PETSc Graph Laplacian: Power Network

Model a city-scale power grid as a graph and solve the Laplacian (L v = p) for node voltages. This demonstrates PETSc KSP with GAMG for sparse graph systems with Dirichlet constraints. Run `mpirun -n 4 python examples/11_petsc_graph_laplacian.py --nx 80 --ny 40 --load 0.8`.

PETSc Wave Propagation: Quake Simulation

March the 2D acoustic wave equation to mimic earthquake ground motion across a city basin. Uses DMDA vectors, MPI halo exchange, and explicit leapfrog stepping with absorbing boundaries. Run `mpirun -n 8 python examples/12_petsc_wave_propagation.py --nx 400 --ny 240 --steps 2000`.

PETSc Data Parallel ML: Logistic Regression

Demonstrates MPI-driven gradient averaging for a logistic classifier across large datasets. Wraps NumPy arrays in PETSc vectors to reuse BLAS updates while sharing gradients via MPI collectives. Run `mpirun -n 16 python examples/13_petsc_data_parallel_gd.py --samples 1000000 --features 128 --epochs 10`.

PETSc TAO Deep Classifier

Uses TAO with a redundant vector to keep full weights on every rank while averaging gradients via MPI. Shows how to pack and unpack two hidden layers, enabling custom deep-learning research pipelines without heavyweight frameworks. Run `mpirun -n 16 python examples/14_petsc_deep.py --samples 200000 --features 64 --hidden1 256 --hidden2 128`.

PETSc/MPI Genomics GWAS Scan

Partitions variants across ranks and computes chi-square association statistics for each SNP. Demonstrates how PETSc vectors store distributed scores while MPI reductions find the top hits. Run `mpirun -n 32 python examples/15_petsc_gwas.py --samples 50000 --variants 1000000`.

Summary and Conclusion

Key Takeaways

We assembled Laplacians, nonlinear residuals, and objective gradients that respect partitioned halos—core mechanics for battery thermal models, seismic simulations, and power grids. We practiced gradient sharing and parameter updates with MPI reductions, providing the blueprint for integrating custom deep-learning workflows on HPC systems. We learned how to choose PETSc components (solvers, preconditioners, TAO strategies) via runtime options, enabling rapid experimentation without recompiling.

You can scale Python from a single workstation to shared clusters by combining `mpi4py`, PETSc, SLEPc, and TAO, matching the tools used in production HPC environments. Structured codes

(DMDA) and unstructured meshes (DMPlex) share the same MPI-aware patterns. Swapping grids or physics modules does not require redesigning the solver flow. PETSc's layered architecture (Vec/Mat, KSP, SNES, TAO) unlocks linear solves, nonlinear PDEs, eigenvalue analysis, and optimization without rewriting your application. Data parallel ideas carry across domains: gradient reductions for machine learning mirror the collective communications in PDE solvers and graph analytics.

The MPI standard documentation provides authoritative definitions. The `mpi4py` guide covers API usage and code examples. The PETSc manual offers solver selection and configuration guidance.

Final Thoughts

Keep iterating: start with small test cases, verify convergence and stability, then scale to regional or enterprise-sized problems. Lean on PETSc's documentation and mailing lists. When problems grow beyond single nodes, these tools and communities help you stay productive. Workshop 2 set the CPU-parallel foundation. Workshop 3 will layer GPU acceleration on top of the MPI concepts.

Next in this series

Workshop 2 covered CPU-based parallelism. GPU acceleration builds directly on MPI concepts.

Workshop 3 will focus areas include GPU architectures and how to access GPU resources from Python.

- Workshop 3 (October 30): Accelerating your code with GPUs