MPI 2D Convolution – HOWTO

This project implements a distributed 2D convolution operator using `mpi4py` and hand-written (Numba-accelerated) kernels. The primary goals are correctness, reproducibility, and the ability to benchmark strong/weak scaling as well as throughput under different problem sizes.

Environment

- Python 3.10+
- MPI runtime (OpenMPI, MPICH, Intel MPI, etc.)
- Python packages:
  - `mpi4py`
  - `numpy`
  - `numba`
  - `opencv-python` (image I/O)
  - Optional validation backends: `scipy`, `torch`

Install Python dependencies:

```
python -m pip install mpi4py numpy numba opencv-python scipy torch
```

Repository Layout

```
mpi_conv.py        # CLI entry point
domain.py          # 2D block decomposition helpers
comm.py            # Scatter/Gather helpers, halo exchange
kernels.py         # Convolution kernels (Numba + Python fallback)
io_utils.py        # OpenCV I/O and synthetic generation
validate.py        # Reference implementations (naive/SciPy/Torch)
bench.py           # CSV logging helpers
report_utils.py    # Aggregation utilities
README_HOWTO.md    # This guide
```

Quick Start

Execute a single-run strong-scaling experiment on synthetic data:

```
mpirun -n 4 python mpi_conv.py \
  --synthetic 2048 2048 \
  --kernel-size 7 \
  --stride 1 \
  --padding same \
  --px 2 --py 2 \
  --csv results.csv
```

Load an image, convolve with a custom kernel, and save the output (Numba disabled for debugging):

```
mpirun -n 2 python mpi_conv.py \
  --image data/cat.png \
  --kernel kernels/laplacian.npy \
  --padding same \
  --stride 1 \
  --save-output output/cat_laplace.png \
  --numba-disable
```

Validate correctness against naive, SciPy, and PyTorch references:

```
mpirun -n 4 python mpi_conv.py \
  --synthetic 64 64 \
  --kernel-size 5 \
  --padding same \
  --stride 2 \
  --cin 3 --cout 2 \
  --check --check-scipy --check-torch
```

Benchmark Matrix

Suggested matrix for comprehensive evaluation (adjust path/grid as needed):

```
Image sizes:   256², 512², 1024², 2048²
Kernel sizes:  3, 5, 7, 11
Strides:       1, 2
Padding:       0, K//2  (valid / same)
Batches:       1, 4, 8, 16, 32
Ranks:         1, 2, 4, 8, 16  (px × py factorizations)
```

Strong scaling example (fixed 2048^2 image, 7×7 kernel):

```
mpirun -n 1  python mpi_conv.py --synthetic 2048 2048 --kernel-size 7 --padding
same --stride 1 --px 1 --py 1  --csv strong.csv --baseline 10.0
mpirun -n 4  python mpi_conv.py --synthetic 2048 2048 --kernel-size 7 --padding
same --stride 1 --px 2 --py 2  --csv strong.csv --baseline 10.0
mpirun -n 16 python mpi_conv.py --synthetic 2048 2048 --kernel-size 7 --padding
same --stride 1 --px 4 --py 4 --csv strong.csv --baseline 10.0
```

Weak scaling example (increase problem size with ranks):

```
mpirun -n 1  python mpi_conv.py --synthetic 1024 1024 --kernel-size 7 --padding
same --px 1 --py 1 --csv weak.csv
mpirun -n 4  python mpi_conv.py --synthetic 2048 2048 --kernel-size 7 --padding
same --px 2 --py 2 --csv weak.csv
mpirun -n 16 python mpi_conv.py --synthetic 4096 4096 --kernel-size 7 --padding
same --px 4 --py 4 --csv weak.csv
```

Throughput sweep across batch sizes:

```
for b in 1 4 8 16 32; do
  mpirun -n 8 python mpi_conv.py \
    --synthetic 1024 1024 \
    --kernel-size 5 \
    --stride 2 \
    --padding same \
    --cin 3 --cout 8 \
    --batch $b \
    --px 2 --py 4 \
    --csv throughput.csv
done
```

CSV Output

Each run appends a row (if `--csv` provided) with the following schema:

```
ts, H, W, K, S, P, Cin, Cout, batch, ranks, px, py, halo,
t_total, t_comp, t_comm, speedup, eff, host
```

- `t_total`: wall-clock runtime (s) measured with `MPI.Wtime`.
- `t_comp`: median compute time across ranks.
- `t_comm`: median communication time (scatter/halo/gather).
- `speedup`: optional; computed when `--baseline` is supplied.
- `eff`: `speedup / ranks`.

Use `report_utils.py` (extend as needed) to aggregate/plot the CSV.

Notes

- Halo exchange uses a two-stage non-blocking scheme (vertical then horizontal) to enable compute/comm overlap.
- Kernels default to Numba-accelerated implementations; `--numba-disable` forces pure Python loops for debugging.
- Validation is optional; enable with `--check`. SciPy/Torch comparisons require their respective packages.
- All MPI I/O occurs on rank 0; other ranks operate on sub-domains derived from `domain.decompose_2d`.

- The implementation avoids MKL/cuBLAS—compute is performed by explicit loops (compiled by Numba when available).