The Journal of Open Source Software

# jaxDecomp: JAX Library for 3D Domain Decomposition and Parallel FFTs

**Wassim Kabalan** [1], **François Lanusse** [2,3], **Alexandre Boucaud**[1], **and Eric Aubourg**[1]

**1** Université Paris Cité, CNRS, Astroparticule et Cosmologie, F-75013 Paris, France **2** Université Paris-Saclay, Université Paris Cité, CEA, CNRS, AIM, 91191, Gif-sur-Yvette, France **3** Flatiron Institute, Center for Computational Astrophysics, 162 5th Avenue, New York, NY 10010, USA

**DOI:** 10.21105/joss.08852

**Software**

- Review
- Repository
- Archive

**Editor:** Patrick Diehl

**Reviewers:**

- @rafmudaf
- @k20shores

**Submitted:** 25 June 2025
**Published:** 16 May 2026



## Summary

`JAX` (Bradbury et al., 2018) has become a popular framework for machine learning and scientific computing, offering high performance, composability, and distributed computing. However, its use as a full HPC framework has been limited by partial native support for key distributed operations. Libraries like `MPI4JAX` (Häfner & Vicentini, 2021) enable large-scale parallelism but face limitations, notably buffer size constraints and incompatibility with native JAX distribution, making it hard to use with the JAX ecosystem.

The introduction of JAX's unified array API and tools like `pjit` and `custom_partitioning` has made SPMD-style programming more accessible. However, many HPC workflows require specialized operations such as optimized distributed Fast Fourier Transforms (FFTs) or halo exchange operations.

To fill this gap, we present `jaxDecomp`, a fully differentiable JAX library for distributed 3D FFTs and halo exchanges. It wraps NVIDIA's `cuDecomp` library (Romero et al., 2022), exposing its functionality as JAX primitives while maintaining compatibility with JAX transformations like `jit` and `grad`. Beyond basic distributed FFTs, `jaxDecomp` provides halo exchange operations and automatic optimization of communication backends (NCCL, MPI, NVSHMEM) based on the target hardware. Benchmarks show competitive performance with native JAX while adding HPC-specific features.

## Statement of Need

For numerical simulations on HPC systems, a distributed, easy-to-use, and differentiable FFT is essential for achieving peak performance and scalability. While JAX now provides native distributed FFT support, this was introduced only very recently and lacks the specialized HPC features required by many applications. There is a pressing need for a solution that provides not only distributed FFTs but also halo exchanges, optimized communication backends, and seamless integration with existing cluster infrastructure.

In scientific applications such as cosmological particle mesh (PM) simulations, specialized frameworks like `FlowPM` (Modi et al., 2020) built on `mesh-TensorFlow` (Shazeer et al., 2018) or JAX-based codes like `pmwd` (Li et al., 2022) often struggle to scale beyond single-node memory limits or rely on manual distribution strategies. These challenges highlight the need for a scalable, high-performance approach to distributed FFTs that remains differentiable for advanced algorithms (like Hamiltonian Monte Carlo (Brooks et al., 2011) or the No-U-Turn Sampler (NUTS) (Hoffman & Gelman, 2011)).

# Implementation

## Distributed FFT Algorithm

`jaxDecomp` performs 3D FFTs by applying 1D FFTs along the Z, Y, and X axes, with global transpositions between steps to ensure each axis is locally accessible. This enables fully local computation while distributing the workload across devices.

The table below summarizes the FFT-transpose sequence:

| Steps | Operation Description |
|---|---|
| FFT along Z | Batched 1D FFT along the Z-axis. |
| Transpose Z to Y | Transpose to $Z \times X \times Y$. Partition the Y-axis |
| FFT along Y | Batched 1D FFT along the Y-axis. |
| Transpose Y to X | Transpose to $Y \times Z \times X$. Partition the X-axis |
| FFT along X | Batched 1D FFT along the X-axis. |

`jaxDecomp` uses pencil and slab decomposition schemes to distribute 3D data across GPUs. Each FFT step is followed by a transposition that reshapes and rebalances the array to align the next axis for local computation. More technical details, including decomposition strategies and axis layouts, are available in the documentation.

## Distributed Halo Exchange

`jaxDecomp` includes efficient halo exchange operations required in stencil computations and PDE solvers. It supports multiple backends—`NCCL`, `MPI`, and `NVSHMEM`—to adapt to different cluster architectures and communication patterns. Details and visuals of the halo exchange logic are available in the documentation.

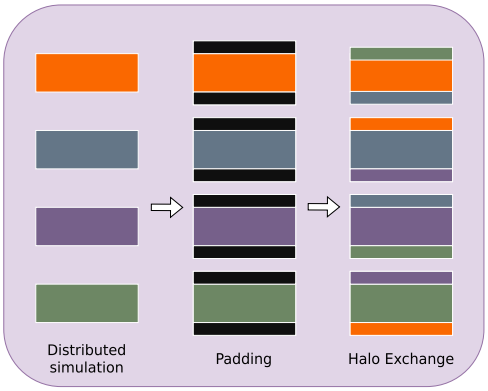


**Figure 1:** Visualization of the distributed halo exchange process in `jaxDecomp`

# Benchmarks

Benchmarks were run on the Jean Zay supercomputer using NVIDIA H100 GPUs to evaluate the scaling behavior of distributed 3D FFTs across 2 to 256 GPUs, comparing single-precision (float32) and double-precision (float64) performance.

For strong scaling, a fixed global grid ($512^3$, $1024^3$, $2048^3$, or $4096^3$) is distributed across an increasing number of GPUs. For weak scaling, a fixed local volume of 1,048,576 elements per GPU is maintained as the GPU count grows. Both experiments test multiple pencil and slab decomposition strategies, reporting the fastest configuration at each GPU count.

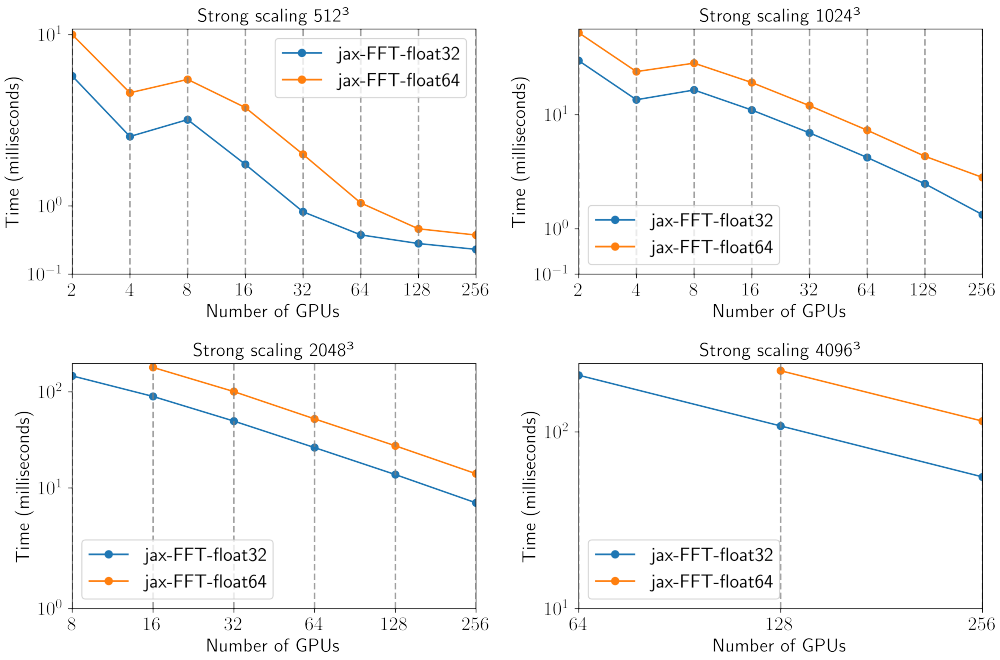


**Figure 2:** Strong scaling of distributed 3D FFTs on the Jean Zay supercomputer (H100 GPUs). Each subplot shows a fixed global grid size with execution time decreasing as GPUs increase.

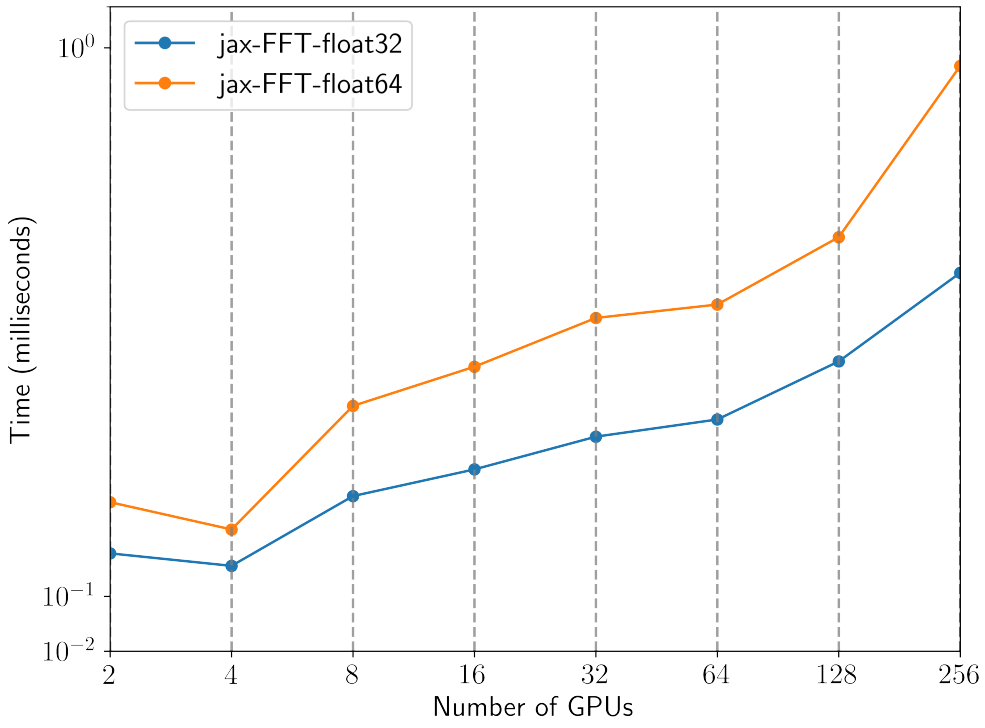


**Figure 3:** Weak scaling with a fixed local volume of 1,048,576 elements per GPU. Execution time remains nearly flat as GPU count increases from 2 to 256, demonstrating high parallel efficiency.

## Stability and releases

We aim for 100% test coverage across all core functionalities: FFT, halo exchange, and transposition. The code has been tested on the Jean Zay supercomputer, with simulations distributed on 64 GPUs. The package is available on PyPI and can be installed via `pip install jaxDecomp`.

### Contributing and Community

Contributions are welcome. The project follows clear guidelines and uses `yapf` and `pre-commit` for formatting. A full guide is available in the repository's `CONTRIBUTING.md`. Users and developers are invited to participate by opening issues, submitting pull requests, or joining discussions via GitHub.

## Acknowledgements

This work was granted access to the HPC resources of IDRIS under the allocation 2024-AD011014949 made by GENCI. The computations in this work were, in part, run at facilities supported by the Scientific Computing Core at the Flatiron Institute, a division of the Simons Foundation.

We also acknowledge the SCIPOL scipol.in2p3.fr funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (PI: Josquin Errard, Grant agreement No. 101044073).

#### Appendix

Particle-Mesh Example (PM Forces) The following example computes gravitational forces using a Particle-Mesh (PM) scheme in a JAX-based setup, running on multiple GPUs with `jaxDecomp`.while remaining fully differentiable.

```
import jax.numpy as jnp
import jaxdecomp

def pm_forces(density):
    # `density` is a 3D distributed array of shape (Nx, Ny, Nz) is defined over the simu
    delta_k = jaxdecomp.fft.pfft3d(density)
    ky, kz, kx = jaxdecomp.fft.fftfreq3d(delta_k)
    kk = kx**2 + ky**2 + kz**2
    laplace_kernel = jnp.where(kk == 0, 1.0, -1.0 / kk)
    pot_k = delta_k * laplace_kernel
    forces = [-jaxdecomp.fft.pifft3d(1j * k * pot_k) for k in [kx, ky, kz]]
    return jnp.stack(forces, axis=-1)
```

A more detailed example of an LPT simulation can be found in the jaxdecomp_lpt example.

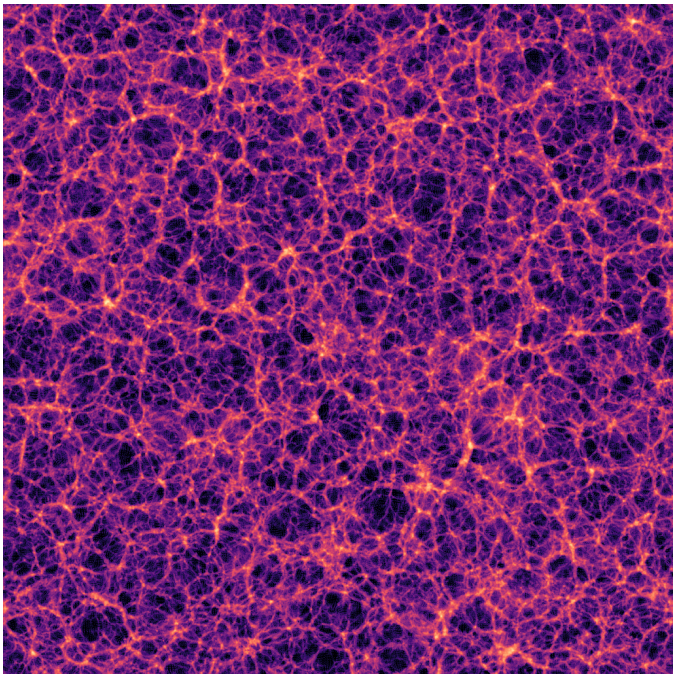

**Figure 4:** Visualization of an LPT density field at z=0 for a $2048^3$ grid generated using `jaxDecomp`.

I refer the reader to the full API description in the jaxDecomp documentation.

## References


Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.5.2). http://github.com/google/jax

Brooks, S., Gelman, A., Jones, G., & Meng, X.-L. (2011). *Handbook of Markov Chain Monte Carlo*. Chapman; Hall/CRC. https://doi.org/10.1201/b10905

Häfner, D., & Vicentini, F. (2021). mpi4jax: Zero-copy MPI communication of JAX arrays. *Journal of Open Source Software*, *6*(65), 3419. https://doi.org/10.21105/joss.03419

Hoffman, M. D., & Gelman, A. (2011). *The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo*. https://doi.org/10.48550/arXiv.1111.4246

Li, Y., Lu, L., Modi, C., Jamieson, D., Zhang, Y., Feng, Y., Zhou, W., Kwan, N. P., Lanusse, F., & Greengard, L. (2022). *pmwd: A differentiable cosmological particle-mesh N-body library*. https://doi.org/10.48550/arXiv.2211.09958

Modi, C., Lanusse, F., & Seljak, U. (2020). *FlowPM: Distributed TensorFlow implementation of the FastPM cosmological N-body solver*. https://doi.org/10.48550/arXiv.2010.11847

Romero, J., Costa, P., & Fatica, M. (2022). Distributed-memory simulations of turbulent flows on modern GPU systems using an adaptive pencil decomposition library. *Proceedings of the Platform for Advanced Scientific Computing Conference*. https://doi.org/10.1145/3539781.3539797

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., & Hechtman, B. (2018). *Mesh-TensorFlow: Deep learning for supercomputers*. https://doi.org/10.48550/arXiv.1811.02084