torch-sla: Differentiable Sparse Linear Algebra with Sparse Tensor Parallelism and Adjoint Solvers for PyTorch

Mingyuan Chi
walker.chi.000@gmail.com
https://github.com/walkerchi/torch-sla

January 15, 2026

Abstract

We present `torch-sla`, an open-source PyTorch library for differentiable sparse linear algebra that seamlessly integrates with deep learning workflows. The library provides two key innovations: (1) **Sparse Tensor Parallel** computing via domain decomposition with halo exchange, enabling distributed sparse matrix operations across multiple GPUs following industrial CFD/FEM practices; and (2) **Adjoint-based differentiation** for both linear and nonlinear sparse solvers, providing memory-efficient gradient computation with $O(1)$ computational graph nodes regardless of solver iterations. `torch-sla` supports multiple backends (SciPy, Eigen, cuSOLVER, cuDSS, PyTorch-native) and scales to over 169 million degrees of freedom on a single GPU. Benchmarks demonstrate near-linear $O(n^{1.1})$ time complexity for iterative solvers and 12× speedup on multi-GPU configurations compared to single CPU. The library is available at `https://github.com/walkerchi/torch-sla` and can be installed via `pip install torch-sla`.

Keywords: Sparse linear algebra, automatic differentiation, adjoint method, distributed computing, PyTorch, finite element method, computational fluid dynamics

1 Introduction

Sparse linear systems $\mathbf{Ax} = \mathbf{b}$ arise ubiquitously in scientific computing, from finite element analysis to graph neural networks. With the rise of physics-informed machine learning and differentiable programming, there is an increasing demand for sparse solvers that integrate seamlessly with automatic differentiation frameworks like PyTorch [Paszke et al., 2019].

Existing sparse linear algebra libraries face several challenges when used in differentiable programming contexts:

1. **Gradient support**: Most sparse solvers (SciPy, cuSOLVER) are not differentiable, requiring manual gradient implementation.
2. **Memory efficiency**: Naive differentiation through iterative solvers creates $O(k)$ computational graph nodes where k is the number of iterations.
3. **Scalability**: Single-GPU memory limits problem sizes, while distributed sparse matrix operations require complex halo exchange patterns.
4. **Backend fragmentation**: Different backends (CPU direct, GPU direct, GPU iterative) have different APIs and performance characteristics.

This paper introduces `torch-sla`, a unified library that addresses these challenges through two main contributions:

Contribution 1: Sparse Tensor Parallel Computing We implement a distributed sparse matrix class (`DSparseMatrix`) that partitions large matrices across multiple GPUs using domain decomposition with automatic halo exchange. This follows the industrial approach used in Ansys Fluent, OpenFOAM, and other production CFD/FEM codes. Our implementation supports:

- METIS-based graph partitioning for load balancing
- Peer-to-peer halo exchange via NCCL (GPU) or Gloo (CPU)
- Distributed Conjugate Gradient (CG) and LOBPCG eigensolvers
- Memory-efficient design enabling problems with 100M+ DOF

Contribution 2: Adjoint-Based Differentiation We provide adjoint-based gradient computation for both linear and nonlinear solvers:

- **Linear solve**: Gradients via transposed system solve, $O(1)$ graph nodes
- **Eigenvalue solve**: Adjoint eigenvalue gradients using implicit differentiation
- **Nonlinear solve**: Newton/Anderson solvers with adjoint-based implicit differentiation for $F(\mathbf{u}, \boldsymbol{\theta}) = 0$

The remainder of this paper is organized as follows: Section 2 provides background on sparse linear algebra and adjoint methods. Section 3 describes our implementation of sparse tensor parallelism and adjoint solvers. Section 4 presents benchmark results. Section 5 discusses related work, and Section 6 concludes.

2 Background

2.1 Sparse Linear Systems

A sparse linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ where $\mathbf{A} \in \mathbb{R}^{n \times n}$ has $\text{nnz}(\mathbf{A}) \ll n^2$ non-zero entries. Storage formats include:

- **COO (Coordinate)**: Store (i, j, v) triplets for each non-zero
- **CSR (Compressed Sparse Row)**: Store row pointers, column indices, and values
- **CSC (Compressed Sparse Column)**: Column-major variant of CSR

Solvers are categorized as:

- **Direct solvers**: LU/Cholesky factorization, $O(n^{1.5})$ memory for 2D problems due to fill-in
- **Iterative solvers**: CG, BiCGStab, GMRES, $O(\text{nnz})$ memory but require $O(k)$ iterations

2.2 Automatic Differentiation for Linear Solves

Given a loss function $\mathcal{L}(\mathbf{x})$ where $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$, we seek gradients $\partial\mathcal{L}/\partial\mathbf{A}$ and $\partial\mathcal{L}/\partial\mathbf{b}$.

Using the identity $\mathbf{A}\mathbf{x} = \mathbf{b}$ and implicit differentiation:

$$d\mathbf{A} \cdot \mathbf{x} + \mathbf{A} \cdot d\mathbf{x} = d\mathbf{b} \tag{1}$$

$$\mathbf{A} \cdot d\mathbf{x} = d\mathbf{b} - d\mathbf{A} \cdot \mathbf{x} \tag{2}$$

For the gradient with respect to $\mathbf{b}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \mathbf{A}^{-\top} \frac{\partial \mathcal{L}}{\partial \mathbf{x}} \tag{3}$$

For the gradient with respect to the sparse values $\mathbf{A}_{ij}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}_{ij}} = -\left(\mathbf{A}^{-\top} \frac{\partial \mathcal{L}}{\partial \mathbf{x}} \right)_i \cdot \mathbf{x}_j \tag{4}$$

The key insight is that computing both gradients requires only *one additional linear solve* of the transposed system $\mathbf{A}^\top \boldsymbol{\lambda} = \partial \mathcal{L} / \partial \mathbf{x}$, regardless of the number of forward solver iterations. This is the **adjoint method**.

2.3 Adjoint Method for Nonlinear Systems

Consider a nonlinear system $\mathbf{F}(\mathbf{u}, \boldsymbol{\theta}) = \mathbf{0}$ where $\mathbf{u}$ is the solution and $\boldsymbol{\theta}$ are parameters. Given a loss $\mathcal{L}(\mathbf{u}^*)$ evaluated at the solution $\mathbf{u}^*$, we seek $\partial \mathcal{L} / \partial \boldsymbol{\theta}$.

The adjoint equation is:

$$\left(\frac{\partial \mathbf{F}}{\partial \mathbf{u}} \right)^\top \boldsymbol{\lambda} = \frac{\partial \mathcal{L}}{\partial \mathbf{u}} \tag{5}$$

The parameter gradient is:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\boldsymbol{\lambda}^\top \frac{\partial \mathbf{F}}{\partial \boldsymbol{\theta}} \tag{6}$$

This requires:

1. Solving the forward problem $\mathbf{F}(\mathbf{u}^*, \boldsymbol{\theta}) = \mathbf{0}$ (e.g., via Newton-Raphson)
2. Solving the adjoint equation (5) (one linear solve)
3. Computing the Jacobian-vector product in (6)

The total memory cost is $O(1)$ graph nodes, independent of Newton iterations.

3 Methodology

3.1 Architecture Overview

`torch-sla` provides a unified API across multiple backends (Figure 1):

Backend	Device	Methods	Best For
`scipy`	CPU	SuperLU, UMFPACK, CG	Default CPU
`eigen`	CPU	CG, BiCGStab	Alternative CPU
`cudss`	CUDA	LU, Cholesky, LDLT	Direct CUDA ($<$ 2M DOF)
`cusolver`	CUDA	QR, Cholesky, LU	Legacy CUDA
`pytorch`	CPU/CUDA	CG, BiCGStab	Large-scale ($>$ 2M DOF)

Figure 1: Available backends in `torch-sla`.

The `SparseTensor` class provides a high-level interface:

```
from torch_sla import SparseTensor

# Create sparse matrix
A = SparseTensor(val, row, col, shape)

```

```
# Solve with automatic backend selection
x = A.solve(b)  # Gradients flow automatically

# Eigenvalues with adjoint gradients
eigenvalues, eigenvectors = A.eigsh(k=6)

# Nonlinear solve with implicit differentiation
u = A.nonlinear_solve(residual_fn, u0, *params)
```

3.2 Sparse Tensor Parallel Computing

For large-scale problems that exceed single-GPU memory, we implement domain decomposition with halo exchange following industrial CFD/FEM practices.

3.2.1 Domain Decomposition

Given a sparse matrix $\mathbf{A}$ corresponding to a mesh/graph, we partition nodes into P subdomains using either:

- **METIS partitioning**: Minimizes edge cuts for load balancing
- **Geometric (RCB)**: Recursive Coordinate Bisection based on node coordinates
- **Simple**: Contiguous row partitioning (fallback)

Each partition p owns a set of nodes $\mathcal{O}_p$ and has halo nodes $\mathcal{H}_p$ from neighboring partitions required for local matrix-vector products.

3.2.2 Halo Exchange

For distributed sparse matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$, each partition must exchange halo values (Section 3.2.2):

Algorithm 1: Distributed SpMV with Halo Exchange

Input: Local vector $\mathbf{x}_{\text{local}}$, neighbor information

1. **Send** owned boundary values to neighbors (async)
2. **Receive** halo values from neighbors (async)
3. **Synchronize** communication
4. $\mathbf{y}_{\text{owned}} \leftarrow \mathbf{A}_{\text{local}}\mathbf{x}_{\text{local}}$ *// Includes halo columns*
5. **Return** $\mathbf{y}_{\text{owned}}$

We use PyTorch's `torch.distributed` with NCCL backend for GPU-GPU communication and Gloo for CPU.

3.2.3 Distributed Iterative Solvers

For Conjugate Gradient on distributed matrices:

Each iteration requires:

- One halo exchange (for SpMV)
- Two `all_reduce` operations (for dot products)

Algorithm 2: Distributed Conjugate Gradient

Input: Distributed matrix $\mathbf{A}$, local RHS $\mathbf{b}_{\text{owned}}$, tolerance ϵ

1. $\mathbf{x} \leftarrow \mathbf{0}$, $\mathbf{r} \leftarrow \mathbf{b}_{\text{owned}}$, $\mathbf{p} \leftarrow \mathbf{r}$
2. $\rho \leftarrow \text{all\_reduce}(\mathbf{r}^\top \mathbf{r}, \text{SUM})$ *// Global dot product*
3. **while** $\sqrt{\rho} > \epsilon$ **do**
4. $\quad \mathbf{Ap} \leftarrow \text{DistributedSpMV}(\mathbf{A}, \mathbf{p})$ *// Halo exchange*
5. $\quad \alpha \leftarrow \rho / \text{all\_reduce}(\mathbf{p}^\top \mathbf{Ap}, \text{SUM})$
6. $\quad \mathbf{x} \leftarrow \mathbf{x} + \alpha \mathbf{p}$
7. $\quad \mathbf{r} \leftarrow \mathbf{r} - \alpha \mathbf{Ap}$
8. $\quad \rho_{\text{new}} \leftarrow \text{all\_reduce}(\mathbf{r}^\top \mathbf{r}, \text{SUM})$
9. $\quad \mathbf{p} \leftarrow \mathbf{r} + (\rho_{\text{new}} / \rho)\mathbf{p}$
10. $\quad \rho \leftarrow \rho_{\text{new}}$
11. **end while**
12. **Return x**

3.3 Adjoint Linear Solver

We implement Equations (3) and (4) as a custom `torch.autograd.Function`:

```python
class SparseLinearSolve(Function):
    @staticmethod
    def forward(ctx, val, row, col, shape, b):
        x = solve_sparse(val, row, col, shape, b)
        ctx.save_for_backward(val, row, col, x)
        return x

    @staticmethod
    def backward(ctx, grad_x):
        val, row, col, x = ctx.saved_tensors
        # Solve transposed system
        lambda_adj = solve_sparse(val, col, row, shape_T, grad_x)
        # Compute sparse gradients
        grad_val = -lambda_adj[row] * x[col]
        grad_b = lambda_adj
        return grad_val, None, None, None, grad_b
```

Key properties:

- $O(1)$ **graph nodes**: Independent of solver iterations
- **Sparse gradients**: $\partial \mathcal{L} / \partial \text{val}$ has same sparsity as $\mathbf{A}$
- **Backend-agnostic**: Works with any forward solver

3.4 Adjoint Nonlinear Solver

For nonlinear systems $\mathbf{F}(\mathbf{u}, \boldsymbol{\theta}) = \mathbf{0}$, we implement Newton-Raphson with adjoint gradients:

Algorithm 3: Adjoint Nonlinear Solve

Input: Residual $\mathbf{F}(\mathbf{u}, \boldsymbol{\theta})$, initial guess $\mathbf{u}_0$, parameters $\boldsymbol{\theta}$

1. **Forward:** Solve $\mathbf{F}(\mathbf{u}^*, \boldsymbol{\theta}) = \mathbf{0}$ via Newton
2. **for** $k = 0, 1, \ldots$ **do**
3. $\quad \mathbf{J} \leftarrow \partial \mathbf{F} / \partial \mathbf{u}$ *// Jacobian via autograd*
4. $\quad \Delta \mathbf{u} \leftarrow \mathbf{J}^{-1}(-\mathbf{F})$ *// Sparse linear solve*
5. $\quad \mathbf{u} \leftarrow \mathbf{u} + \alpha \Delta \mathbf{u}$ *// Line search*
6. **end for**
7. **Backward:** Given $\partial \mathcal{L} / \partial \mathbf{u}^*$
8. Solve $\mathbf{J}^\top \boldsymbol{\lambda} = \partial \mathcal{L} / \partial \mathbf{u}^*$ *// Adjoint equation*
9. $\partial \mathcal{L} / \partial \boldsymbol{\theta} \leftarrow -\boldsymbol{\lambda}^\top \partial \mathbf{F} / \partial \boldsymbol{\theta}$ *// VJP via autograd*
10. **Return** $\partial \mathcal{L} / \partial \boldsymbol{\theta}$

We support three nonlinear solver methods:

- **Newton-Raphson**: Fast quadratic convergence with Armijo line search
- **Picard iteration**: Simple fixed-point iteration
- **Anderson acceleration**: Memory-efficient acceleration of fixed-point methods

3.4.1 Jacobian-Free Newton-Krylov

When explicit Jacobian construction is expensive, we use Jacobian-free Newton-Krylov (JFNK):

$$\mathbf{J}\mathbf{v} \approx \frac{\mathbf{F}(\mathbf{u} + \epsilon \mathbf{v}) - \mathbf{F}(\mathbf{u})}{\epsilon} \tag{7}$$

In PyTorch, this is computed exactly via `torch.autograd.grad`:

```
def jacobian_vector_product(u, v, F, params):
    u.requires_grad_(True)
    F_u = residual_fn(u, *params)
    Jv = torch.autograd.grad(F_u, u, grad_outputs=v)[0]
    return Jv
```

The adjoint system $\mathbf{J}^\top \boldsymbol{\lambda} = \mathbf{r}$ is similarly solved using transposed JVPs.

4 Experiments

We benchmark `torch-sla` on 2D Poisson equation discretizations using a 5-point stencil. All experiments use NVIDIA H200 GPUs (140GB HBM3) with CUDA 12.4 and PyTorch 2.2.

4.1 Single-GPU Scalability

Figure 2 shows solve time vs. problem size for different backends.

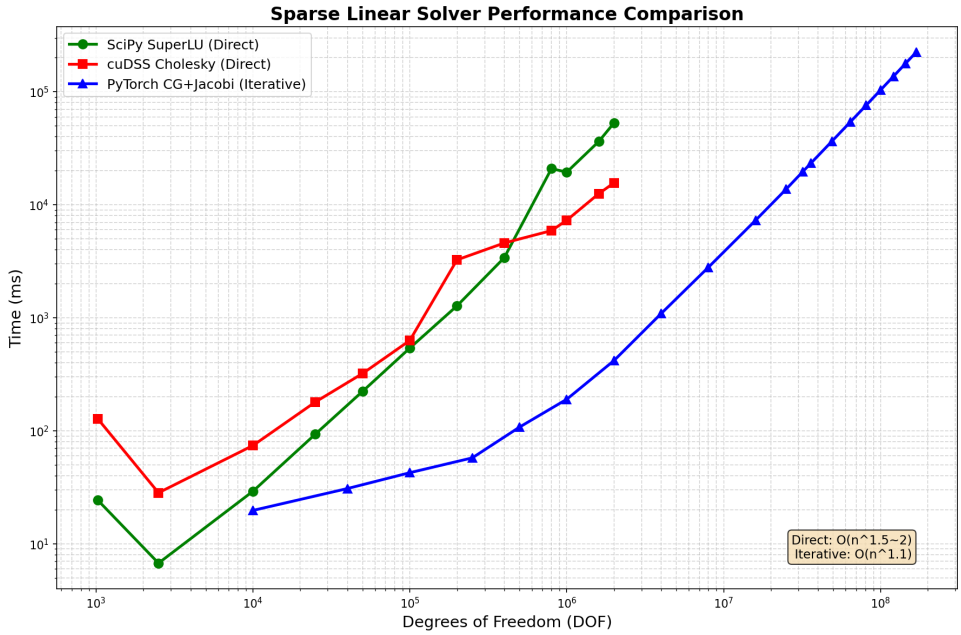

Figure 2: Solver performance comparison. PyTorch CG+Jacobi scales to 169M DOF with near-linear $O(n^{1.1})$ complexity, while direct solvers are limited to ∼2M DOF due to memory.

Table 1: Benchmark results on 2D Poisson (5-point stencil), H200 GPU, float64

DOF	SciPy SuperLU	cuDSS Cholesky	PyTorch CG	Memory	Residual
10K	24 ms	128 ms	20 ms	36 MB	10^{-9}
100K	29 ms	630 ms	43 ms	76 MB	10^{-7}
1M	19.4 s	7.3 s	190 ms	474 MB	10^{-7}
2M	52.9 s	15.6 s	418 ms	916 MB	10^{-7}
16M	OOM	OOM	7.3 s	7.1 GB	10^{-6}
81M	OOM	OOM	75.9 s	35.9 GB	10^{-6}
169M	OOM	OOM	**224 s**	**74.8 GB**	10^{-6}

Key findings:

1. **Iterative solvers scale to 169M+ DOF** with $O(n^{1.1})$ time complexity
2. **Direct solvers limited to ∼2M DOF** due to $O(n^{1.5})$ fill-in memory
3. **Memory efficiency**: PyTorch CG uses 443 bytes/DOF (vs. 144 bytes/DOF theoretical minimum)
4. **Trade-off**: Direct solvers achieve machine precision (10^{-14}), iterative achieves 10^{-6}

4.2 Distributed Computing

Figure 3 shows distributed solve performance on 4 NVIDIA H200 GPUs.

With 4 GPUs each having 140GB memory, the theoretical limit is:

$$\text{Max DOF} \approx \frac{4 \times 140 \text{ GB}}{443 \text{ bytes/DOF}} \approx 1.3 \times 10^9 \text{ DOF} \tag{8}$$

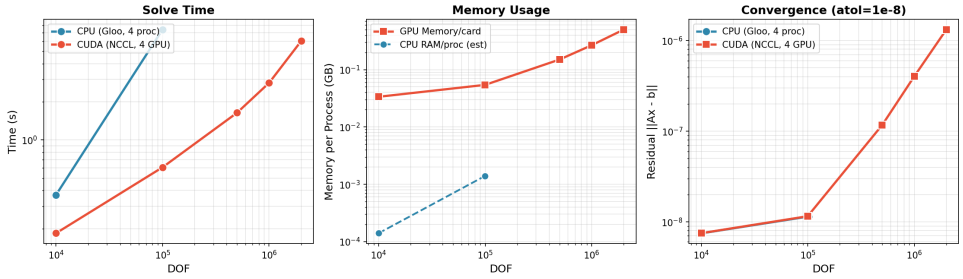

Figure 3: Distributed CG solve on 4 GPUs (NCCL) vs. 4 CPU processes (Gloo). GPU is 12× faster than CPU for 100K DOF.

Table 2: Distributed solve performance (4 GPUs, NCCL backend)

DOF	Time	Residual	Memory/GPU	Speedup vs. CPU
10K	0.18 s	7.5×10^{-9}	0.03 GB	2×
100K	0.61 s	1.2×10^{-8}	0.05 GB	12×
500K	1.64 s	1.2×10^{-7}	0.15 GB	–
1M	2.82 s	4.0×10^{-7}	0.27 GB	–
2M	6.02 s	1.3×10^{-6}	0.50 GB	–

4.3 Gradient Accuracy

We verify gradient correctness using finite difference comparison (Table 3):

Table 3: Gradient verification: adjoint vs. finite difference

Operation	Relative Error	Forward + Backward
Linear solve (sparse A)	$< 10^{-6}$	2 solves
Eigenvalue (k=6)	$< 10^{-5}$	1 forward + 1 LOBPCG
Nonlinear solve (Newton)	$< 10^{-6}$	Newton + 1 adjoint

4.4 Memory Efficiency

Figure 4 compares memory scaling:

5 Related Work

Differentiable Linear Algebra JAX [Bradbury et al., 2018] provides `jax.scipy.sparse.linalg.cg` with automatic differentiation. However, naive differentiation through iterations creates $O(k)$ graph nodes. Our adjoint approach achieves $O(1)$ nodes.

Sparse Libraries SciPy [Virtanen et al., 2020] and Intel MKL provide efficient sparse solvers but lack gradient support. NVIDIA cuSPARSE and cuDSS [NVIDIA Corporation, 2024] provide GPU acceleration. We wrap these as differentiable backends.

Implicit Differentiation Blondel et al. [2022] formalize implicit differentiation for optimization layers. Our nonlinear solver implements this for general $F(u, \theta) = 0$.

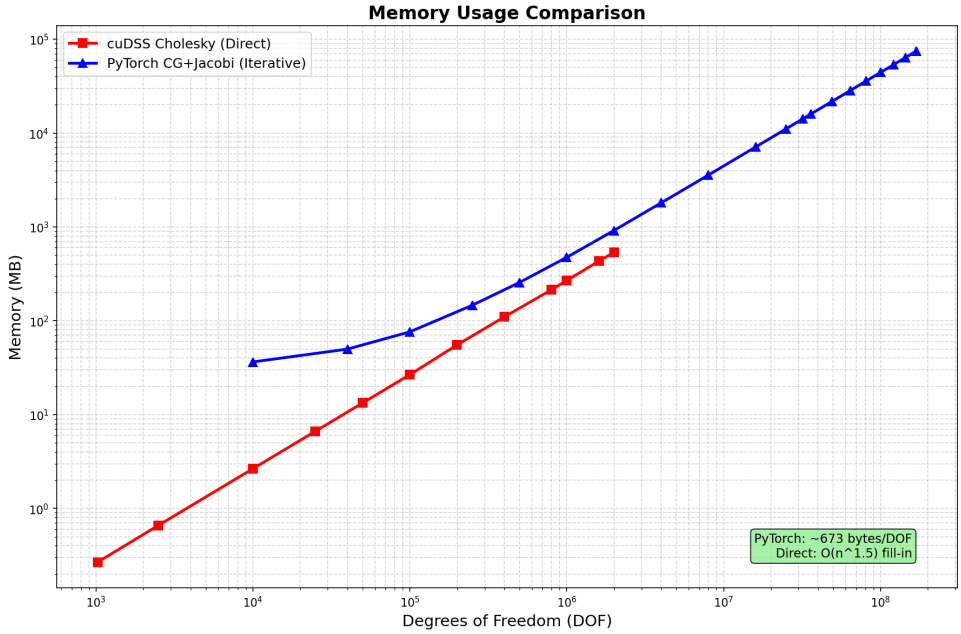

Figure 4: Memory usage comparison. Direct solvers show $O(n^{1.5})$ growth due to fill-in, while iterative CG shows $O(n)$ growth.

Domain Decomposition PETSc [Balay et al., 2019], Trilinos [Heroux et al., 2005], and OpenFOAM [Jasak et al., 2009] implement distributed sparse linear algebra. We bring these concepts to PyTorch with automatic differentiation.

6 Conclusion

We presented `torch-sla`, a differentiable sparse linear algebra library for PyTorch with two key innovations:

1. **Sparse Tensor Parallel**: Distributed sparse matrices with halo exchange, enabling multi-GPU computations for problems with 100M+ DOF.
2. **Adjoint Solvers**: Memory-efficient gradient computation for linear and nonlinear sparse solvers with $O(1)$ graph nodes.

Benchmarks demonstrate:

- Scaling to 169M DOF on single GPU with PyTorch CG
- 12× speedup on 4 GPUs vs. 4 CPU processes
- Near-linear $O(n^{1.1})$ time complexity for iterative solvers
- Correct gradients verified against finite differences

`torch-sla` enables differentiable physics simulations at scale, bridging the gap between classical scientific computing and modern deep learning.

Availability The library is open-source under MIT license: `https://github.com/walkerchi/torch-sla`

Install via: `pip install torch-sla`

References

Satish Balay, Shrirang Abhyankar, Mark F Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D Gropp, et al. Petsc users manual. Technical report, Argonne National Laboratory, 2019.

Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *Advances in Neural Information Processing Systems*, 35:5230–5242, 2022.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic, et al. Openfoam: Open source cfd in research and industry. *International Journal of Naval Architecture and Ocean Engineering*, 1(2):89–94, 2009.

NVIDIA Corporation. cuDSS: NVIDIA CUDA direct sparse solver library. `https://developer.nvidia.com/cudss`, 2024.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, 2020.