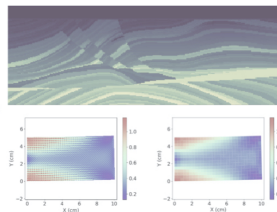
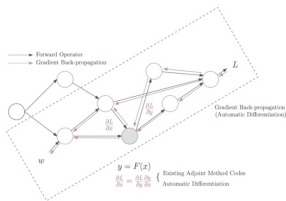
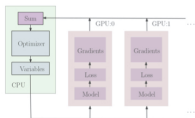
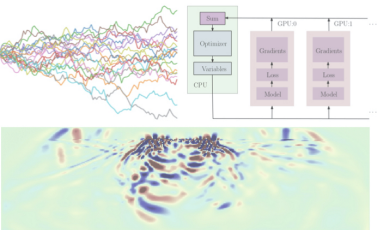


ADCME MPI: Distributed Machine Learning for Computational Engineering

Kailai Xu and Eric Darve

<https://github.com/kailaix/ADCME.jl>



- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 Distributed Computing for Computational Engineering

Forward Problem



Inverse Problem



Inverse Modeling

We can formulate inverse modeling as a PDE-constrained optimization problem

$$\min_{\theta} L_h(u_h) \quad \text{s.t.} \quad F_h(\theta, u_h) = 0$$

- The **loss function** L_h measures the discrepancy between the prediction u_h and the observation u_{obs} , e.g., $L_h(u_h) = \|u_h - u_{\text{obs}}\|_2^2$.
- θ is the **model parameter** to be calibrated.
- The **physics constraints** $F_h(\theta, u_h) = 0$ are described by a system of partial differential equations or differential algebraic equations (DAEs); e.g.,

$$F_h(\theta, u_h) = A(\theta)u_h - f_h = 0$$

Function Inverse Problem

$$\min_{\boldsymbol{f}} L_h(u_h) \quad \text{s.t.} \quad F_h(\boldsymbol{f}, u_h) = 0$$

What if the unknown is a **function** instead of a set of parameters?

- Koopman operator in dynamical systems.
- Constitutive relations in solid mechanics.
- Turbulent closure relations in fluid mechanics.
- ...

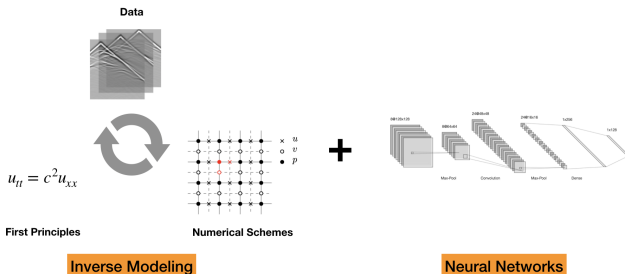
The candidate solution space is **infinite dimensional**.

Machine Learning for Computational Engineering

$$\min_{\theta} L_h(u_h) \quad \text{s.t.} \quad \boxed{F_h(\textcolor{red}{NN}_{\theta}, u_h) = 0} \leftarrow \text{Solved numerically}$$

- ① Use a deep neural network to approximate the (high dimensional) unknown function;
- ② Solve u_h from the physical constraint using a numerical PDE solver;
- ③ Apply an unconstrained optimizer to the reduced problem

$$\min_{\theta} L_h(\textcolor{red}{u}_h(\theta))$$

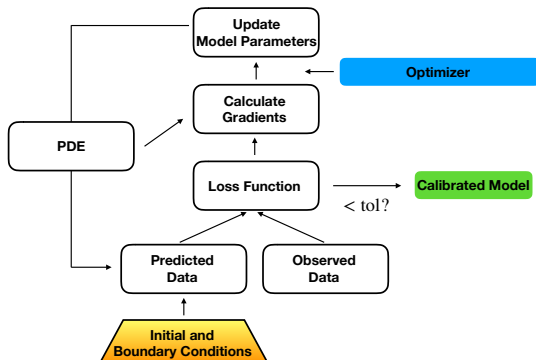


Gradient Based Optimization

$$\min_{\theta} L_h(u_h) \quad \text{s.t.} \quad F_h(\theta, u_h) = 0 \quad \Leftrightarrow \quad \min_{\theta} L_h(u_h(\theta))$$

- We can now apply a gradient-based optimization method if we can calculate a descent direction g^k

$$\theta^{k+1} \leftarrow \theta^k - \alpha g^k$$



Outline

- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 Distributed Computing for Computational Engineering

Automatic Differentiation

The fact that bridges the **technical** gap between machine learning and inverse modeling:

- Deep learning (and many other machine learning techniques) and numerical schemes share the same computational model: composition of individual operators.

Mathematical Fact

Back-propagation



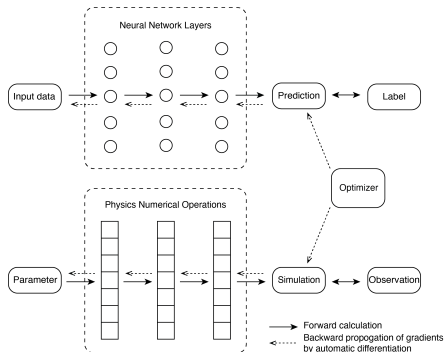
Reverse-mode

Automatic Differentiation



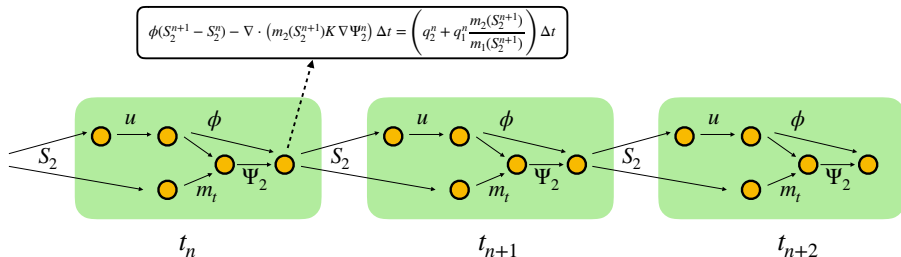
Discrete

Adjoint-State Method

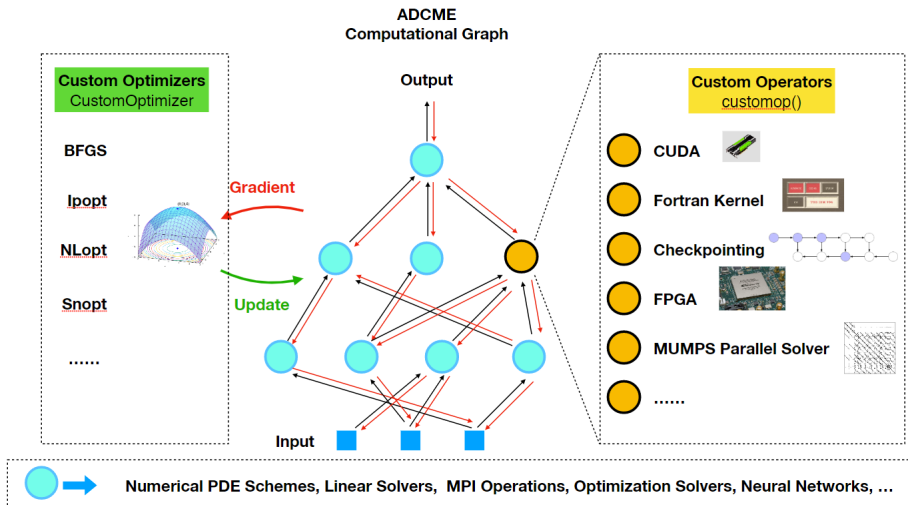


Computational Graph for Numerical Schemes

- To leverage automatic differentiation for inverse modeling, we need to express the numerical schemes in the “AD language”: computational graph.
- No matter how complicated a numerical scheme is, it can be decomposed into a collection of operators that are interlinked via state variable dependencies.



ADCME: Computational-Graph-based Numerical Simulation



How ADCME works

- ADCME translates your **high level** numerical simulation codes to computational graph and then the computations are delegated to a heterogeneous task-based parallel computing environment through TensorFlow runtime.

$$\begin{aligned}\operatorname{div} \sigma(u) &= f(x) & x \in \Omega \\ \sigma(u) &= C \varepsilon(u) \\ u(x) &= u_0(x) & x \in \Gamma_u \\ \sigma(x)n(x) &= t(x) & x \in \Gamma_n\end{aligned}$$

```
mmesh = Mesh(50, 50, 1/50, degree=2)

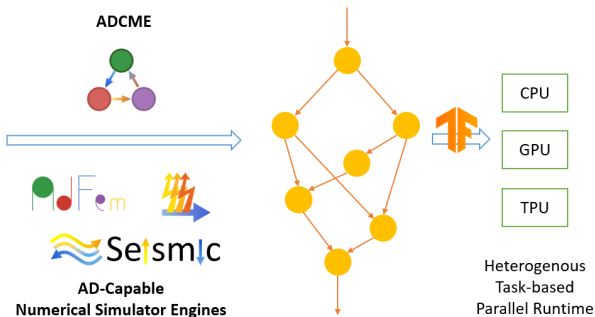
left = bcnode((x,y)->x(1e-5), mmesh)
right = bcnode((x1,y1,x2,y2)->(x1>0.049-1e-5) && (x2>0.049-1e-5), mmesh)

t1 = eval_f_on_boundary_edge((x,y)->1.0e-4, right, mmesh)
t2 = eval_f_on_boundary_edge((x,y)->0.0, right, mmesh)
rhs = compute_fem_traction_term(t1, t2, right, mmesh)

nu = 0.3
x = gauss_nodes(mmesh)
E = abs(fc(x, [20, 20, 20, 1]))>squeeze)
# E = constant(eval_f_on_gauss_pts(f, mmesh))

D = compute_plane_stress_matrix(E, nu*ones(get_n_gauss(mmesh)))
k = compute_fem_stiffness_matrix(D, mmesh)

bdval = [eval_f_on_boundary_node((x,y)->0.0, left, mmesh);
         eval_f_on_boundary_node((x,y)->0.0, left, mmesh)]
DOF = [left:left + mmesh.ndof]
K, rhs = impose_dirichlet_boundary_conditions(K, rhs, DOF, bdval)
u = K\rhs
```

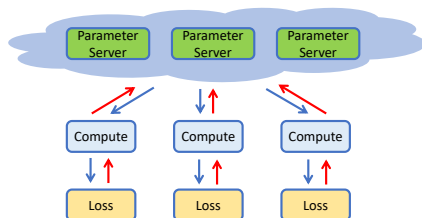
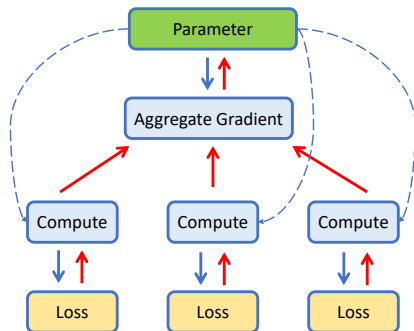


Outline

- 1 Inverse Modeling
- 2 Automatic Differentiation
- 3 Distributed Computing for Computational Engineering

Common Distributed Computing Patterns in DL

$$L(\theta) = \sum_{i=1}^N (NN(x_i; \theta) - y_i)^2$$



Distributed Computing in ML for Computational Engineering

Consider a time-dependent PDE, where the state variable

$$u_k = [u_k^{(1)} \ u_k^{(2)} \ \cdots \ u_k^{(P)}]$$

is stored on P machines. Each time step requires a distributed numerical solver.

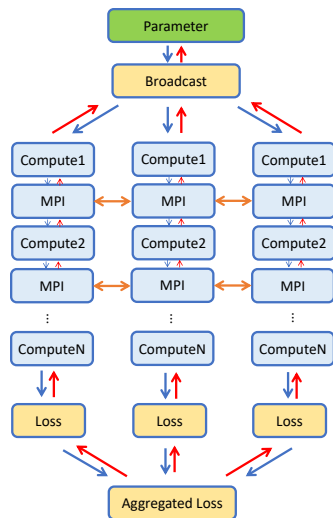
$$\min_{\theta} L(u_n)$$

$$\text{s.t. } A(\theta)u_2 = h(u_1; \theta) + g$$

$$A(\theta)u_3 = h(u_2; \theta) + g$$

$$\vdots$$

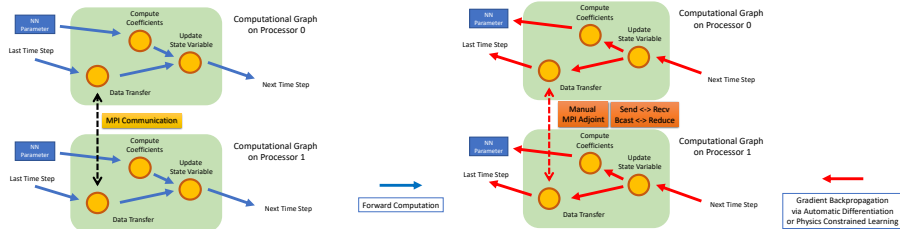
$$A(\theta)u_n = h(u_{n-1}; \theta) + g$$



ADCME-MPI

ADCME-MPI abstracts distributed computing as a node in the computational graph. The ADCME-MPI model is **transparent**.

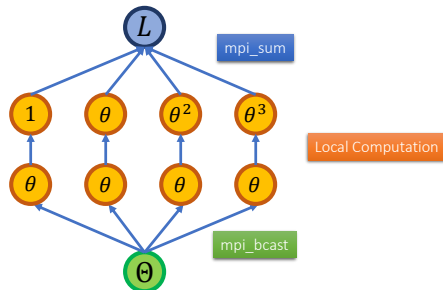
- ADCME takes responsibility for MPI communication and gradient back-propagation across clusters;
- users can adapt their single processor codes to a distributed computing environment with little efforts.



Example

Consider a simple function:

$$L(\theta) = 1 + \theta + \theta^2 + \theta^3$$



```
using ADCME
```

```
mpi_init() # initialize MPI
```

```
theta0 = placeholder(1.0)
```

```
theta = mpi_bcast(theta0)
```

```
l = theta^mpi_rank()
```

```
L = mpi_sum(l)
```

```
g = gradients(L, theta0)
```

```
# initialize a Session
```

```
sess = Session(); init(sess)
```

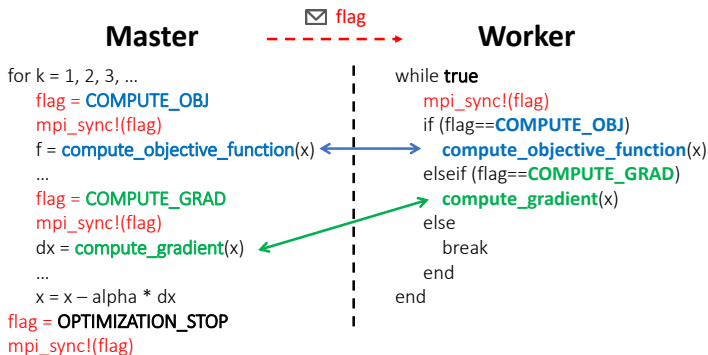
```
L_value = run(sess, L)
```

```
g_value = run(sess, g)
```

```
mpi_finalize() # finalize MPI
```

Distributed Optimization

In the ADCME-MPI, we can convert a serial optimizer to a distributed optimizer by inserting some communication codes:

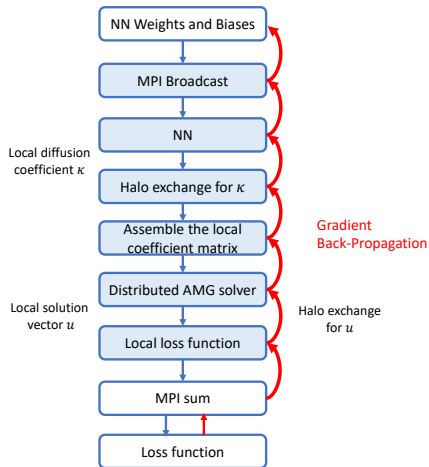
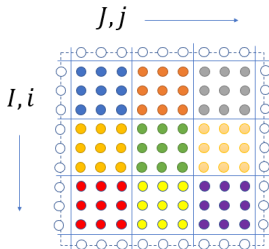


Benchmarks

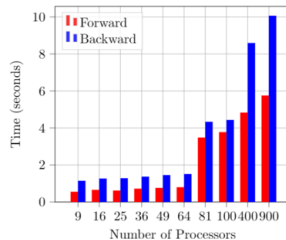
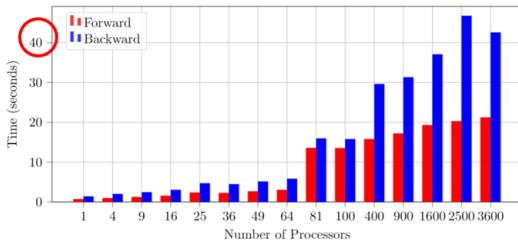
$$\min_{\theta} J(\theta) := \sum_{i \in \mathcal{I}} (u(x_i) - u_i)^2$$

$$\text{s.t. } \nabla \cdot (\textcolor{red}{NN}_{\theta}(\mathbf{x}) \nabla u(\mathbf{x})) = f(\mathbf{x}), \mathbf{x} \in \Omega$$

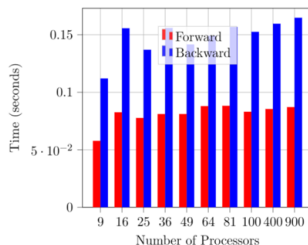
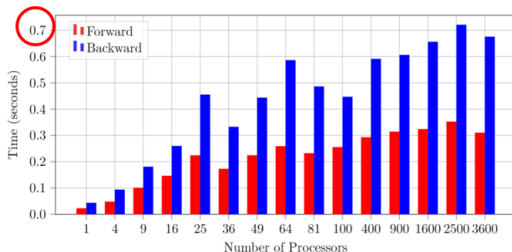
$$u(\mathbf{x}) = 0, \mathbf{x} \in \partial\Omega$$



Total Time



ADCMME Overhead = Total Time – Hyper Time



1 Core per MPI processor

4 Core per MPI processor

For more technical details, benchmarks, or use cases:

- AAI Conference Paper: *ADCME MPI: Distributed Machine Learning for Computational Engineering*
- Full paper: *Distributed Machine Learning for Computational Engineering using MPI*
<https://arxiv.org/pdf/2011.01349.pdf>
- Software documentation:
<https://kailaix.github.io/ADCME.jl/dev/>

A General Approach to Inverse Modeling

