OMDP Commodore Workshop, September 9 – 12, 2024

Exploring Python-based frameworks for geophysical modeling

Tuomas Karna[1], Frank Schlimbach[1], Chuck Yount[1], Matthew Piggott[2]

[1] Intel Corporation

[2] Imperial College, London

Motivation

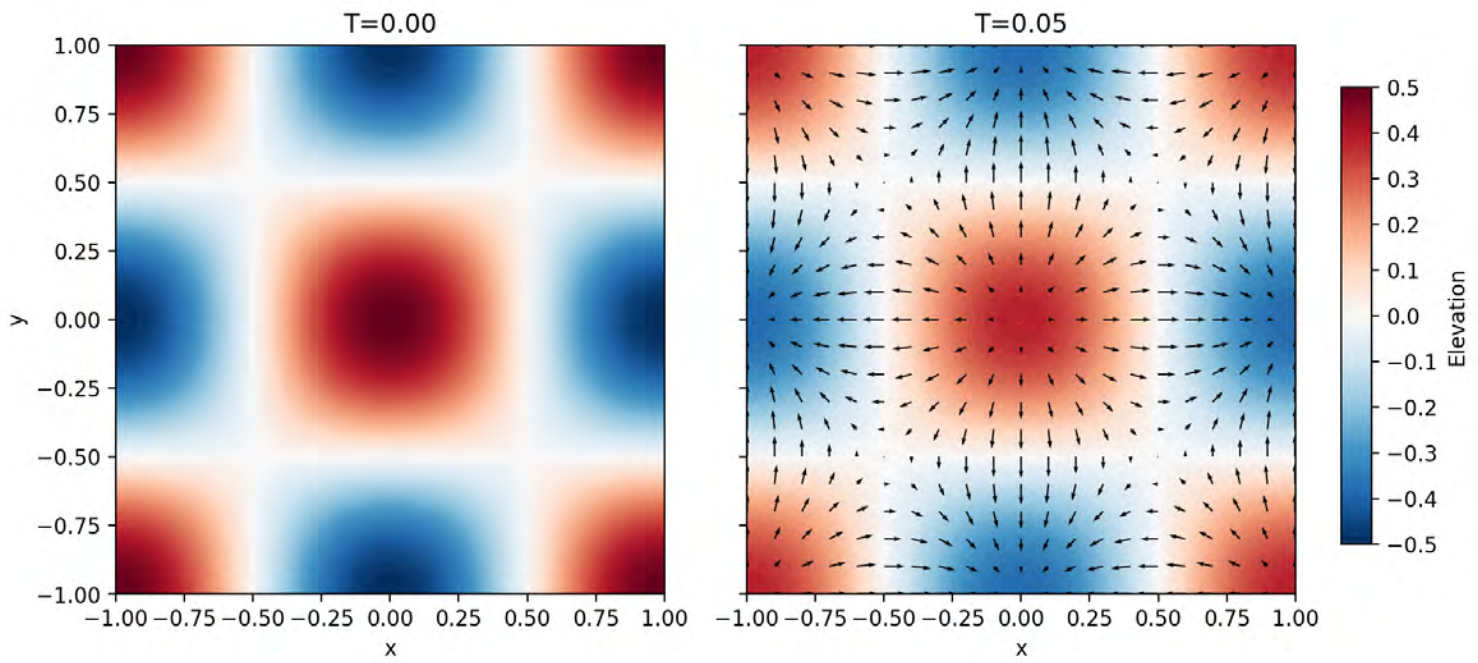
- Existing models (Fortran90) are becoming
  - Hard to port to accelerators
  - Hard to maintain
- Python-based frameworks have become standard in machine learning
  - PyTorch, JAX, …
- Can something similar work for geophysical modeling as well?
- Goals
  - High-level user API: describe the problem without low-level details
  - Flexible, standard API
  - Scalability and performance portability: distributed CPU, GPU, …
  - Adjoint/inverse modeling

Evaluating modeling frameworks

- Suitability for ocean/atmosphere modeling
  - Programming interface
  - Computational performance
  - Portability (CPU/GPU/…)
- We need simple yet representative benchmarks
  - Easy to implement in any language or framework
  - Represent patterns in real simulations, “dwarfs”
  - Correctness test, analytical solution
- **2D structured-grid finite volume benchmarks (C-grid)**
  - Wave equation
  - Shallow water equations

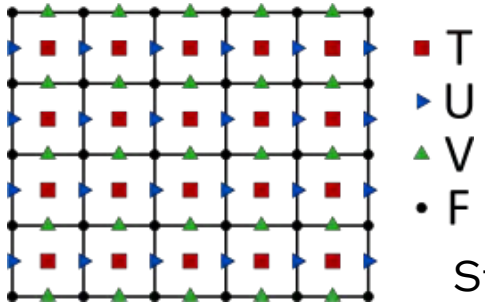
Wave equation benchmark

$$\frac{\partial \eta}{\partial t} + h\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) = 0,$$
$$\frac{\partial u}{\partial t} + g\frac{\partial \eta}{\partial x} = 0,$$
$$\frac{\partial v}{\partial t} + g\frac{\partial \eta}{\partial y} = 0,$$



Analytical solution: 2D standing wave

- Very simple,
  quick to implement on any framework
- Proxy for stencil-like CFD problems
  on a staggered grid
- 3-stage Runge-Kutta solver (SSPRK-33)



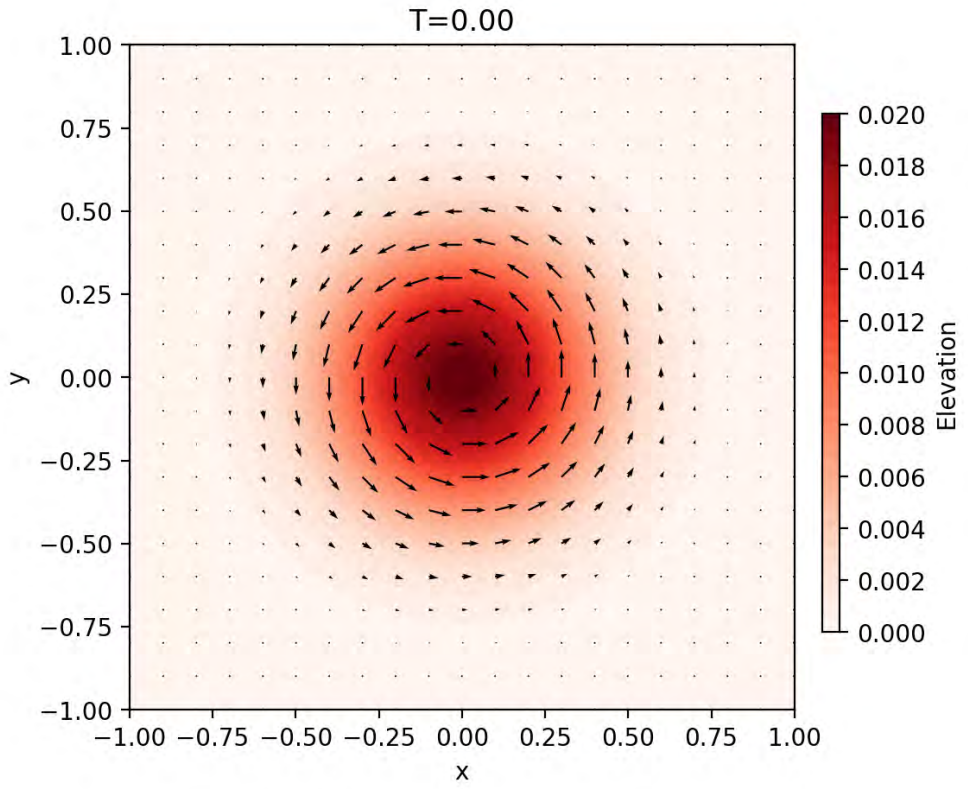
Staggered C-grid

Shallow water benchmark

- Vector-invariant form

$$H = \eta + h$$
$$q = (f + \zeta)/H$$
$$E^{kin} = \frac{1}{2}\sqrt{u^2 + v^2}$$
$$\frac{\partial \eta}{\partial t} + \frac{\partial Hu}{\partial x} + \frac{\partial Hv}{\partial y} = 0,$$
$$\frac{\partial u}{\partial t} + qHv - \frac{\partial E^{kin}}{\partial x} + g\frac{\partial \eta}{\partial x} = 0,$$
$$\frac{\partial v}{\partial t} - qHu - \frac{\partial E^{kin}}{\partial y} + g\frac{\partial \eta}{\partial y} = 0,$$

- More complex, nonlinear
- Energy-conserving formulation
  Arakawa and Hsu (1990)
- 3-stage Runge-Kutta solver (SSPRK-33)



Analytical solution: Geostrophic gyre

- Realistic use cases:
  ocean modeling, flood/tsunami forecasts, …

Implementations

Python frameworks

- **PyTorch, JAX**
  - Existing machine learning frameworks*
  - Numpy-like user interface
  - Just-in-time compiler (JIT)
  - CPU (single node) + GPU support
  - Built-in adjoint modeling capability
- **Sharded Array for Python (Sharpy)**
  - Fully distributed Numpy-like framework
  - LLVM/MLIR-based JIT

Reference implementations

- NumPy (Python)
  - Verification only, not useful for benchmarking (serial execution)
- YASK (Yet Another Stencil Kit)
  - Advanced stencil compiler for CPUs
  - Example of highly optimized code
- **NEMO** (2D shallow water example)
  - Proxy of existing Fortran models
  - Also uses a 3-stage RK solver

* No machine learning methods are used

Python implementation with NumPy-like arrays

$$\frac{\partial u}{\partial t} + g\frac{\partial \eta}{\partial x} = 0,$$
$$\frac{\partial v}{\partial t} + g\frac{\partial \eta}{\partial y} = 0,$$

```
# pressure gradient -g grad(elev)
dudt = -g * (e[1:, :] - e[:-1, :]) / dx
dvdt = -g * (e[:, 1:] - e[:, :-1]) / dy
# update state variable
u1[1:-1, :] = u[1:-1, :] + dt * dudt
v1[:, 1:-1] = v[:, 1:-1] + dt * dvdt
```

- Gradient operators on a structured grid
- Can be expressed with NumPy-like array notation
- Similar to Fortran slicing
- Array formulation used in all Python frameworks
  - JAX arrays are immutable

Just-in-time compilation

- Sequence of operations can be compiled into a function
- Function decorator (JAX, PyTorch)
- Optimizations
  - Loop fusion
  - Temporary elimination
  - Loop tiling, etc.
  - Target-specific optimizations
  - Parallelization (threading, GPU offloading)
- **Performance + portability**

```
@torch.compile
def step(u, v, elev):
    """
    Forward Euler time step
    """
    dudt[1:-1, :] = -g * (elev[1:, :] - elev[:-1, :])/dx
    dvdt[:, 1:-1] = -g * (elev[:, 1:] - elev[:, :-1])/dy
    delevdt[...] = -h * ((u[1:, :] - u[:-1, :])/dx +
                         (v[:, 1:] - v[:, :-1])/dy)
    u[...] += dt * dudt
    v[...] += dt * dvdt
    elev[...] += dt * delevdt
```

Sharded Array for Python (Sharpy)

Repository: ⚠️ Work in progress ⚠️
https://github.com/IntelPython/sharded-array-for-python

- Array API – a subset of NumPy interface
- User writes "serial" NumPy-like code
- Automatically distributed at runtime (MPI)
  - Automatic halos and communications
  - Currently 1D (row) distribution
- Lowers to LLVM/MLIR just-in-time compiler
  - Loop fusion, temporary elimination, vectorization
  - Minimizes MPI communication operations
- GPU support under development
- Compute follows data
  - User defines where arrays are allocated

- CPU

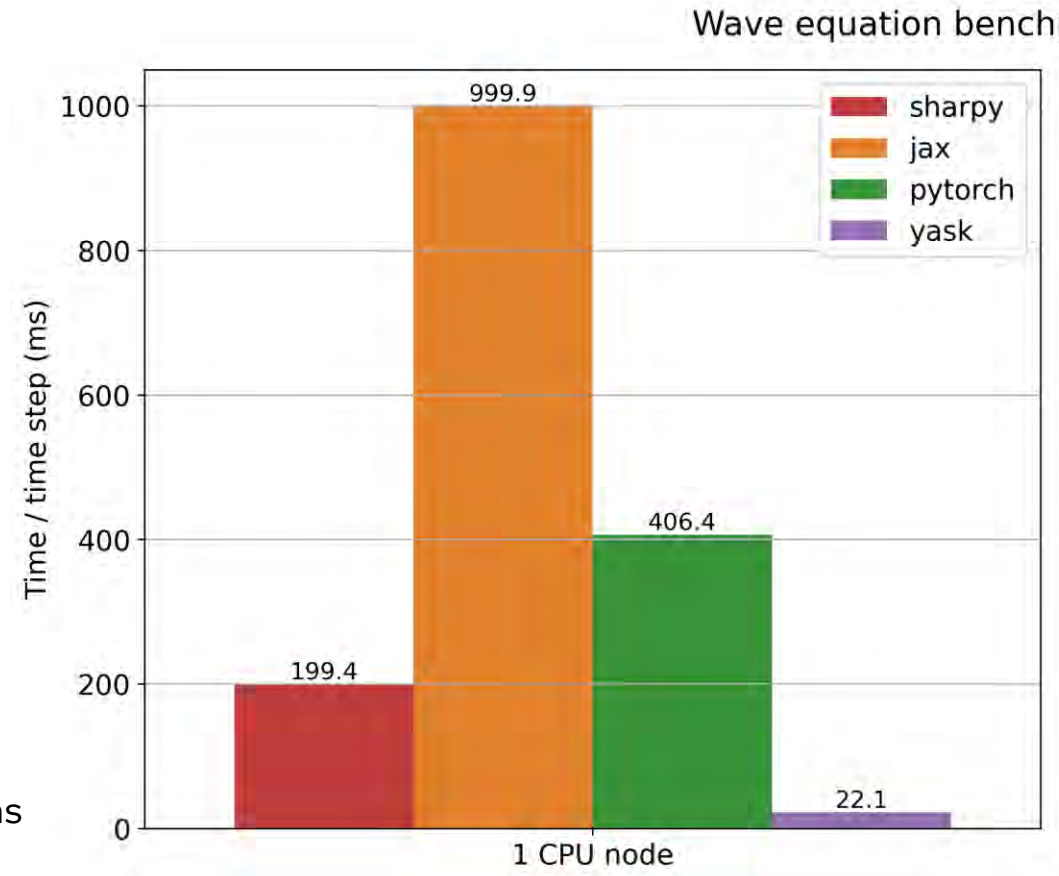
```
import sharpy as sp

a = sp.arange(0, 16, 1)
a[2:10] = a[2:10] + a[3:11] + a[0:8]
```

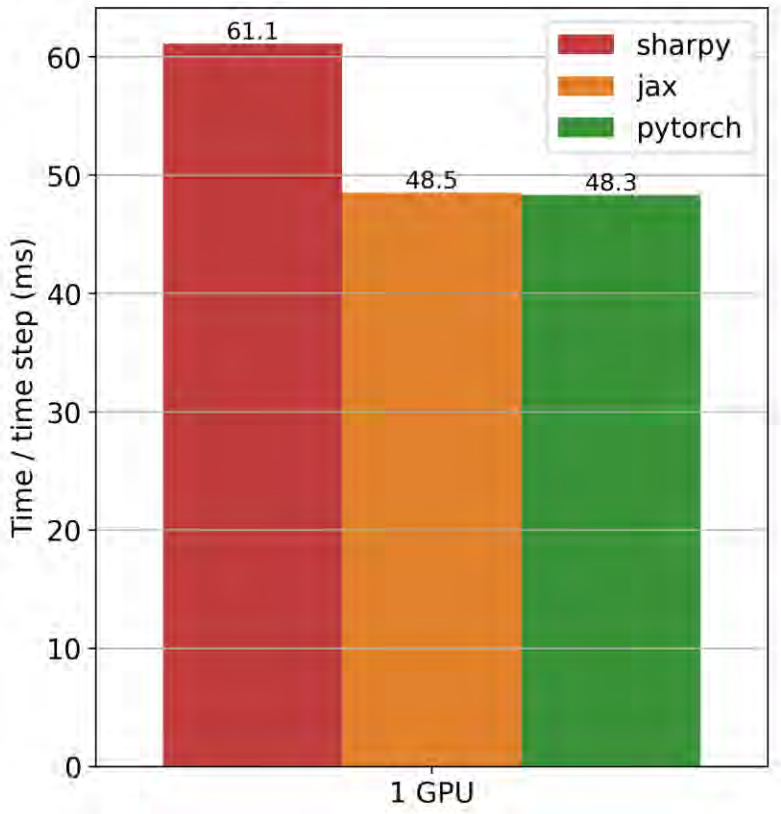
```
# run serially (one process)
python simple-cpu.py

# run on 2 processes
mpirun -n 2 python simple-cpu.py
```

- GPU

```
a = sp.arange(0, 16, 1, device="gpu:0")
a[2:10] = a[2:10] + a[3:11] + a[0:8]
```

Wave equation performance (CPU/GPU)





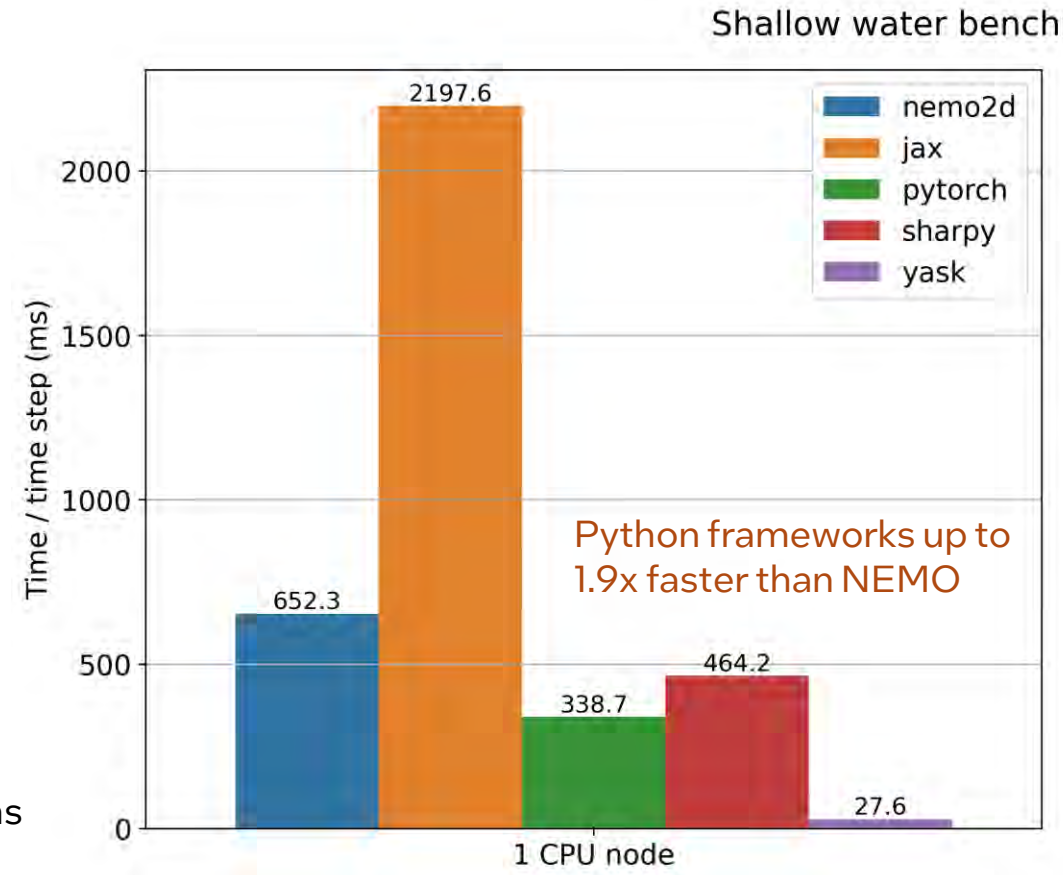
NumPy: 24960 ms

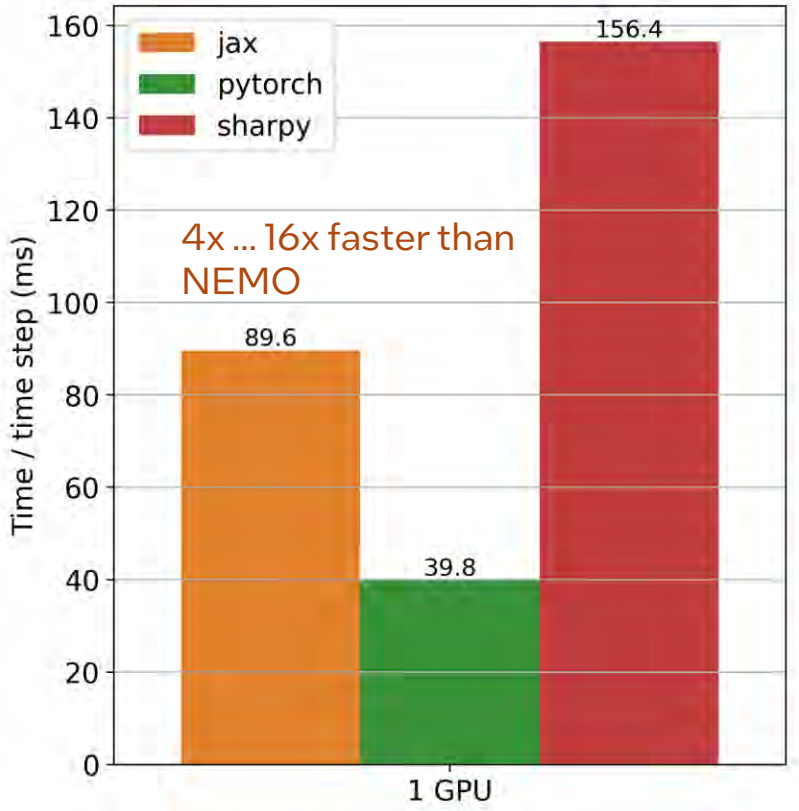
Sharpy: 112 MPI processes
JAX, PyTorch: threading
YASK: 8 MPI processes + 14 openMP threads each

JAX, PyTorch: CUDA backend
Sharpy: MLIR+CUDA

System:
2x Intel® Xeon® Platinum 8480+ CPU, 2x56 cores, 512GB DDR5 RAM, Nvidia H100 Tensor Core GPU

Shallow water equation performance (CPU/GPU)





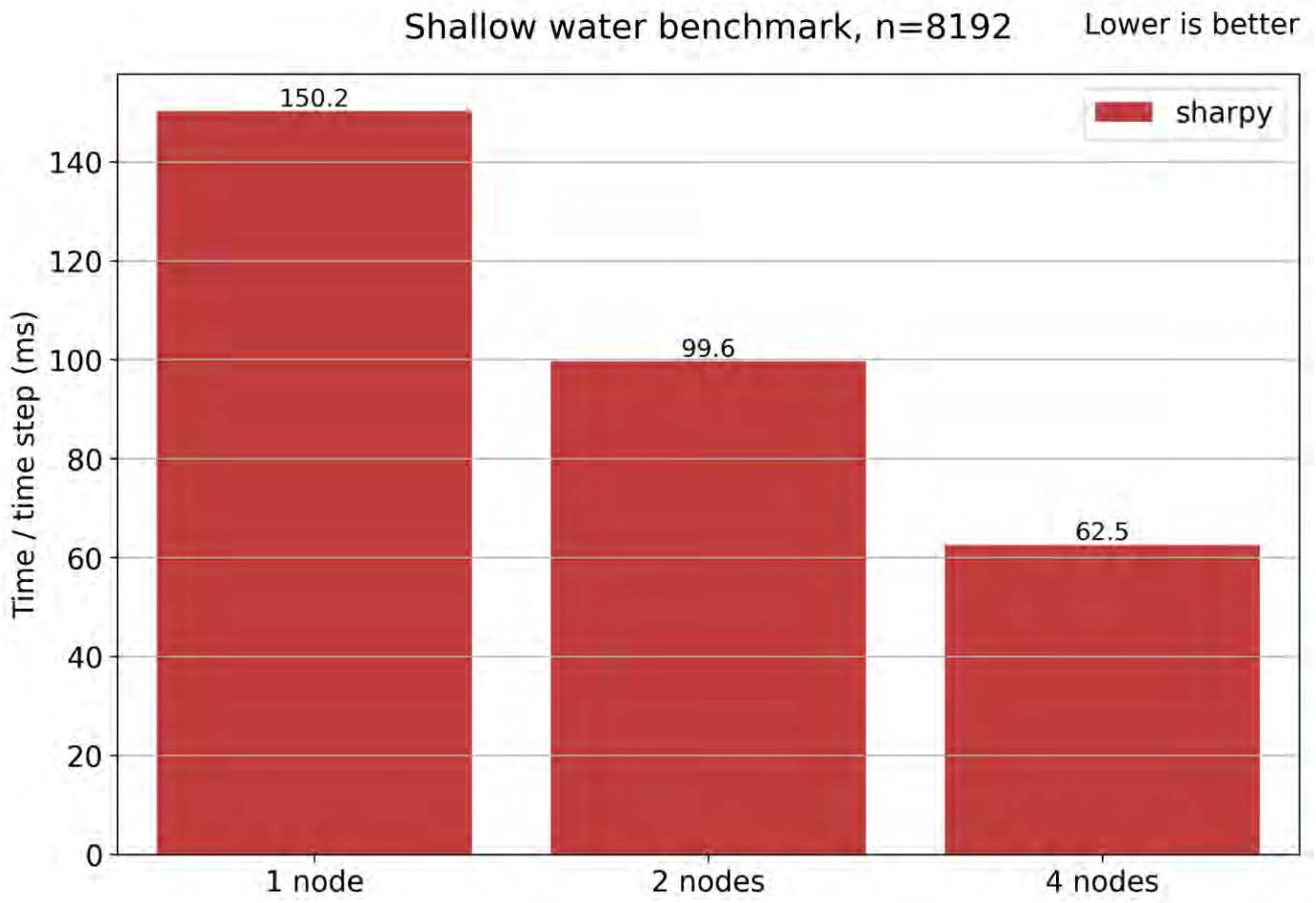
NumPy: 27300 ms

NEMO, Sharpy: 112 MPI processes
JAX, PyTorch: threading
YASK: 8 MPI processes + 14 openMP threads each

JAX, PyTorch: CUDA backend
Sharpy: MLIR+CUDA

System:
2x Intel® Xeon® Platinum 8480+ CPU,
2x56 cores, 512GB DDR5 RAM,
Nvidia H100 Tensor Core GPU

Distributed example (multi-node)



Using 112 MPI processes per node

0.60 scaling efficiency
(row decomposition)

System:
2x Intel® Xeon® CPU Max 9480 Processor CPU,
2x56 cores, 128GB DDR5 RAM

Conclusions

	CPU	GPU	Distributed	Adjoint
JAX	Y	Y	?	Y
PyTorch	Y	Y	?	Y
Sharpy	Y	Y (WIP)	Y	N

- Python frameworks are promising
- Better interface for model developers
- **Performance portability (JIT)**
  - Good performance on CPUs and GPUs
  - Same model code can be run on both environments
- Rigorous benchmarking
  - Correctness test; comparison against state-of-the-art
- Existing solutions: PyTorch and JAX
  - No distributed compute capability yet (?)
- Proof of concept: Sharded Array for Python (sharpy)
  - Automated distribution (MPI) for CPUs and GPUs

Future work

- More complex test cases
- 3D
  - Double gyre?
  - Analytical solution?
- Other potential frameworks?
- Implicit solvers
- Unstructured mesh models

Questions, comments?

email: tuomas.karna@intel.com


intel