

Python Data Processing on Supercomputers for Large Parallel Numerical Simulations

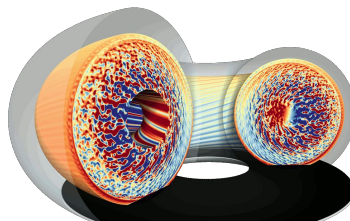
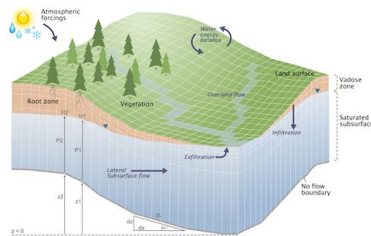
Advised by Bruno Raffin
DataMove team, Inria / Université Grenoble Alpes



Context

Numerical simulations on supercomputers

- Recent supercomputers: Exascale (10^{18} Flops)
- Large scale numerical simulations
 - Gysela: modeling turbulence in tokamak plasmas
 - Parflow: hydrologic model for surface and subsurface flow



- Lot of data produced → analysis required

Data analytics at scale is challenging

- HPC simulations: MPI, low level languages → not suited for data analysis
 - Storing the data produced by the simulation (terabytes per iteration) is costly
- *In process, In situ, In transit* analysis



Doreisa: Dask-on-Ray Enabled In Situ Analytics

- Data analytics tool
- Python + Dask based
- Scalable and efficient

```
def callback(temperature: list[da.Array], pressure: da.Array, *, timestep: int):  
    if len(temperature) == 2:  
        diff = temperature[1] - temperature[0]  
        print("Mean temperature difference:", diff.mean().compute())  
  
        print("Max pressure:", pressure.max().compute())  
  
run_simulation(callback, [  
    ArrayDefinition("temperature", window_size=2),  
    ArrayDefinition("pressure", preprocessing_callback=lambda array: 10 * array),  
])
```

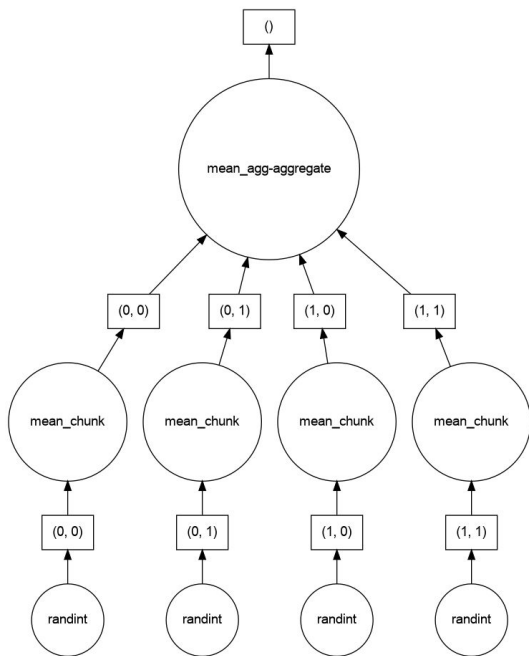


Tools

Dask



- Task graph: DAG defining some computations



- Simple API
 - Dask arrays mimic Numpy arrays

```
import dask.array as da

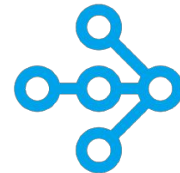
array = da.random.randint(0, 100, size=(100, 100), chunks=(50, 50))
mean = array.mean()
print(mean.compute())
```

✓ 0.0s

49.7582

- Running the computation
 - Scheduler : function taking a Dask graph and computing the value of some keys
 - Threaded, multiprocessing and distributed schedulers
 - Can handle around 4000 tasks per second

Ray



- Lower level framework
- Task (\approx function)
 - Very flexible: a task can create a task
 - Data locality taken into account
- Actor (\approx instance of a class)
 - Allow stateful computations
- Object references
 - Reference to remote data
 - Result of computation or actual data
- Performance aspects
 - Distributed scheduler
 - Distributed reference counting

Dask-on-Ray

- Special scheduler for Dask
- Execute the Dask tasks on a Ray cluster
 - Each node of the Dask task graph is executed in a Ray remote function (ie task)
- Scheduling remains centralized



PDI

- *“The PDI Data Interface”*
 - Coupling C / C++ programs with plugins in charge of using the data
 - Pycall plugin: make data available to Python as Numpy array (without copy)
- Serves as a bridge between C++ / MPI and Python



State of the art

Existing solutions

- **Deisa:** Dask-Enabled In Situ Analytics. PhD work by Amal Gueroudji
 - Extend Dask to work with MPI simulations
 - Computations statically defined at the beginning → number of iterations fixed
 - Rely on ad-hoc patches of the Dask distributed scheduler → not maintainable
 - Doesn't scale well due to the Dask scheduler being centralized
- **Reisa:** Ray-enabled in situ analytics. Master internship by Xico Fernández Lozano
 - Explore the possibilities offered by Ray
 - Not convenient to use (no array API)
 - Not tested at scale (at most 16 nodes)





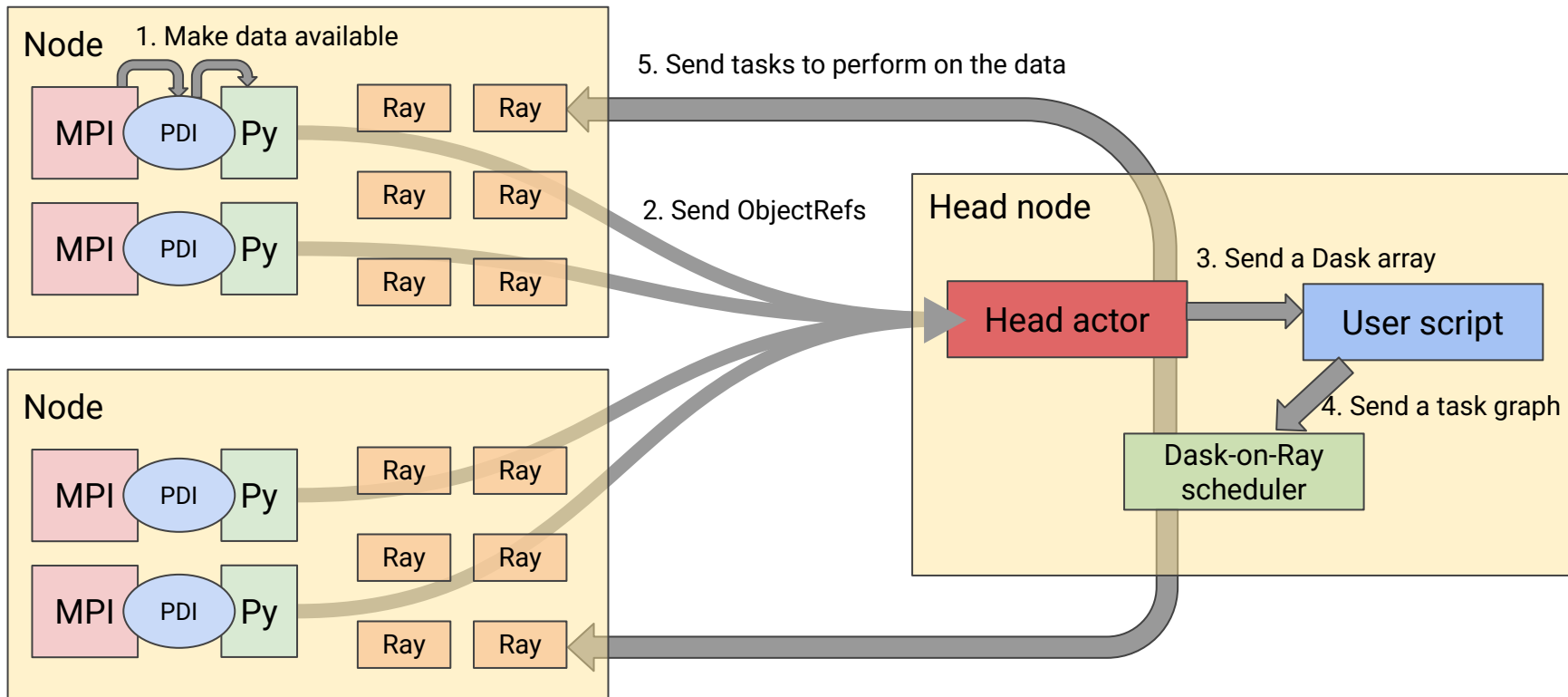
Proof of concept

Main idea

- Use Ray instead of Dask to execute the tasks
 - Faster, more flexible
- Use the Dask array abstraction to provide a simple interface
- Execute the tasks using Dask-on-Ray



Doreisa: proof of concept



Evaluation: Jean Zay

- CPU partition of the Jean Zay supercomputer
- 720 nodes (at most 256 booked at the same time) with:
 - 2 Intel Cascade Lake 6248 processors (2 x 20 cores)
 - 192 Go RAM
- SLURM based



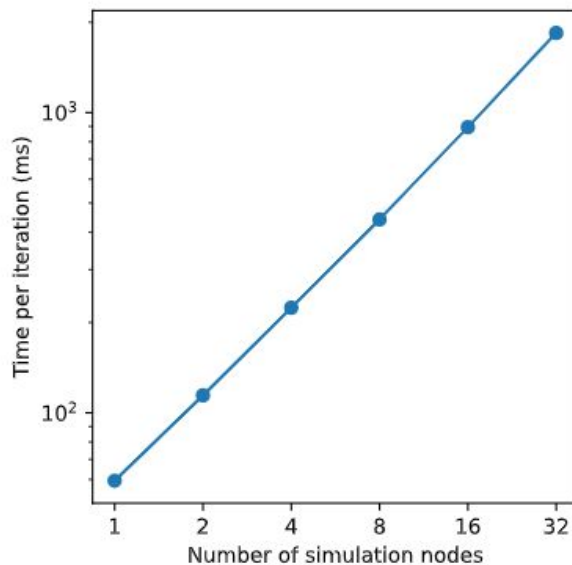
Evaluation protocol

- Executed on Jean Zay (CPU partition, varying number of nodes)
- Ray: 40 worker processes per node
- Array:
 - 40 chunks per node per iteration $\rightarrow 40 \times \text{nbNodes}$ chunks in the array
 - Would correspond to 40 MPI processes on each node (but no simulation in the experiment)
 - Dimension of a chunk: $10 \times 10 \rightarrow$ computation time negligible, focus on Doreisa overhead
 - Weak scaling
- Task: mean of the array



Evaluation

- Still centralized, doesn't scale well enough



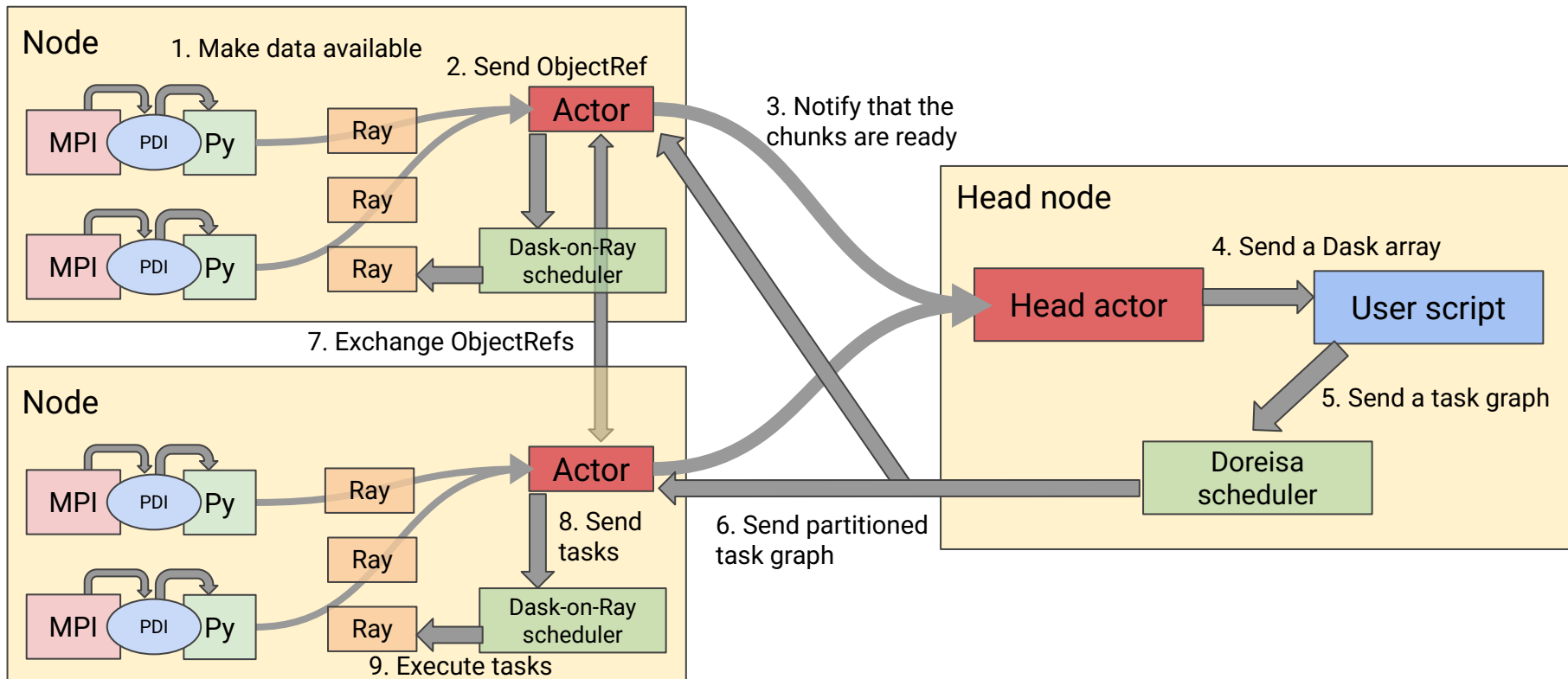
Distributed scheduler

Working at a large scale

- Problem: the head actor is a bottleneck
 - Need to communicate with all the worker processes
 - All the Ray tasks are created by the head node
 - Doesn't take advantage of Ray's distributed scheduler
- Ideas
 - Avoid sending all the ObjectRefs to the head actor
 - Splitting the task graph to have several actors schedule it

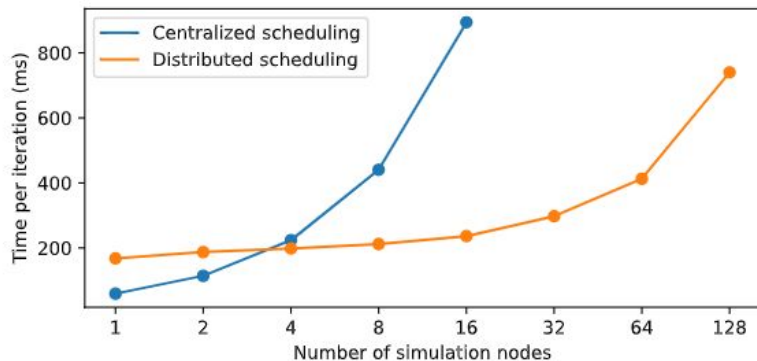


Doreisa: distributed scheduler



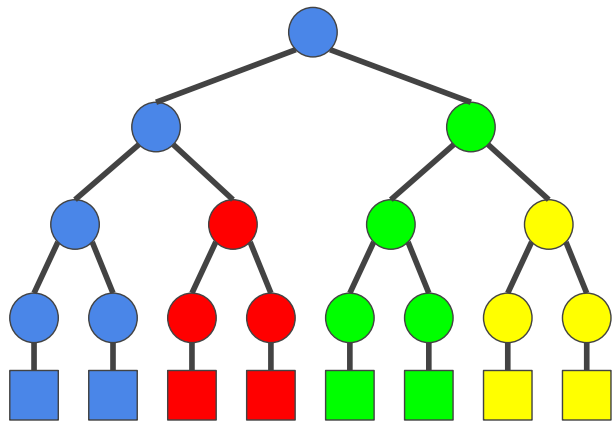
Evaluation

- Same protocol as before
- Works well with up to 32 nodes
 - Doubling the size increases the time by ≈ 20 ms
- New bottleneck appears at 64 nodes
- Issue remains at bigger scale:
 - 255 nodes: 1500 ms / it
- Already great results
 - 256 nodes is the maximum that can be booked at the same time on Jean Zay
 - 2 s / it is not a problem: the simulation is not blocked during this time
 - Much faster than Deisa

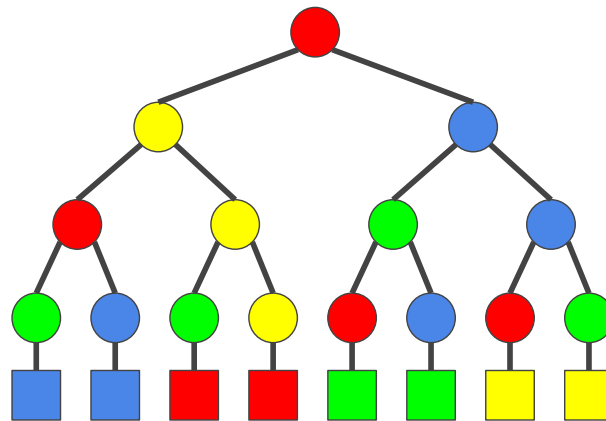


About the partitioning strategy

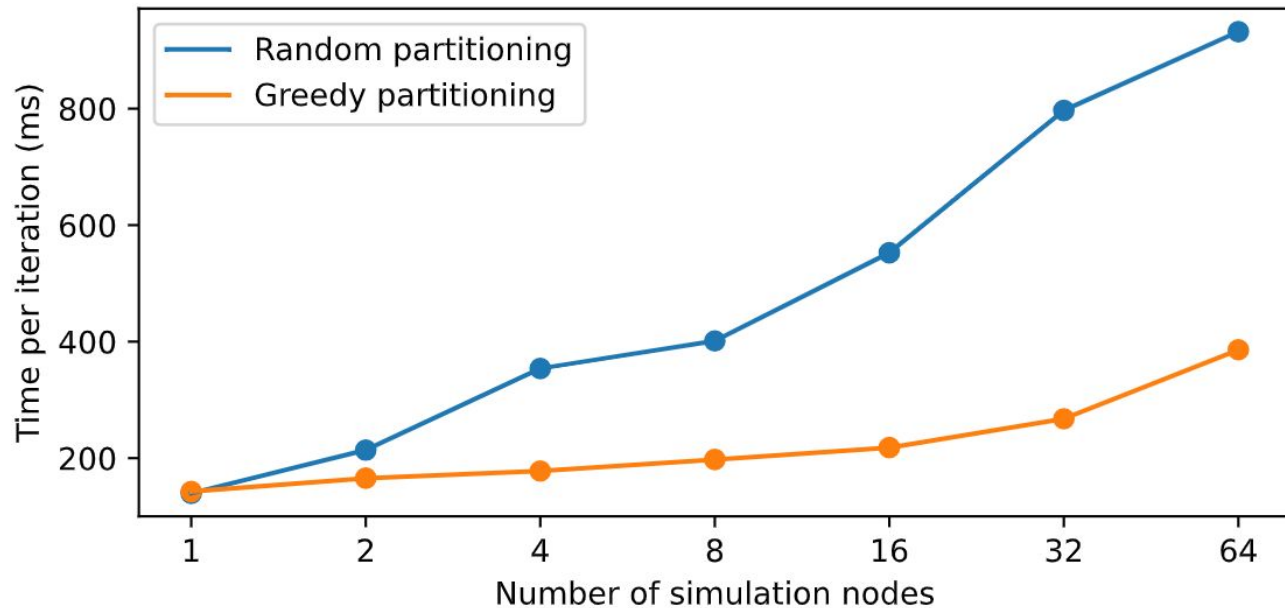
- Greedy partitioning



- Random partitioning

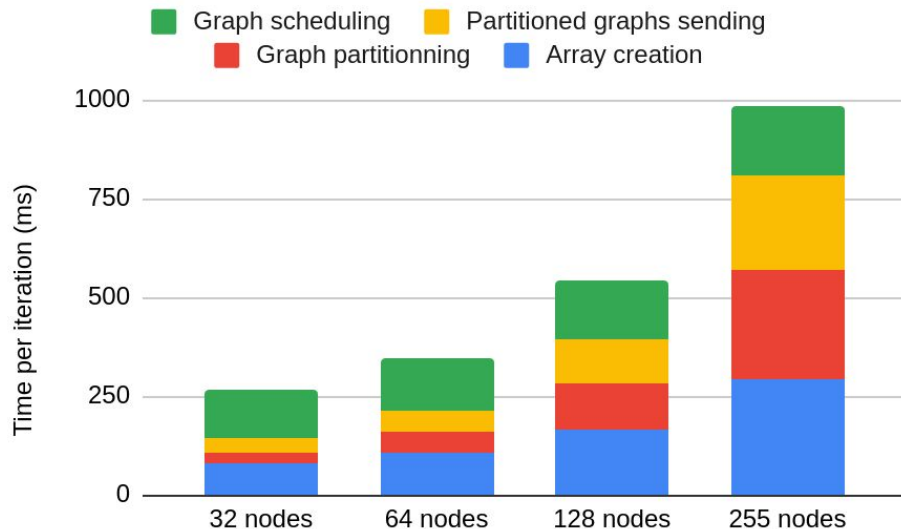
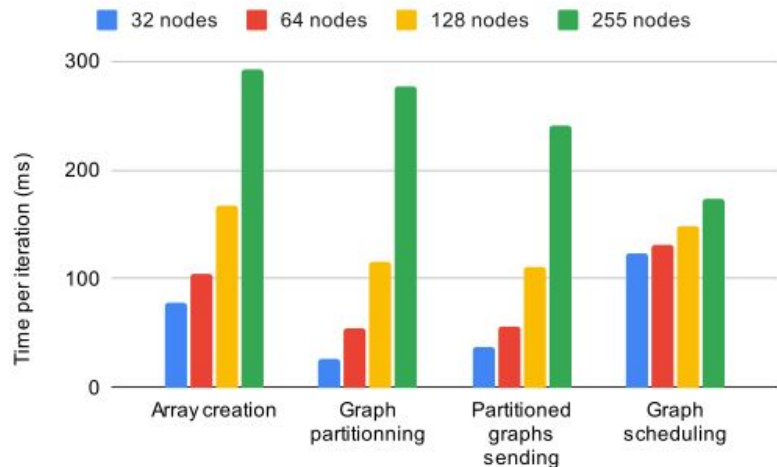


About the partitioning strategy



Finding the new bottleneck

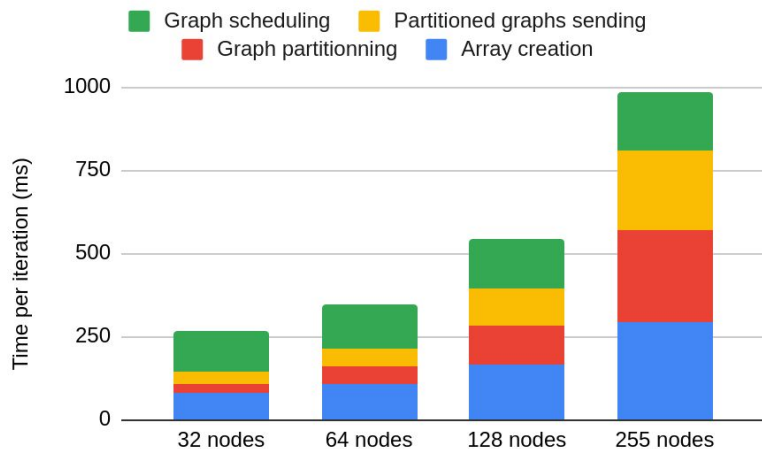
- New bottleneck around 64 nodes
- Problem comes from the centralized part



Asynchronous task graph processing

Idea

- The centralized tasks are expensive
- Perform them ahead of time, before the data is ready → pipelining




User API

- Rely on Dask's persist

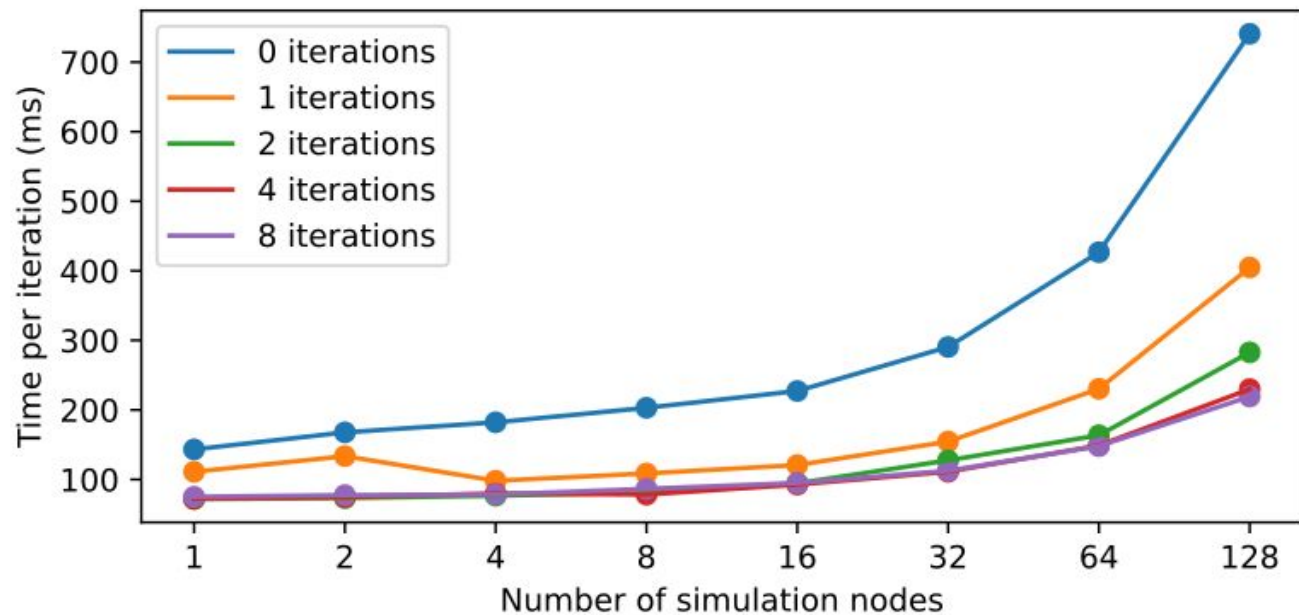
```
def prepare_iteration(array: da.Array, *, timestep: int) -> da.Array:
    # We cannot use compute here since the data is not available yet
    return array.sum().persist()

def simulation_callback(array: da.Array, *, timestep: int, preparation_result: da.Array):
    print(preparation_result.compute())

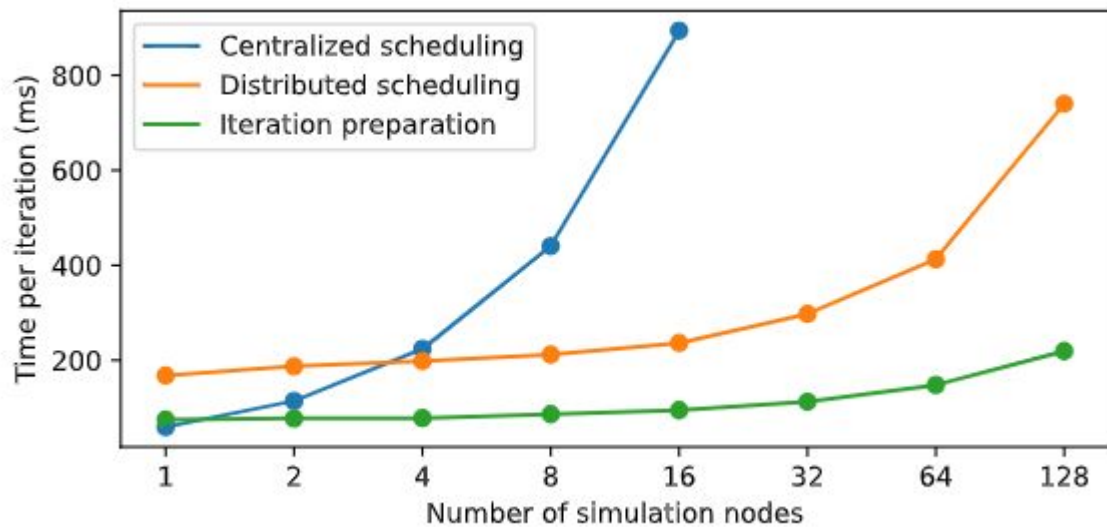
run_simulation(
    simulation_callback,
    [ArrayDefinition("array")],
    max_iterations=NB_ITERATIONS,
    prepare_iteration=prepare_iteration,
    preparation_advance=10,
)
```



Evaluation



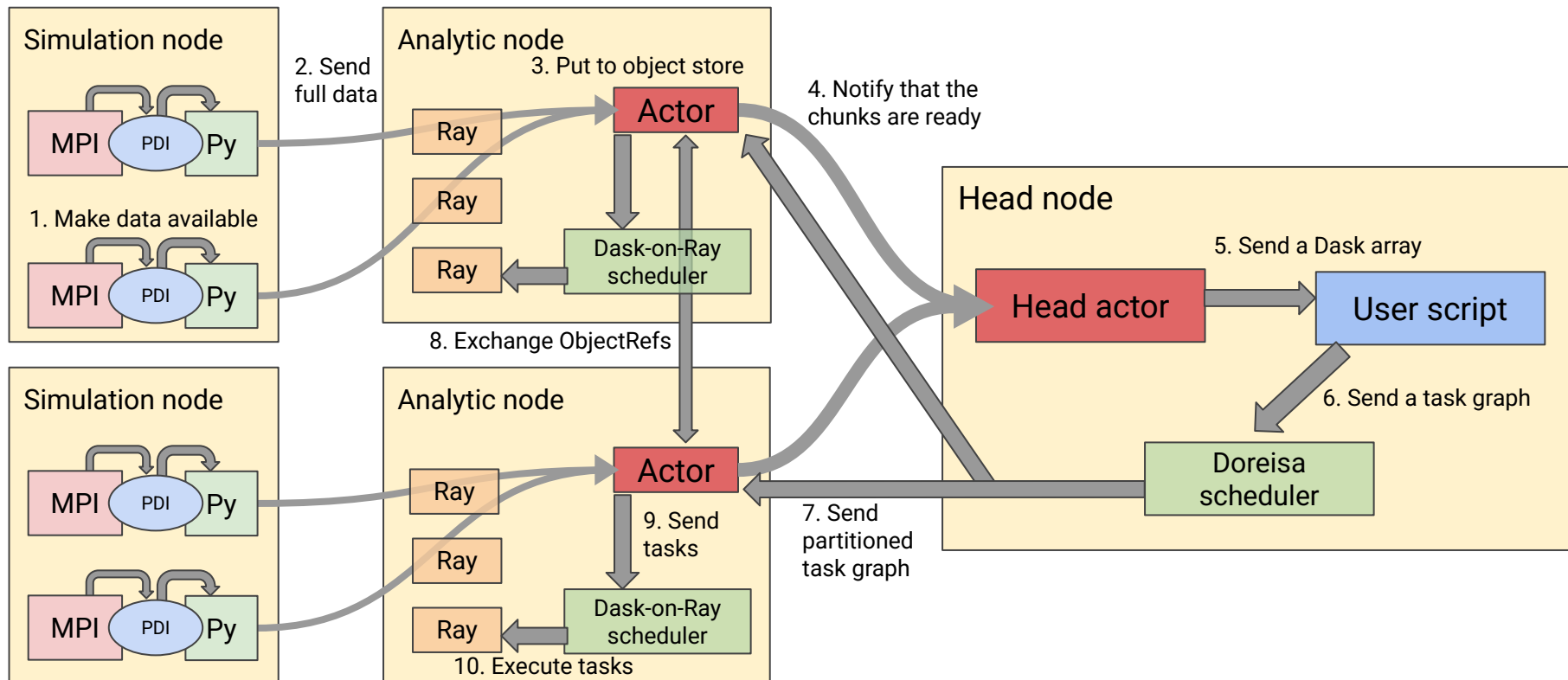
Evaluation





In transit analytics

In transit analytics

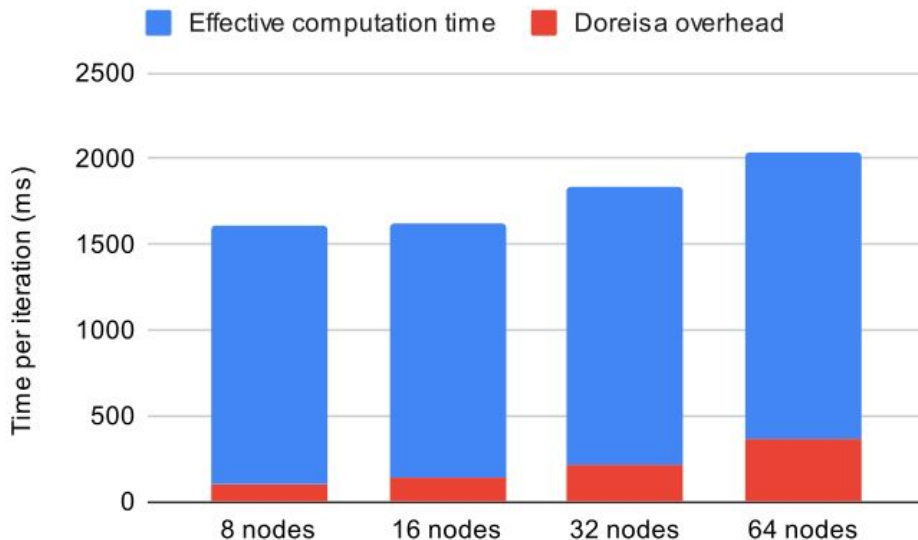


Evaluation

Using bigger chunks of data

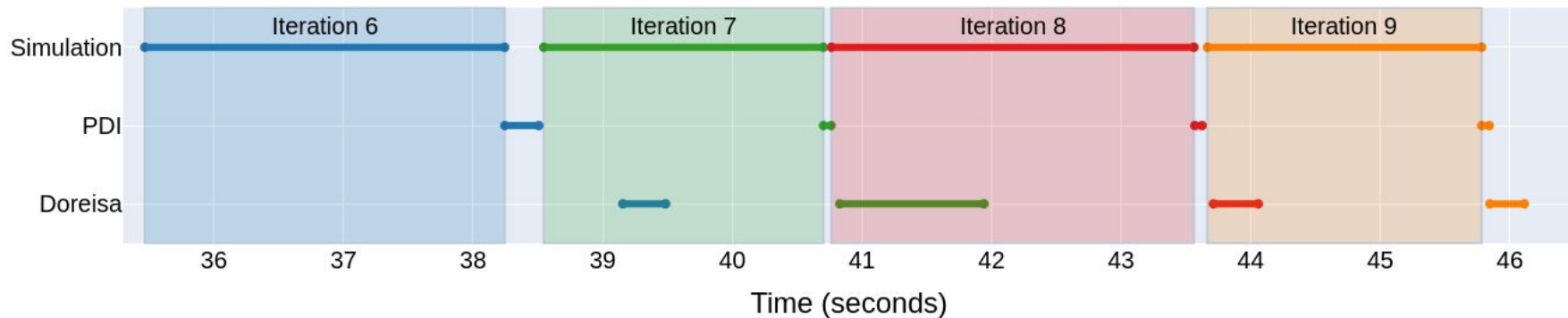
- Array with chunks of dimension 1000 x 1000
- Goal: call $x \mapsto \sin(\sqrt{x+1})$ element wise \rightarrow expensive computation
- Still weak scaling: 40 chunks per node

- Scales well
- Doreisa overhead negligible (even better for the latest version)



Integration with Parflow

- Integration by engineers in the team (simple thanks to PDI)
- Cores shared between the simulation (100) and Doreisa (11)



- Task: mean of an array

Conclusion