The University of Texas at Austin

Parla:
HPC tasks for shared-memory heterogeneous nodes in Python

George Biros, Martin Burtscher, **Mattan Erez**, Milos Gligoric**,** Keshav Pingali, Chris Rossbach,
Jimmy Almgren-Bell, Ian Henrikson, Hochan Lee, Arthur Peters, Jaeyoung Park,
Will Ruys, Sean Stephens, Yineng Yan, Bozhi You

Can Python Do for HPC What It Did for Machine Learning?

Parla: HPC tasks for shared-memory nodes in Python

Rapid development + gradual adoption + heterogeneity

- MPI+'X' where X = Parla
  - MPI has solved the internode scalability problem
  - Per node performance and portability remain challenging
- Programming model: **task orchestration**
  - Sequential program semantics, with tasks and dependences specified by programmer
  - CrossPy for heterogeneous-distributed arrays
- Runtime system
  - Resource-aware task scheduling with automatic data movement
  - Runtime interoperability through Virtual Library Contexts (VLCs)
- Parla kernels
  - Existing libraries (MFEM, NumPy/CuPy, NVIDIA/AMD/Intel, …)
  - Leverage PyKokkos and other efforts (e.g., Numba)
  - Enables incremental adoption

Parla API example: tasks

```
IND = TaskSpace("Independent Tasks")
for i in range(num_gpu):
  s = slice(i * block_size, (i + 1) * block_size)
  @spawn(IND[i], placement=gpu(i))
  def inner_product():
    ai = clone_here(a[s])
    bi = clone_here(b[s])
    partial_sums[i] =  ai @ bi

@spawn(reduce,
       dependencies=IND,
       placement=cpu)
def reduce():
  result = np.sum(partial_sums)
await reduce
```

- Deliberate research-oriented choices with explicit control backed up with (partial) automation
- Naming and TaskSpaces
  - Task instance names for explicit dependencies
    - Future automation planned
  - TaskSpaces for convenient group naming
    - Indexed
    - Sliced
    - Awaited (barriers)
- Task scheduling and resource directive and hints

Parla API example: data movement

```
IND = TaskSpace("Independent Tasks")
for i in range(num_gpu):
  s = slice(i * block_size, (i + 1) * block_size)
  @spawn(IND[i], placement=gpu(i))
  def inner_product():
    ai = clone_here(a[s])
    bi = clone_here(b[s])
    partial_sums[i] =  ai @ bi

@spawn(reduce,
       dependencies=IND,
       placement=cpu)
def reduce():
  result = np.sum(partial_sums)
await result
```

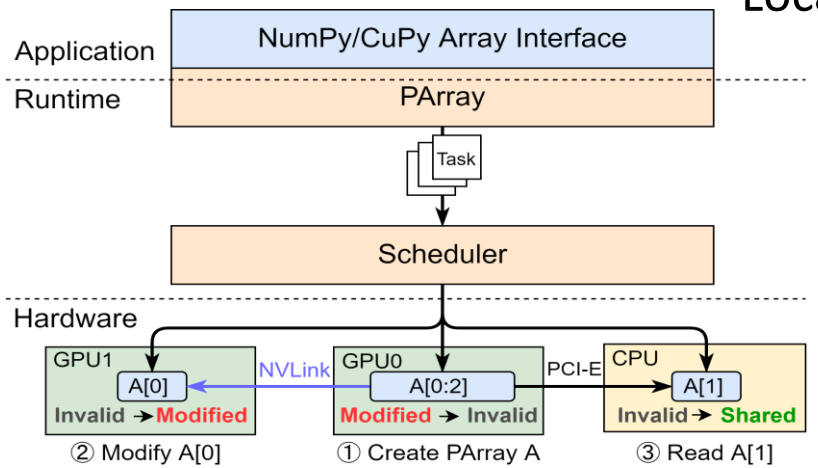
- Deliberate research-oriented choices with explicit control backed up with (partial) automation
- Semi-manual data movement
  - Flexible partially-automated
  - User-specified placement
  - Relative to task location `clone_here(data)`
  - Relative to data location `copy(data1, data2)`

Parla API example: PArrays

```
IND = TaskSpace("Independent Tasks")
for i in range(num_gpu):
  s = slice(i * block_size, (i + 1) * block_size)
  @spawn(IND[i], placement=gpu(i),
         in=[a[s], b[s]], out=[partial_sums[i]])
  def inner_product():
    // ai = clone_here(a[s])
    // bi = clone_here(b[s])
    partial_sums[i] =  a[s] @ b[s]

@spawn(reduce,
       dependencies=IND,
       placement=cpu)
def reduce():
  result = np.sum(partial_sums)
await result
```

- Fully automated data movement
  - PArrays to wrap ndarrays (numpy/cupy, but extensible)
  - Specify ins and outs
  - Parla takes care of the rest
    - Copies and coherence,
    - Scheduled data movement
    - Locality-aware scheduling

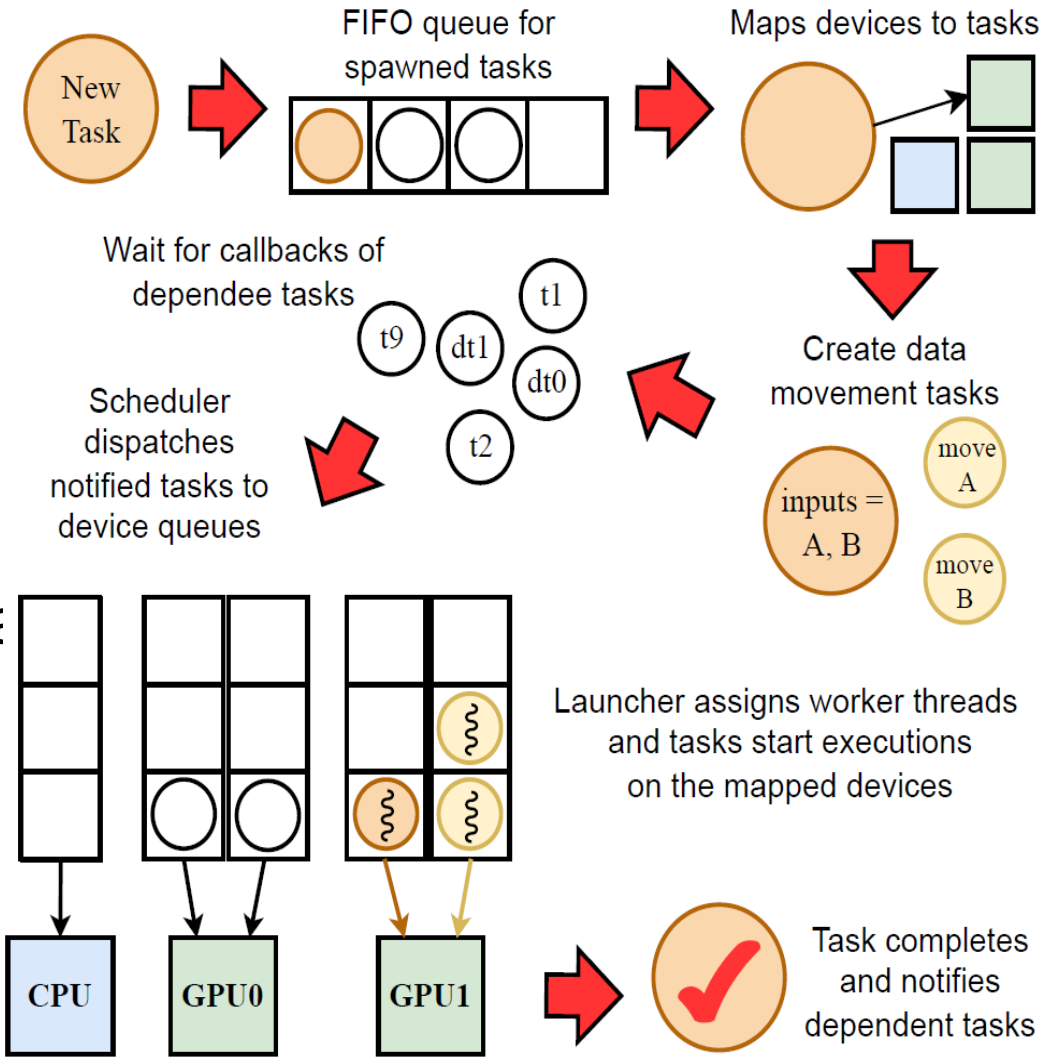


PArrays and CrossPy

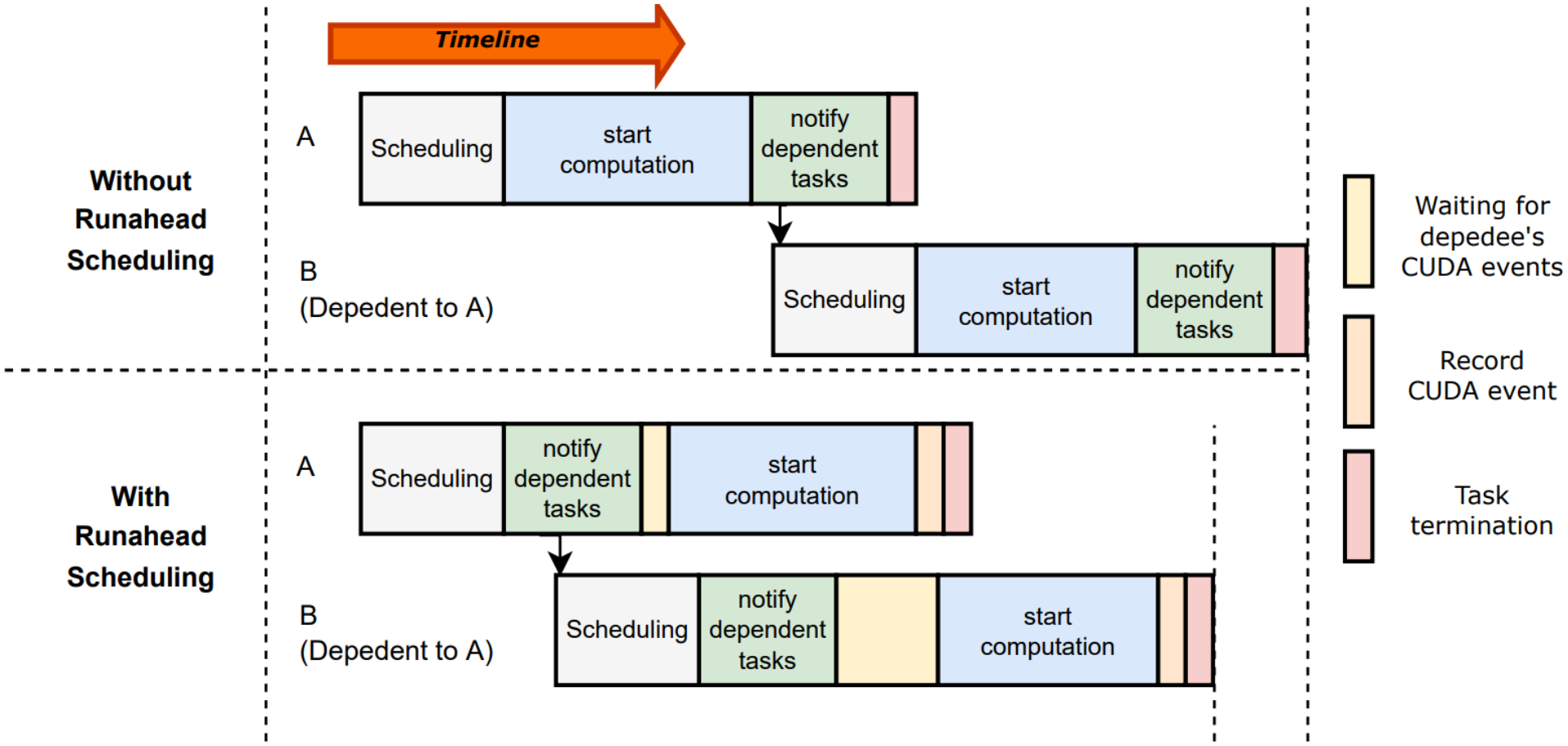
- PArrays track data blocks across devices
  - Parla internal and ideally not directly exposed to programmers
  - Enable automatic data movement and dependence tracking
  - Wrap cupy/numpy arrays (and more)
  - Ideally, automatic wrapping/unwrapping as needed
- CrossPy expresses heterogeneous distributed arrays (across devices)
  - Programmer-facing interface for CPU/GPU array interoperability
  - Parla-facing interface mapped to PArrays

Parla runtime scheduler

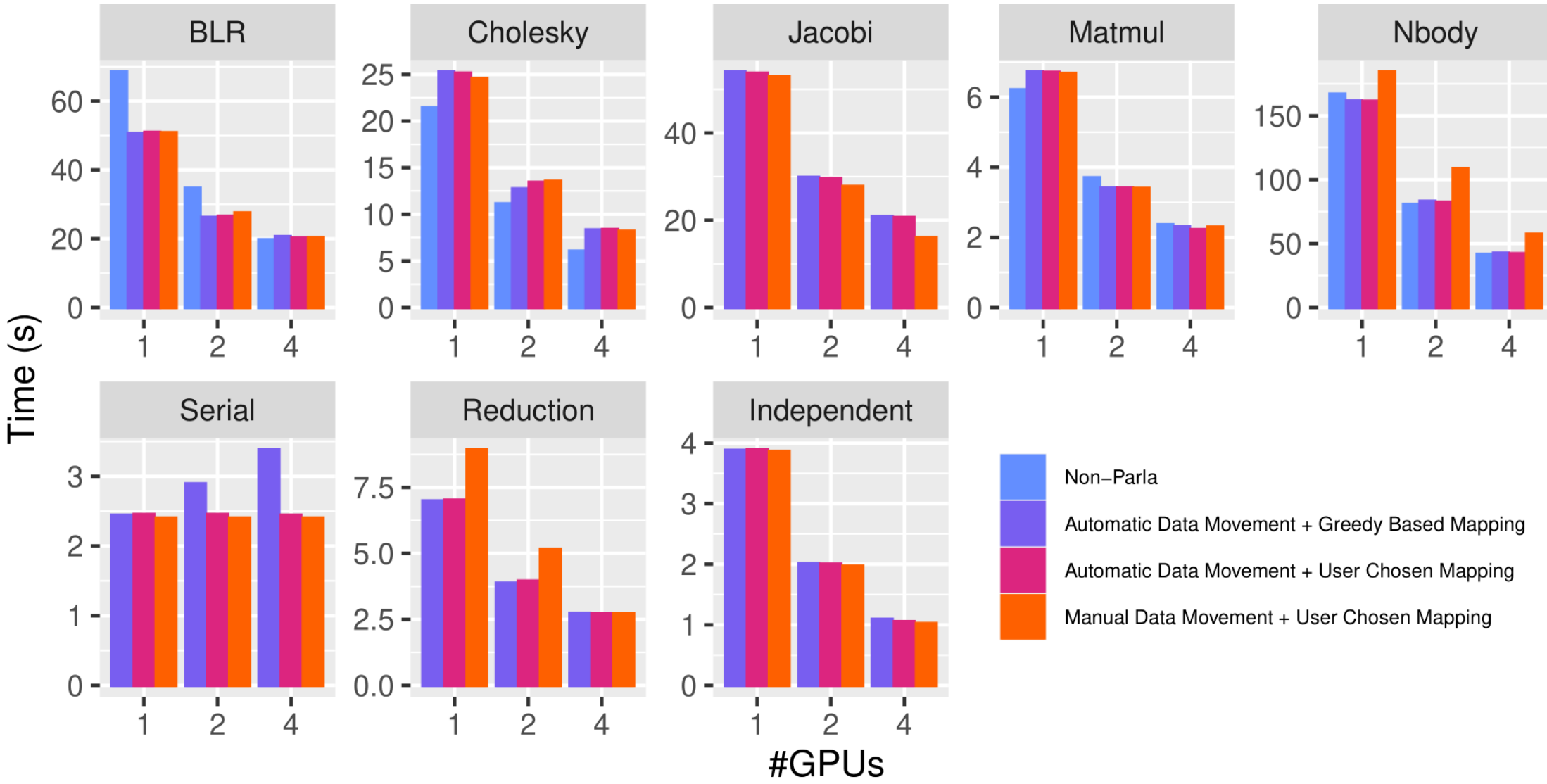
- Designed for “coarse”-grained tasks
  - Kernels and libraries within a task
- Manages execution contexts
  - CPU threads and CUDA streams
- Dynamically schedules tasks and data
  - Dependencies between tasks
  - Placement restrictions and hints
  - Resource usage (memory, acus)
  - Supports multi-device kernels



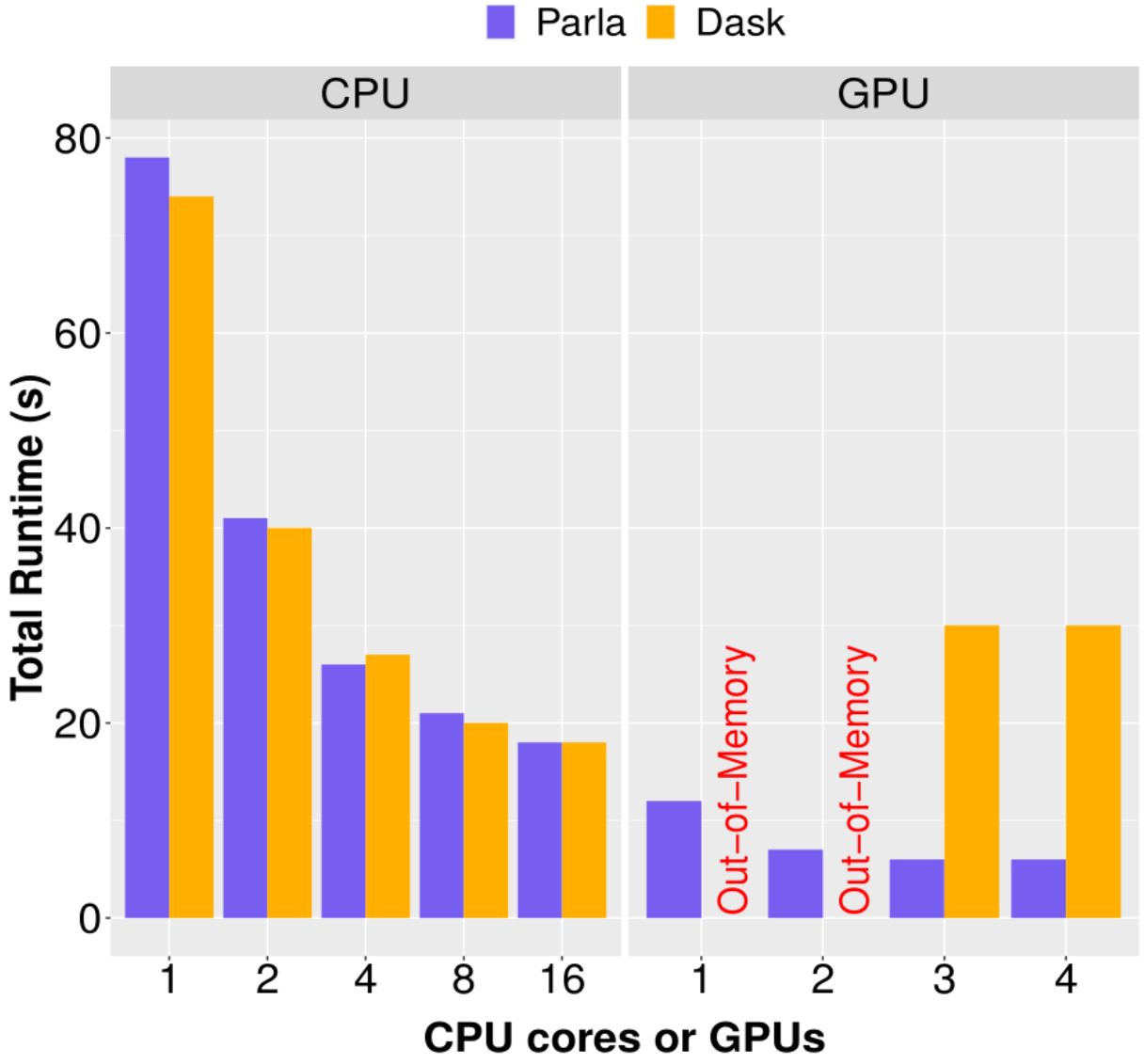
Parla scheduler utilizes GPUs well



Good performance and scalability

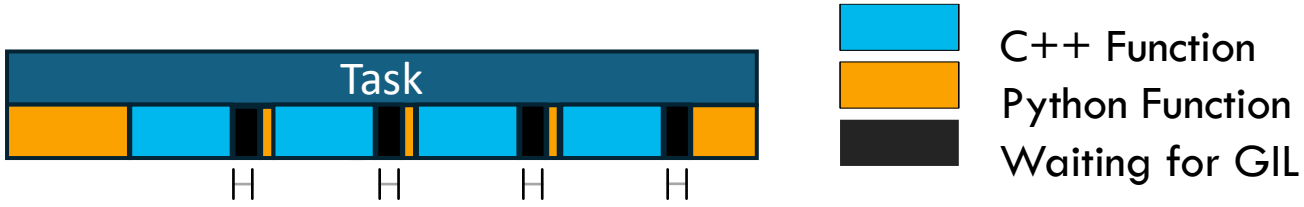


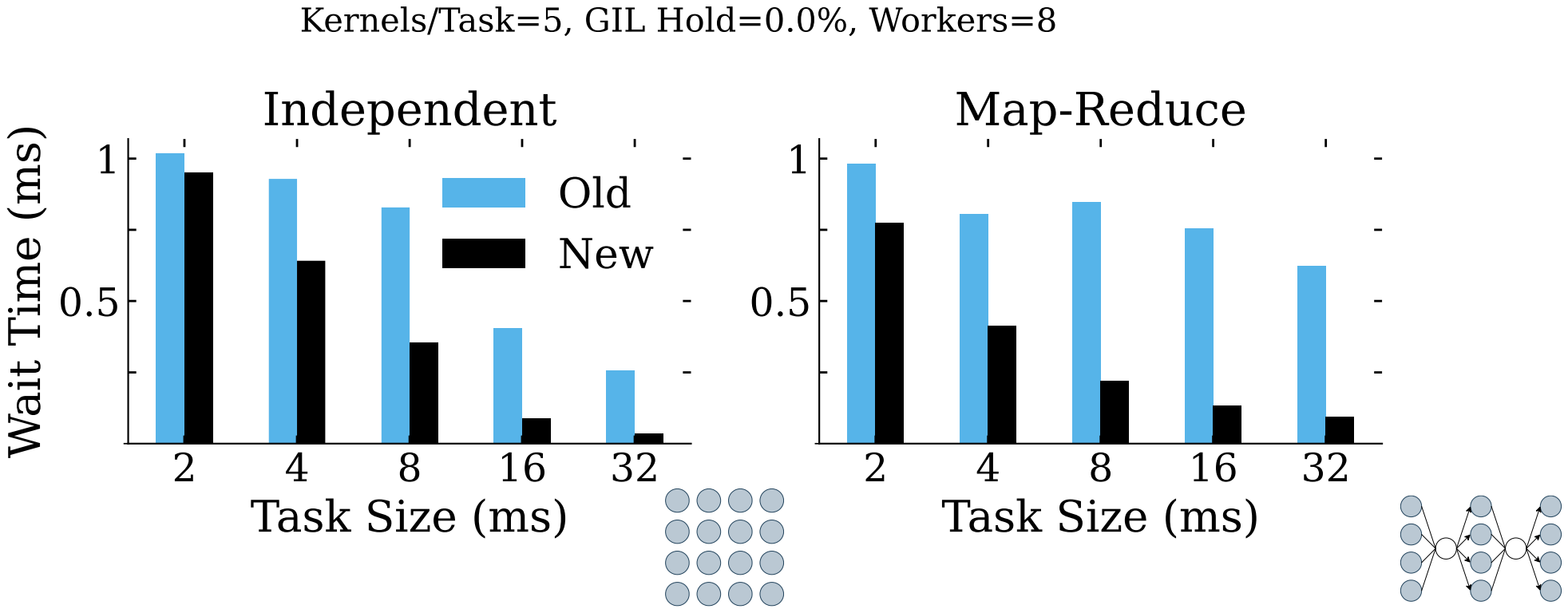
Good performance and scalability (vs. Dask)



Reduction in GIL-contention

New runtime has less time waiting for GIL





The new runtime shows an advantage even in a proposed "No GIL" Python (PEP703)