Ray: A Distributed Framework for Emerging AI Applications & Lineage Stash: Fault Tolerance Off the Critical Path

Shucheng Zhong, Han You, Joshua Segal

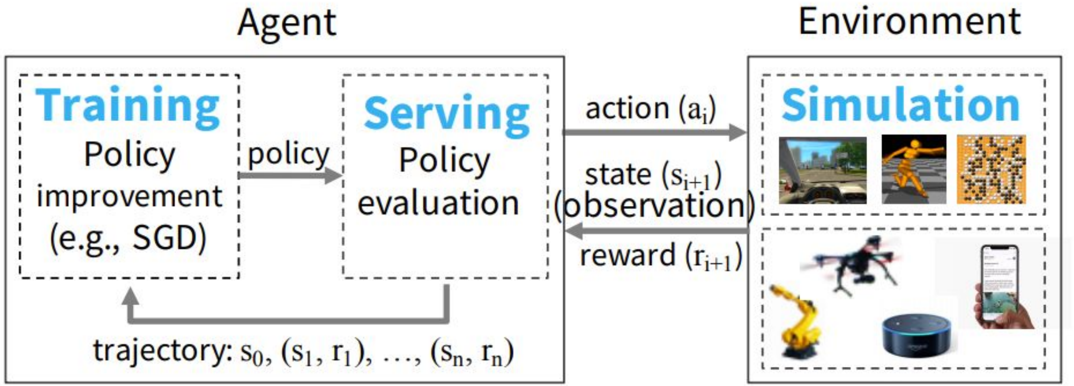
Overview

- Ray
  - Background & Motivation
  - API
  - Architecture
- Lineage Stash
  - Background & Motivation
  - Architecture
- Evaluations of Ray and Lineage Stash

Systems for reinforcement learning

Three parts of RL:

- Simulation
- Training
- Serving



* Figure from *Ray: A Distributed Framework for Emerging AI Applications*

Machine learning ecosystem nowadays

Distributed Training (e.g. Tensorflow, MXNet, Horovod)	Model Serving (e.g. Tensorflow Serving, Clipper)	Simulation (e.g. CIEL, Dask)	Many other systems … ...

Glue a bunch of distributed systems together

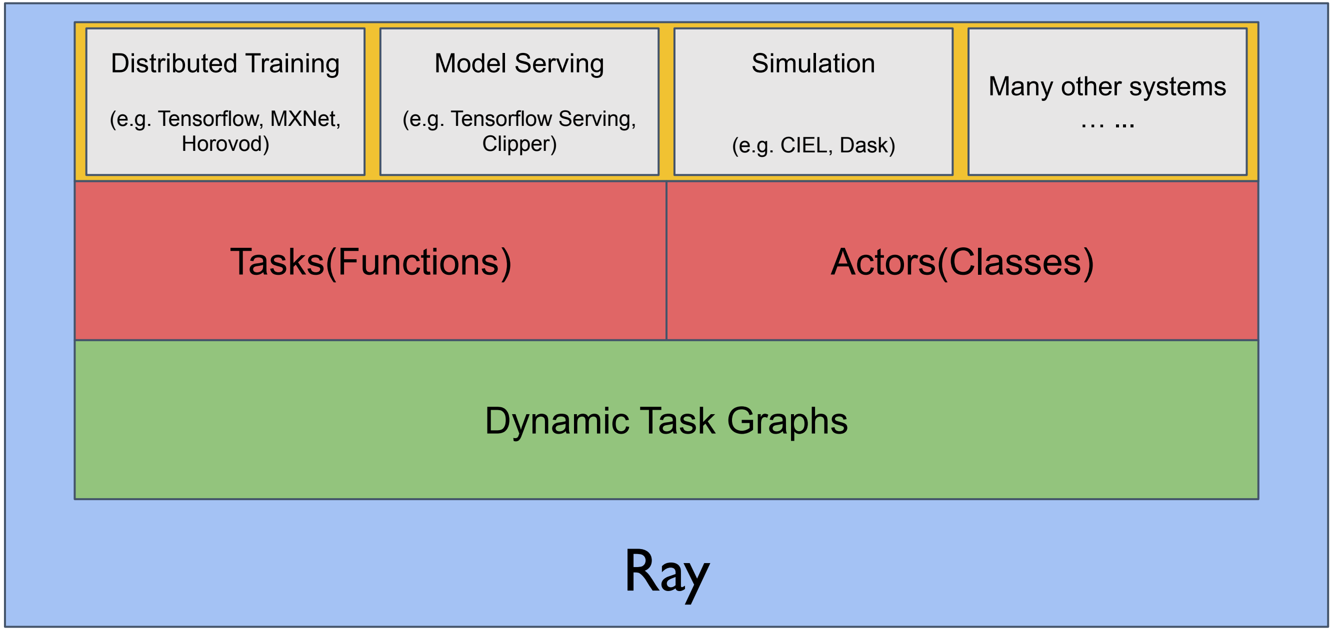
Or build your own system from scratch

What is Ray?

A general-purpose cluster-computing framework that supports

- fine-grained computations (millions of tasks within millisecond-level latency)
- Heterogeneity both in time and in resource usage
- dynamic execution

Overview of Ray



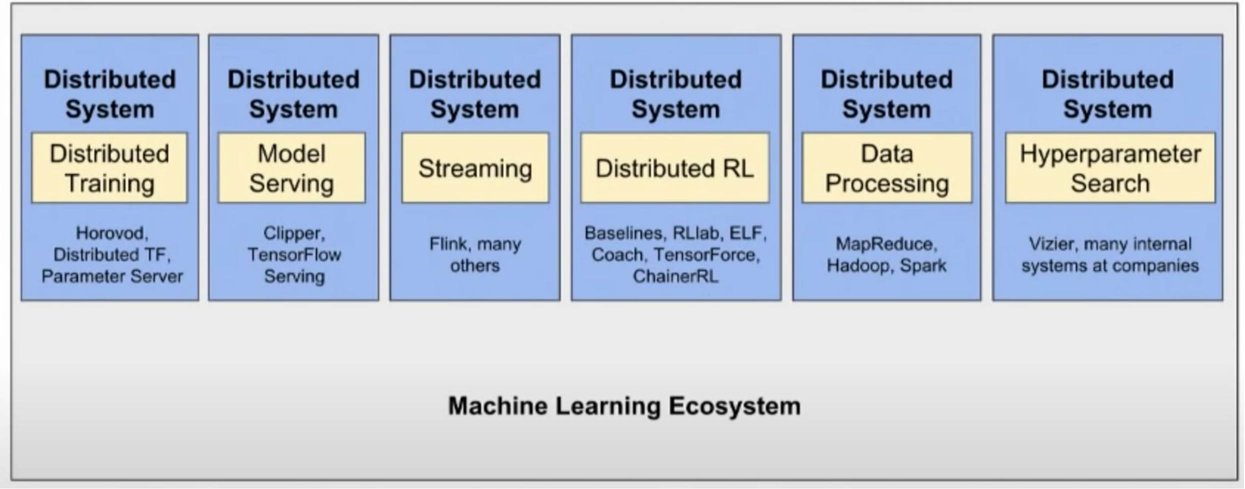
Ray Libraries

Programming Model

Computation Model

Motivation

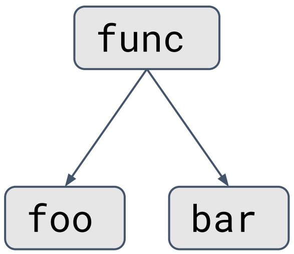
The Old Way: Glueing a bunch of systems together -- or build from scratch



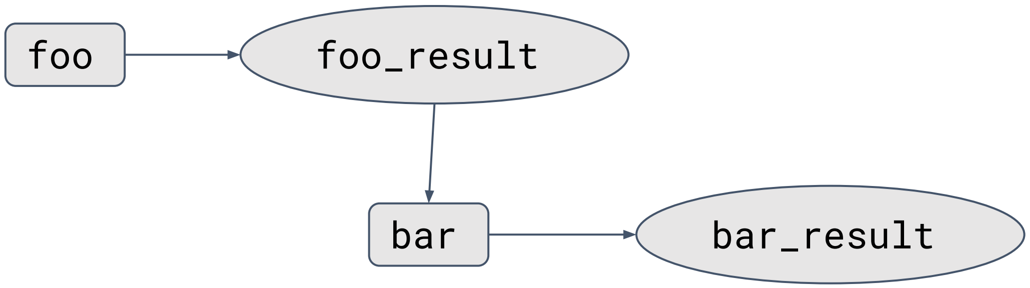
Ray's Computation Model

Dependencies represented as edges in task graph.

```
@ray.remote
def func():
    foo_result = foo.remote()
    bar_result = bar.remote(foo_result)
```



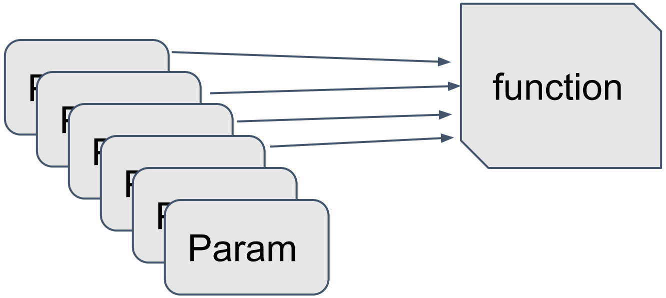
Control dependency



Dataflow dependency

Ray’s Computation Model

Tasks and Actors are automatically triggered when inputs become available.



4 / 6 Available

Ray API: Tasks

```python
@ray.remote
def matrixMut(A, B):
    return A * B

@ray.remote
def matrixFunction(A, B):
    return np.eig(A) * np.eig(B)

@ray.remote
def matrixSum(A, B):
    return A + B
```

$$A \times B + f(B, C)$$

```python
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

Ray API: Actor

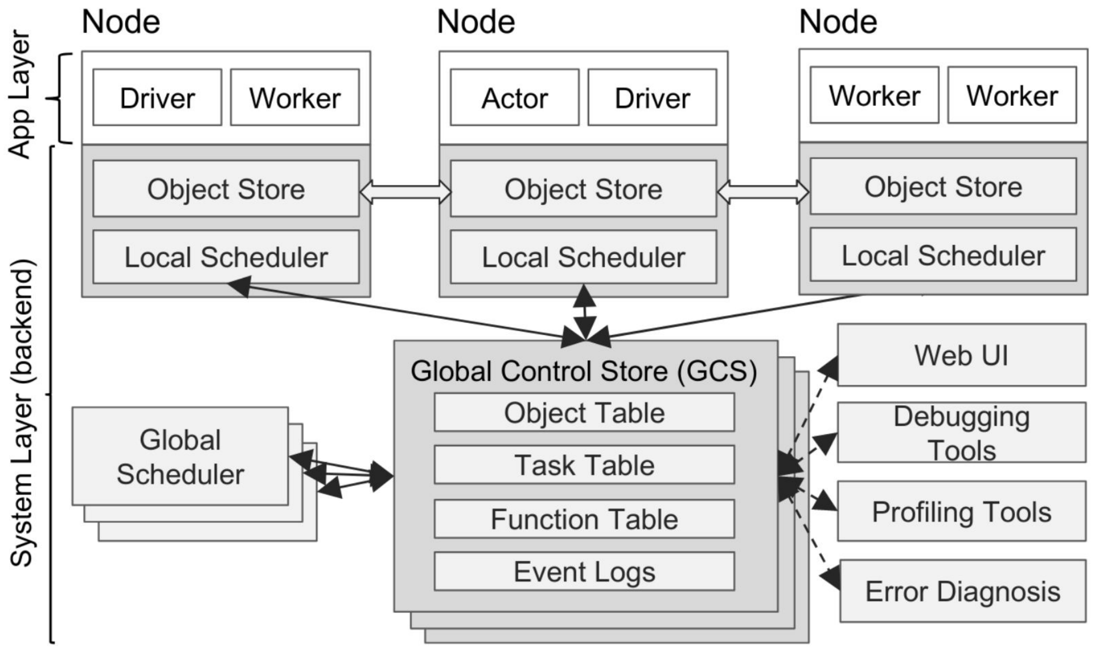
```python
@ray.remote
class Shuffler:
    def __init__(self, A):
        self.A = A

    def shuffle(self, seed):
        self.A = shuffle(self.A, seed)
        return self.A

shuffler = Shuffler.remote(result)
iter1 = shuffler.shuffle.remote(seed)
iter2 = shuffler.shuffle.remote(seed)
iter3 = shuffler.shuffle.remote(seed)
```

$$A' = \text{shuffle}(A,\ s)$$

Architecture



$$\mathrm{A}\times\mathrm{B} + f(\mathrm{B}, \mathrm{C})$$

A: node A
B: node B
C: node B

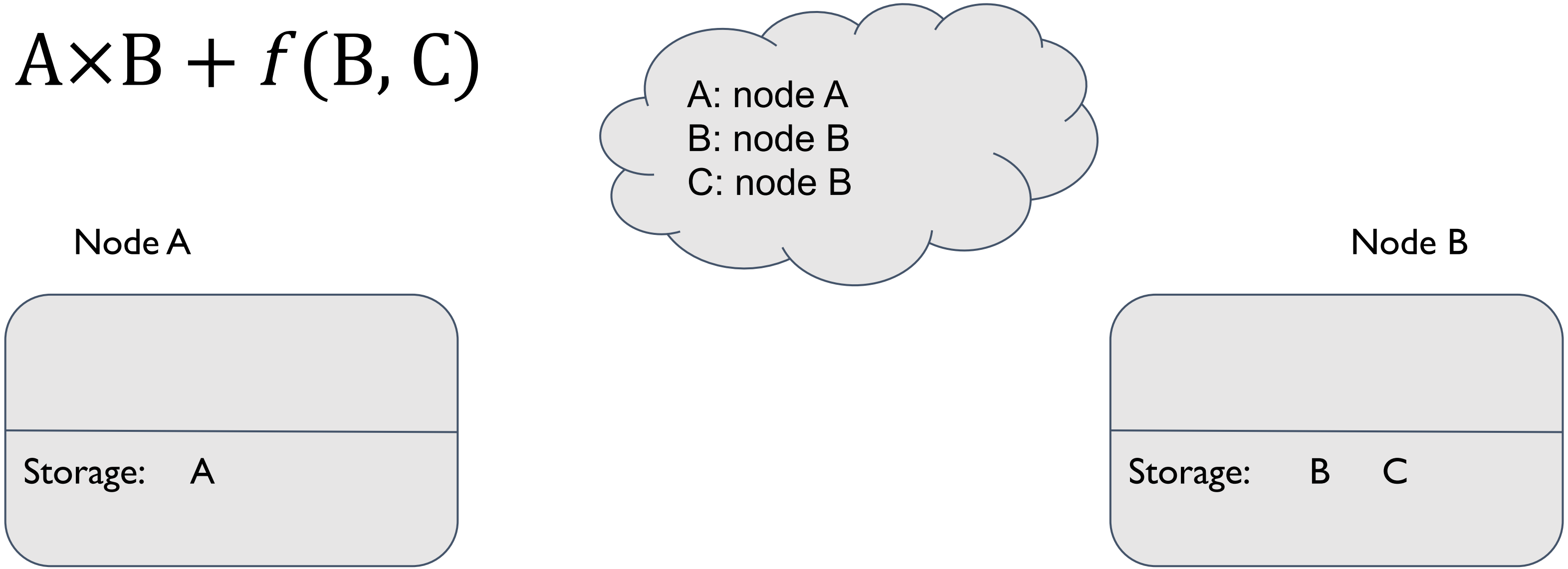
Node A

Storage: A

Node B

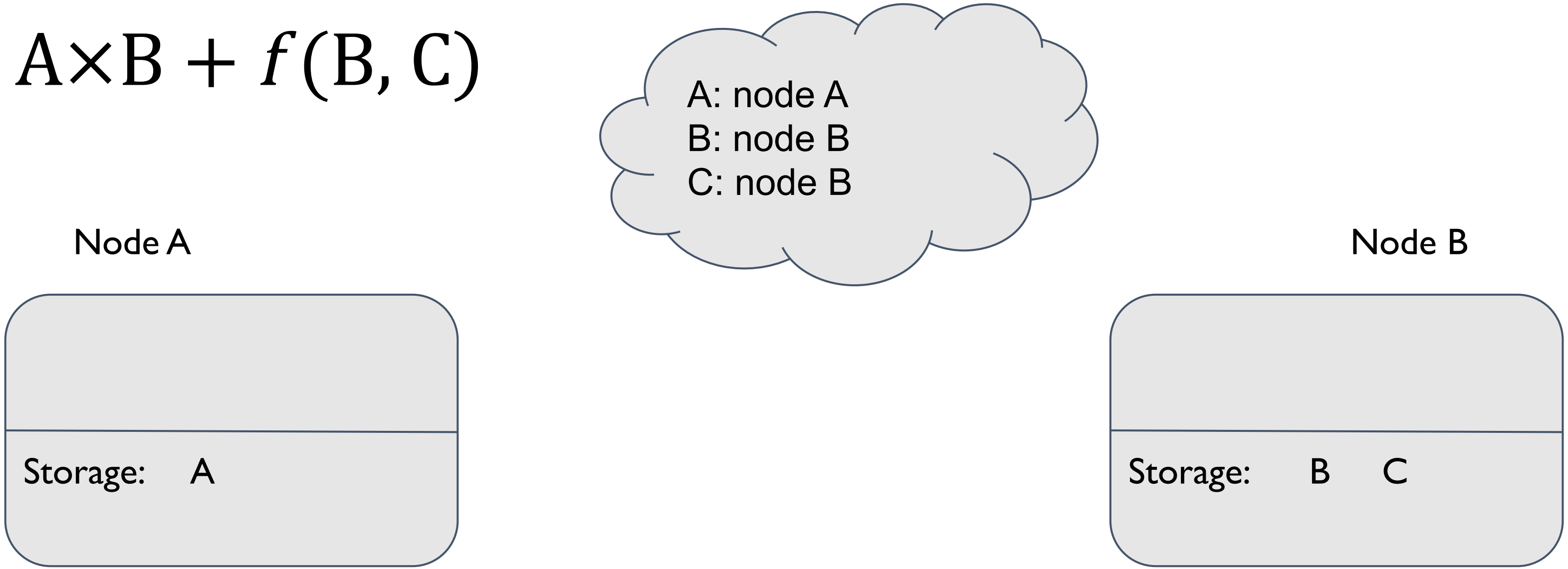
Storage: B C

$$A \times B + f(B, C)$$




```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

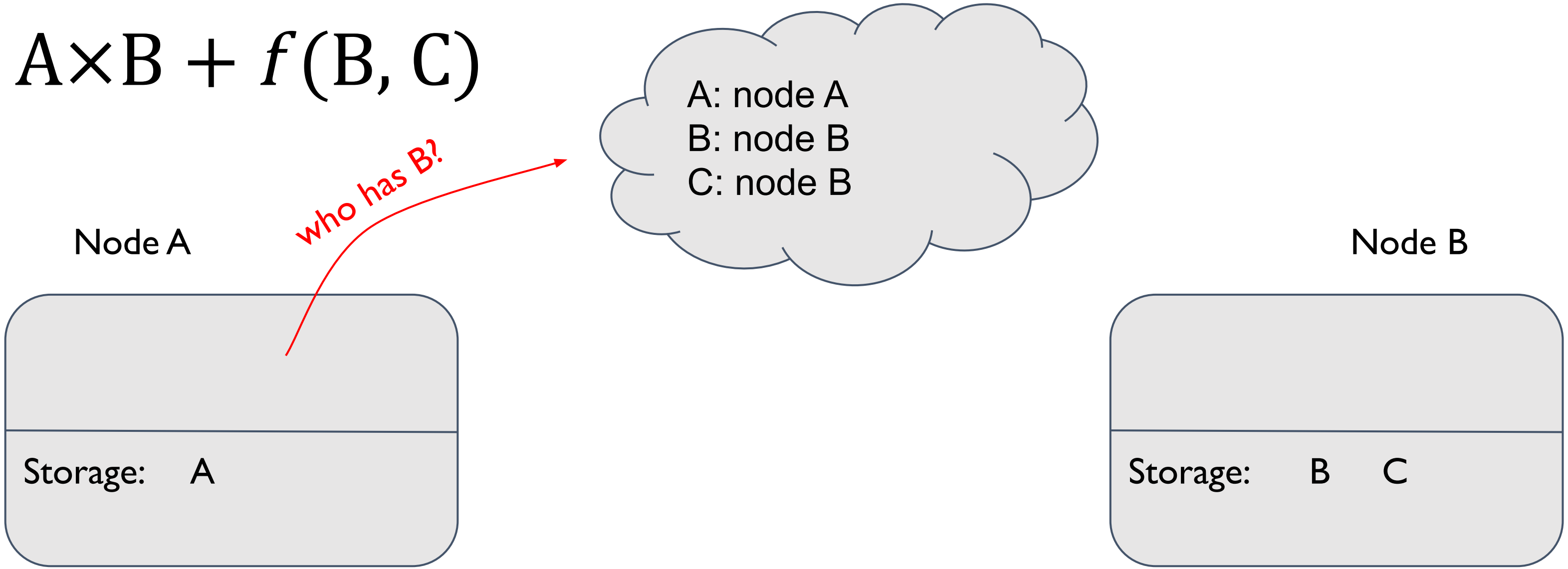
$A \times B + f(B, C)$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

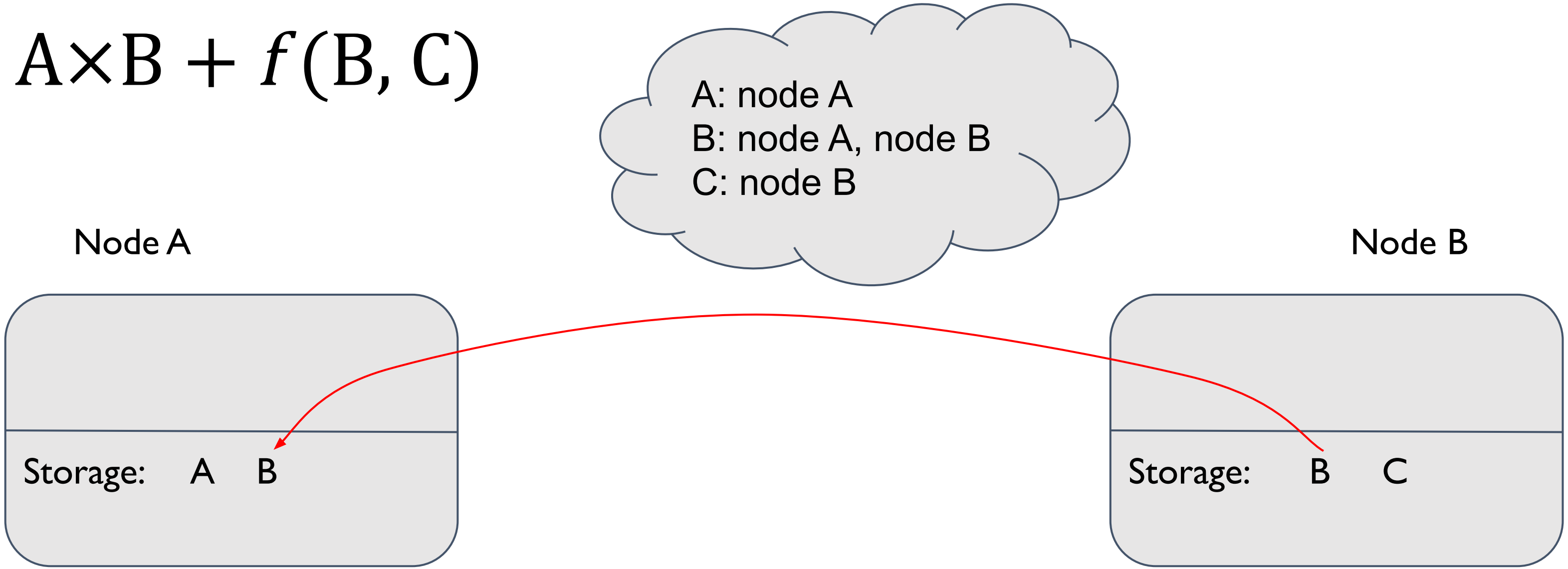
$\mathrm{A}\times\mathrm{B} + f(\mathrm{B}, \mathrm{C})$



Local scheduler of A: Let’s do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

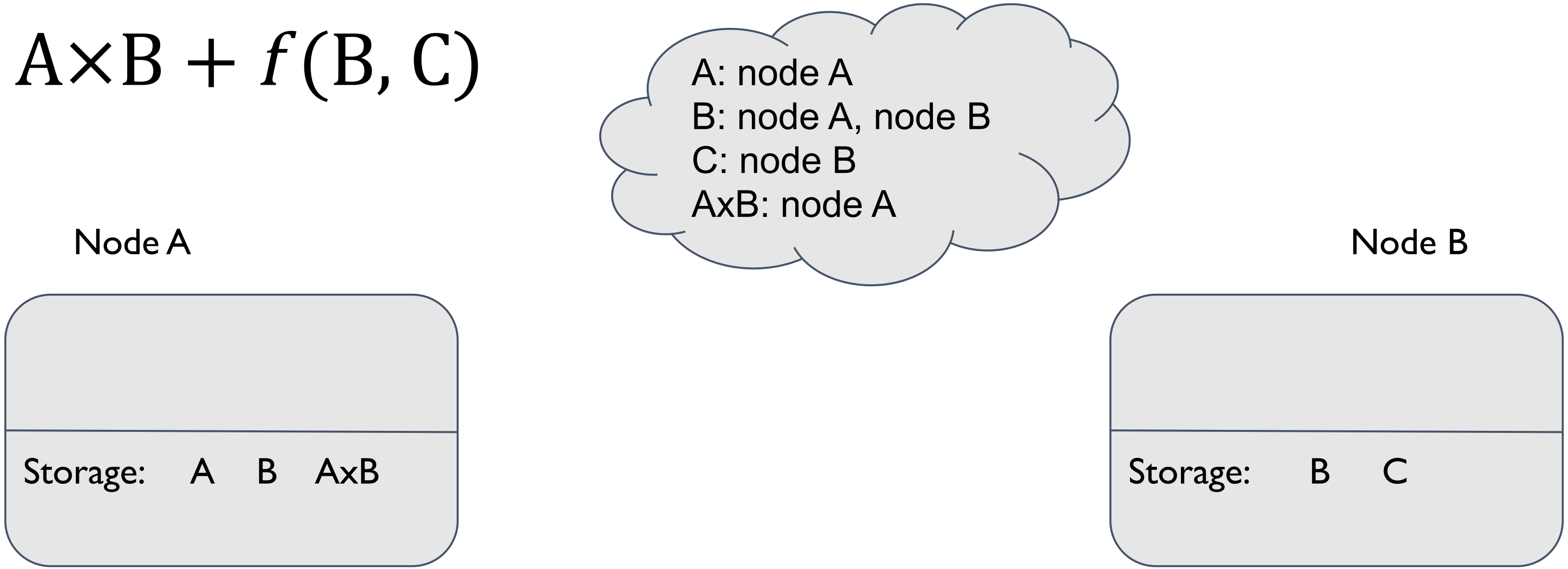
$$A\times B + f(B, C)$$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

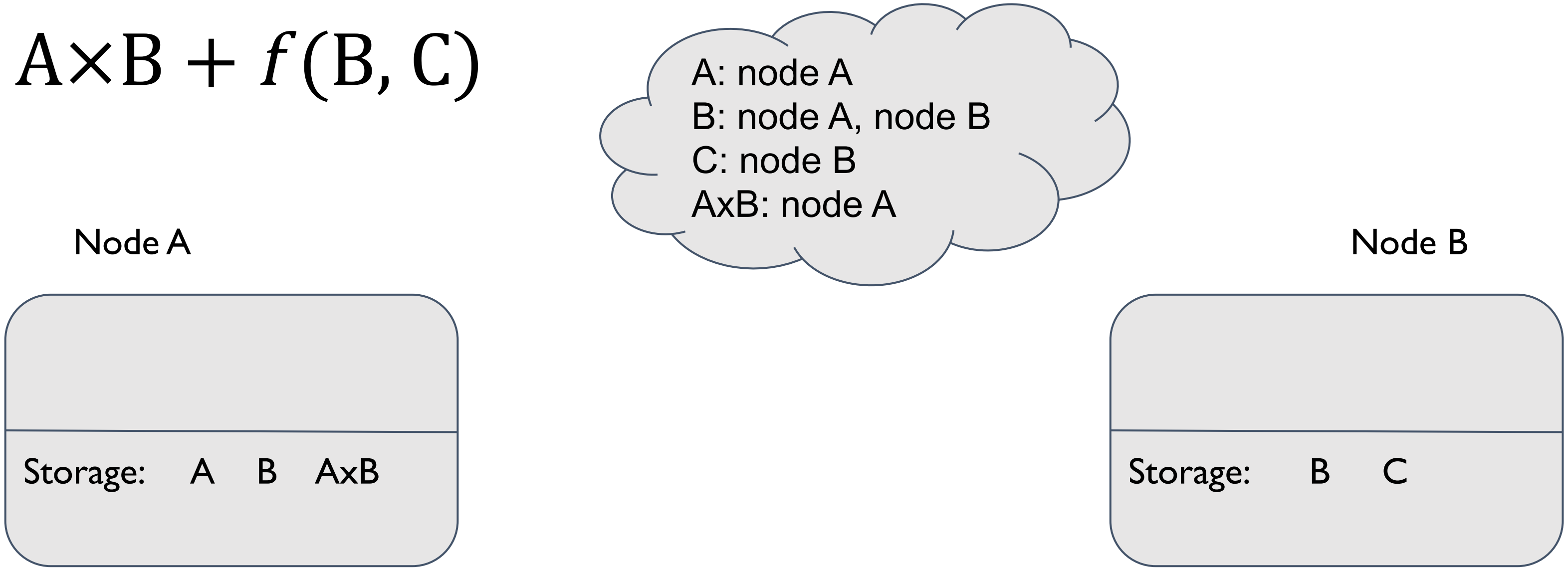
$$A \times B + f(B, C)$$



Local scheduler of A: Let's do this ourselves

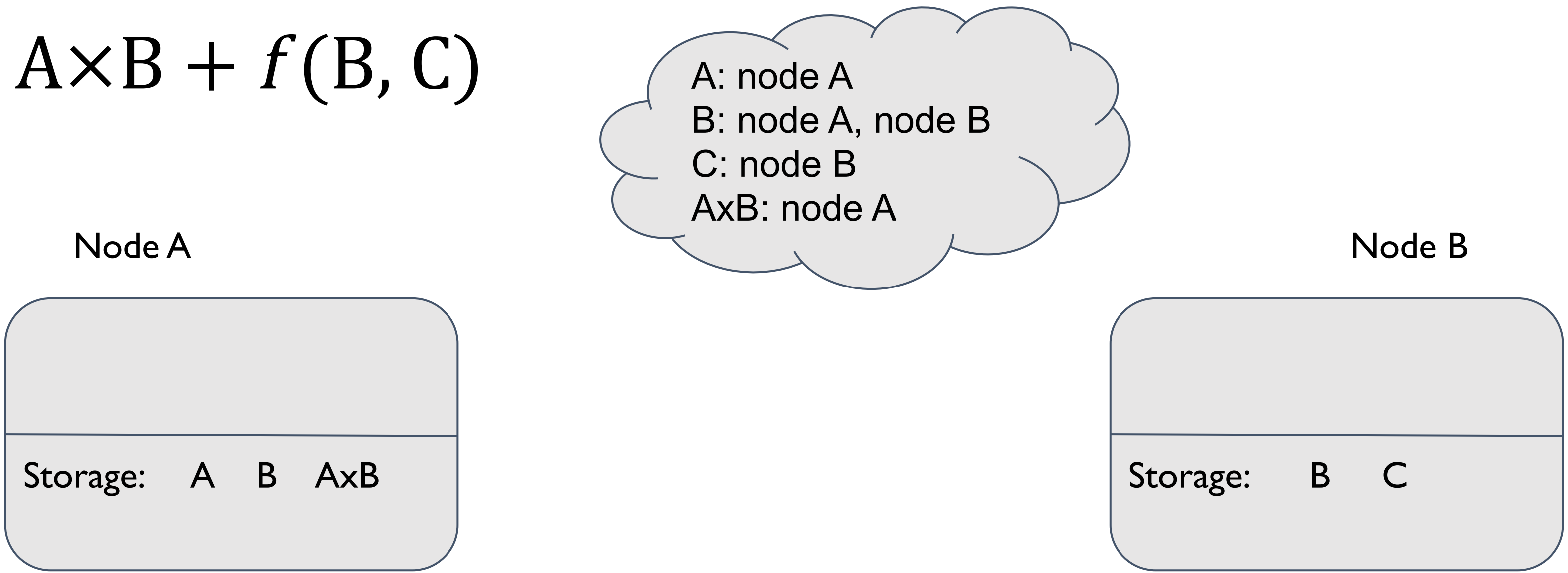
```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$




```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

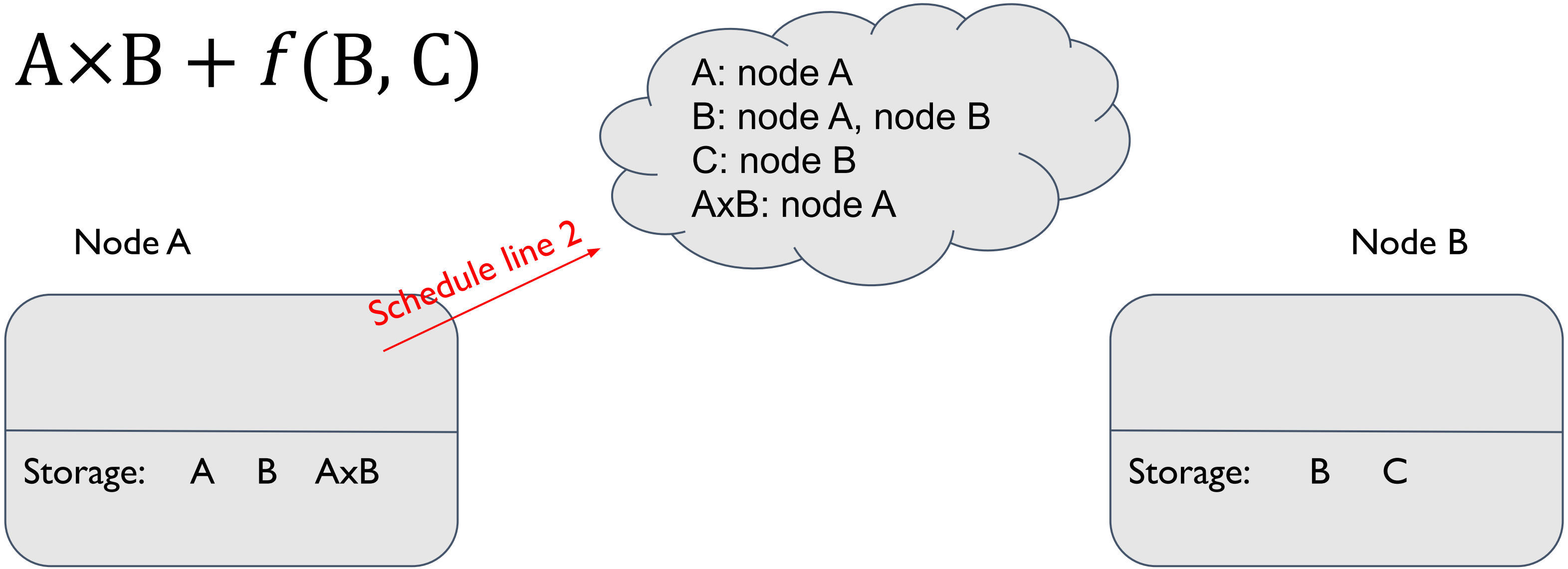
$$A \times B + f(B, C)$$



Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

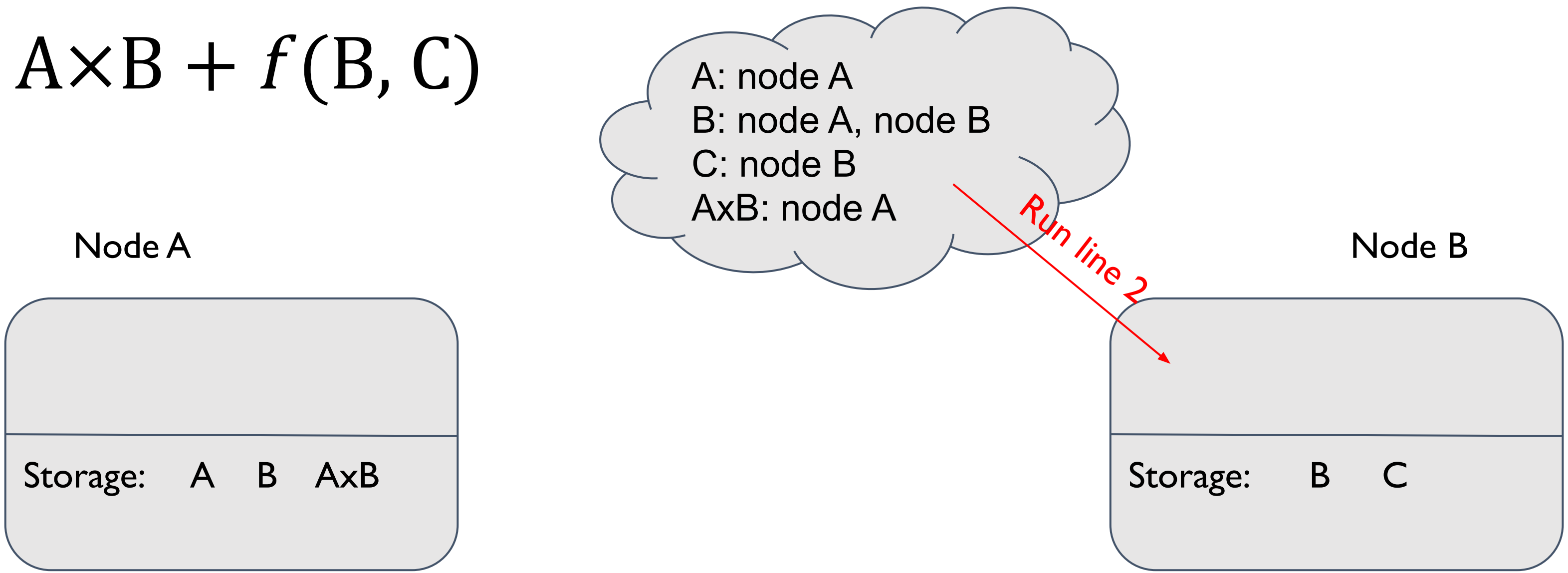
$A \times B + f(B, C)$



Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

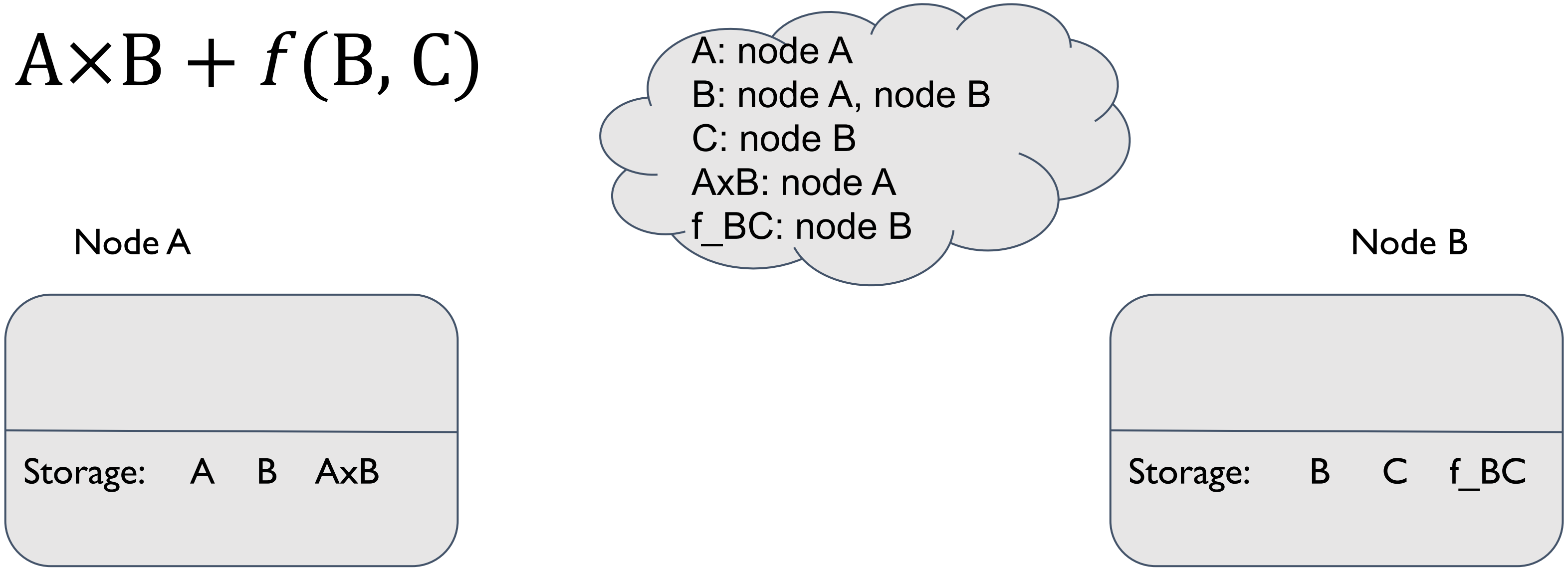
$$A \times B + f(B, C)$$



Local scheduler of A: I don’t want to do this

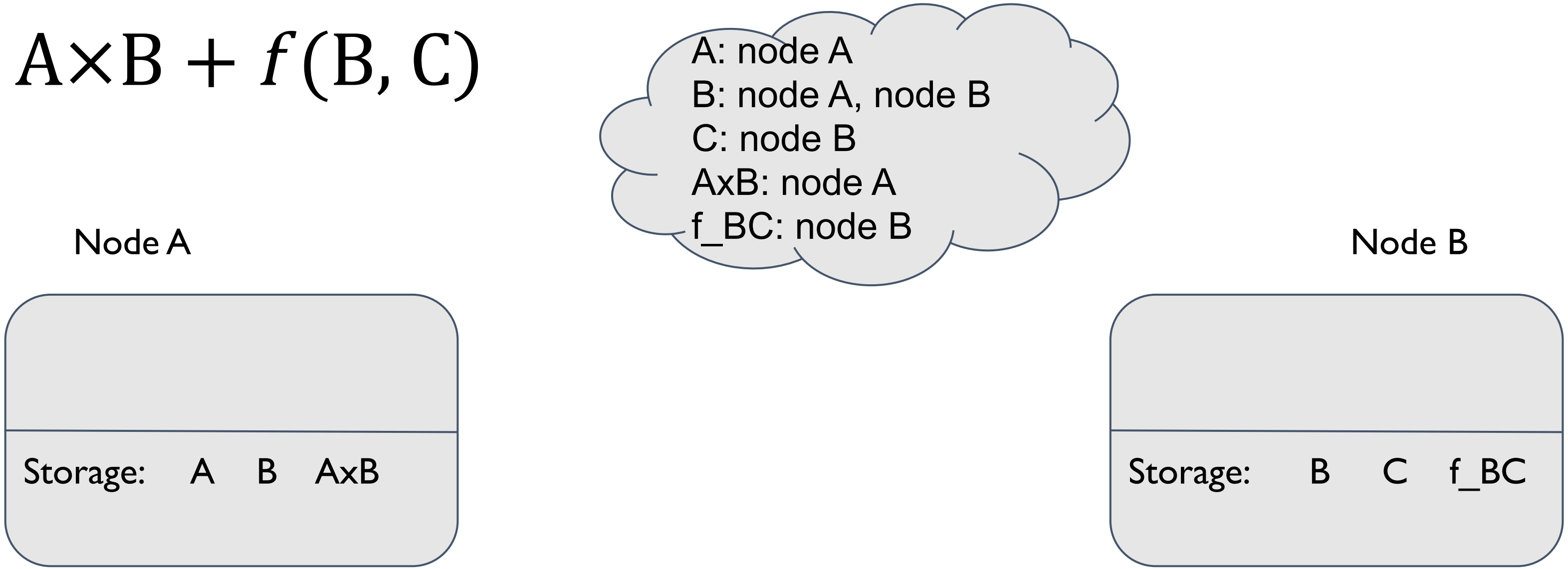
```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$



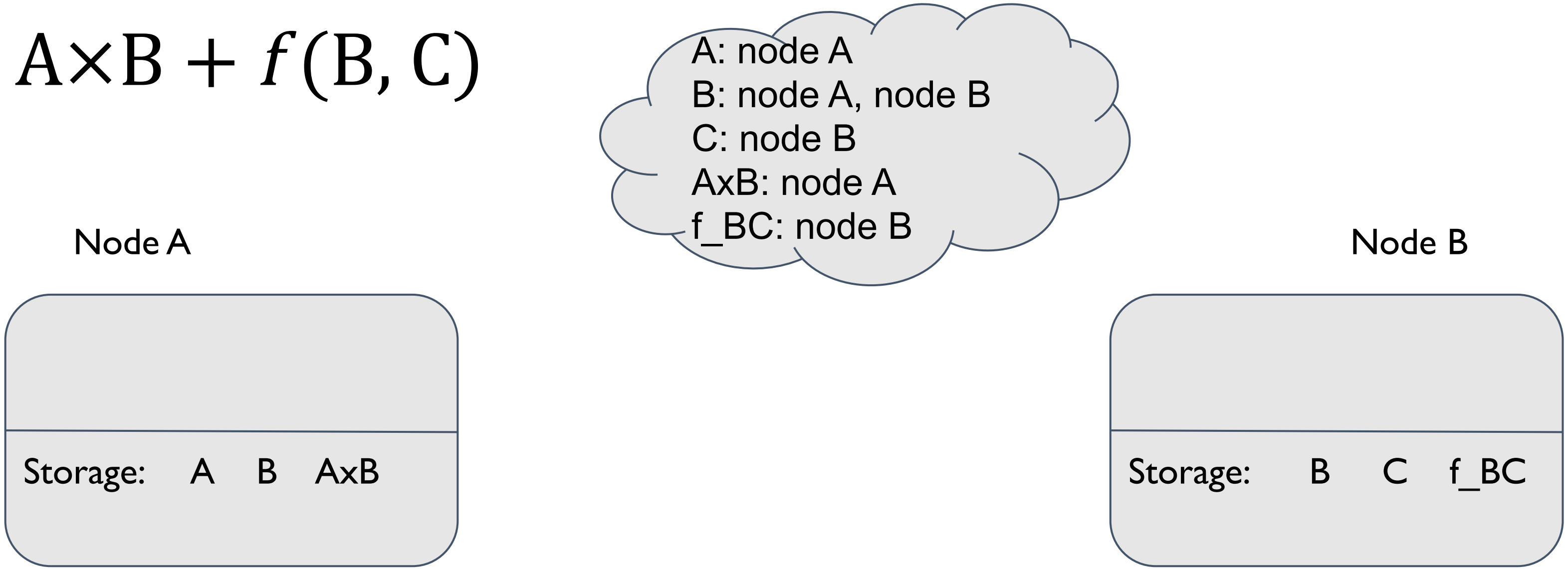

```python
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$




```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

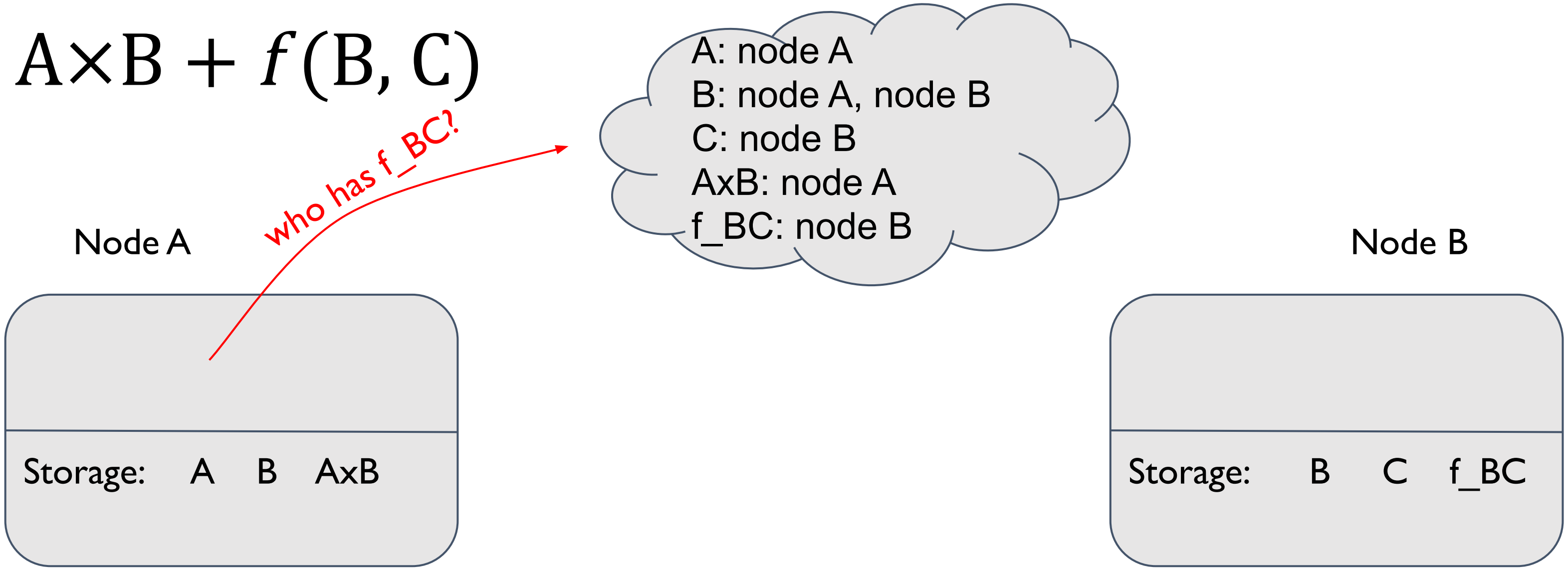
$$\mathrm{A}\times\mathrm{B} + f(\mathrm{B}, \mathrm{C})$$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

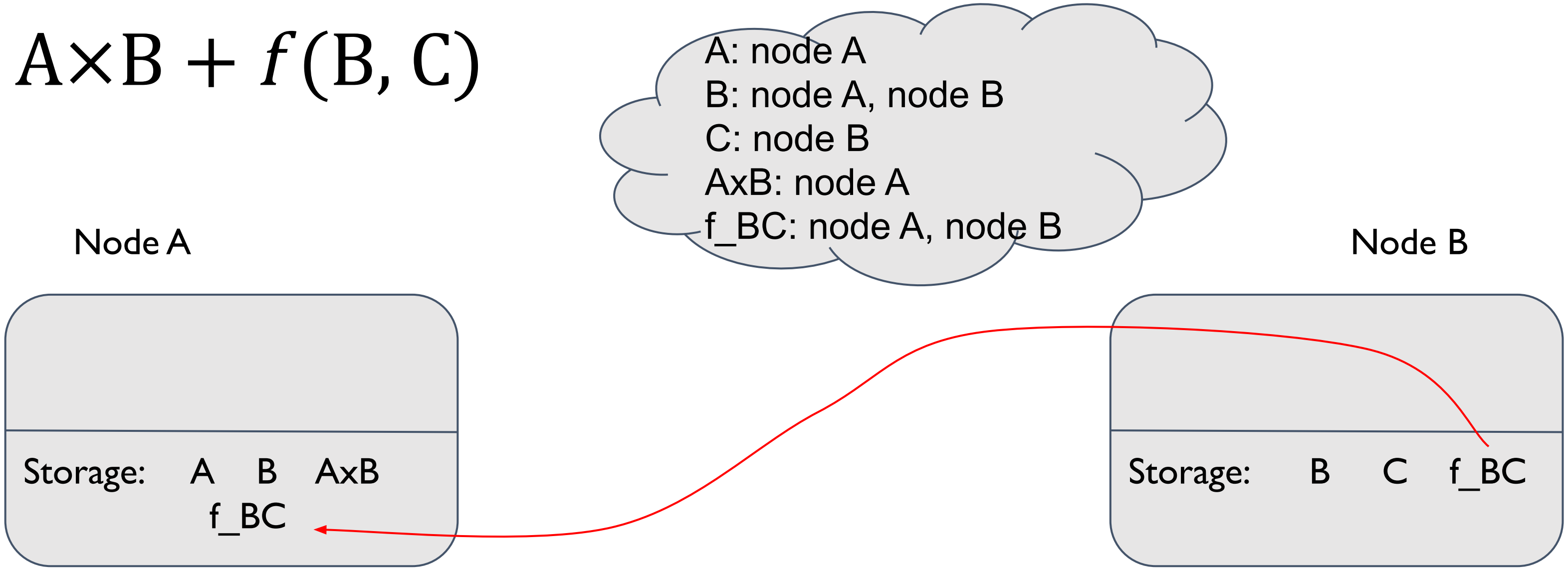
$$A \times B + f(B, C)$$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$A \times B + f(B, C)$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$

A: node A
B: node A, node B
C: node B
AxB: node A
f_BC: node A, node B

Node A

Storage: A B AxB f_BC

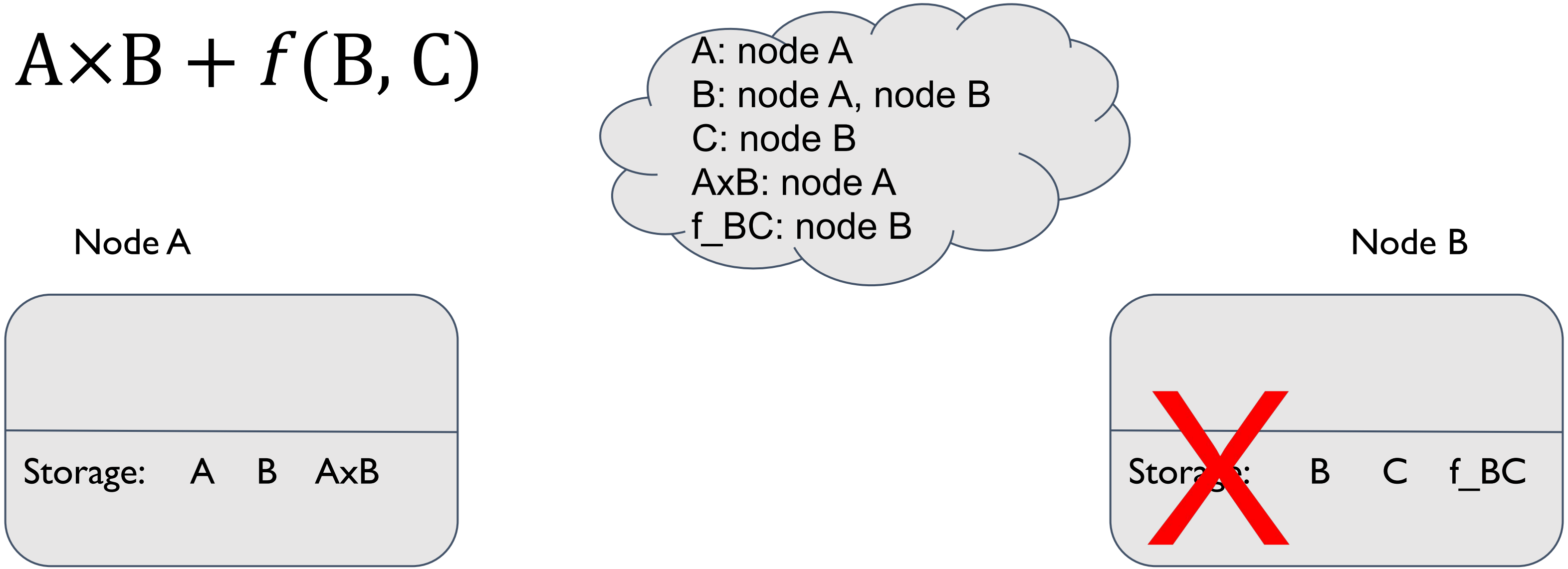
Node B

Storage: B C f_BC

Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A\times B + f(B, C)$$



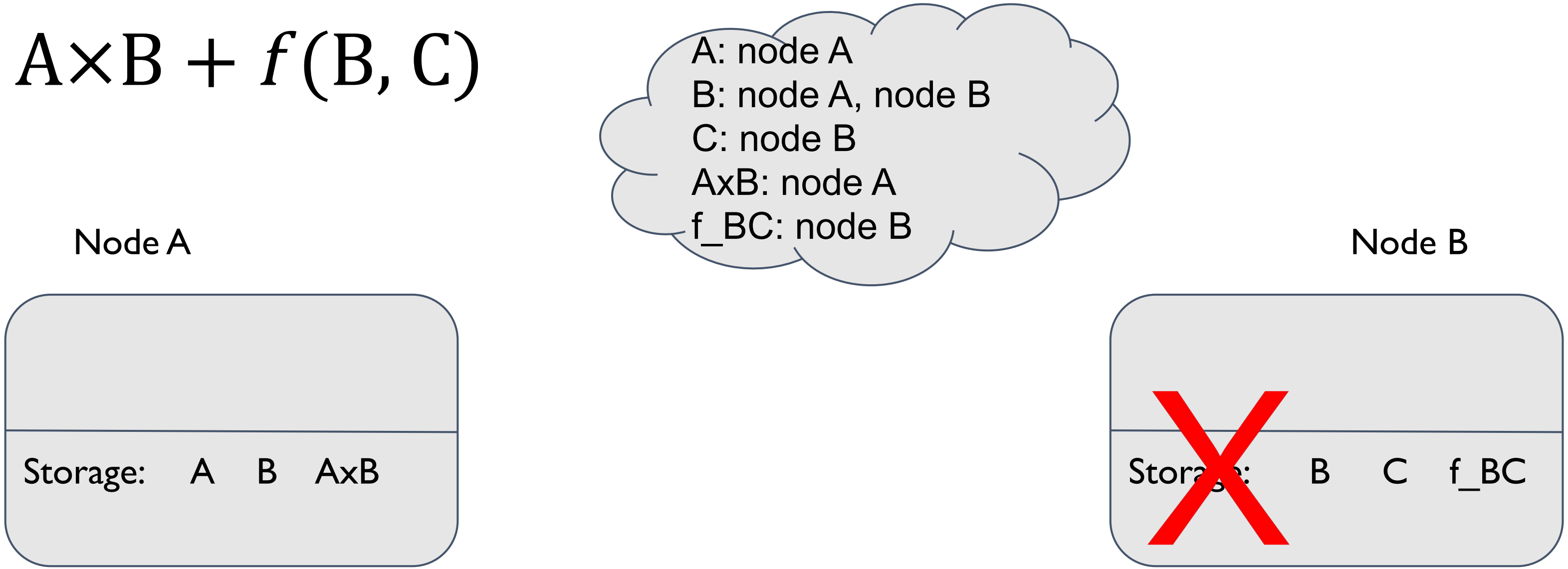
Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```


That didn't go as planned

Break!

$$A \times B + f(B, C)$$



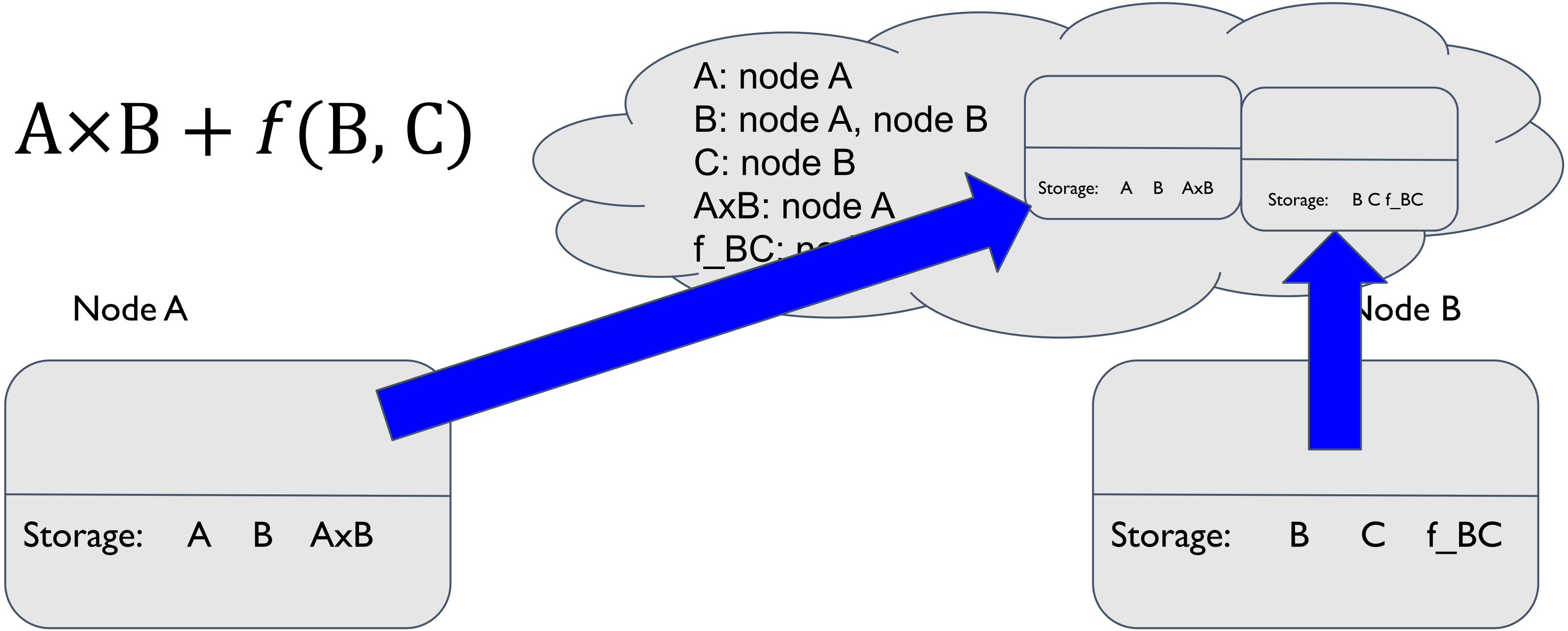
Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

Attempt 1: Global Checkpoint

- save the entire state of the workers periodically

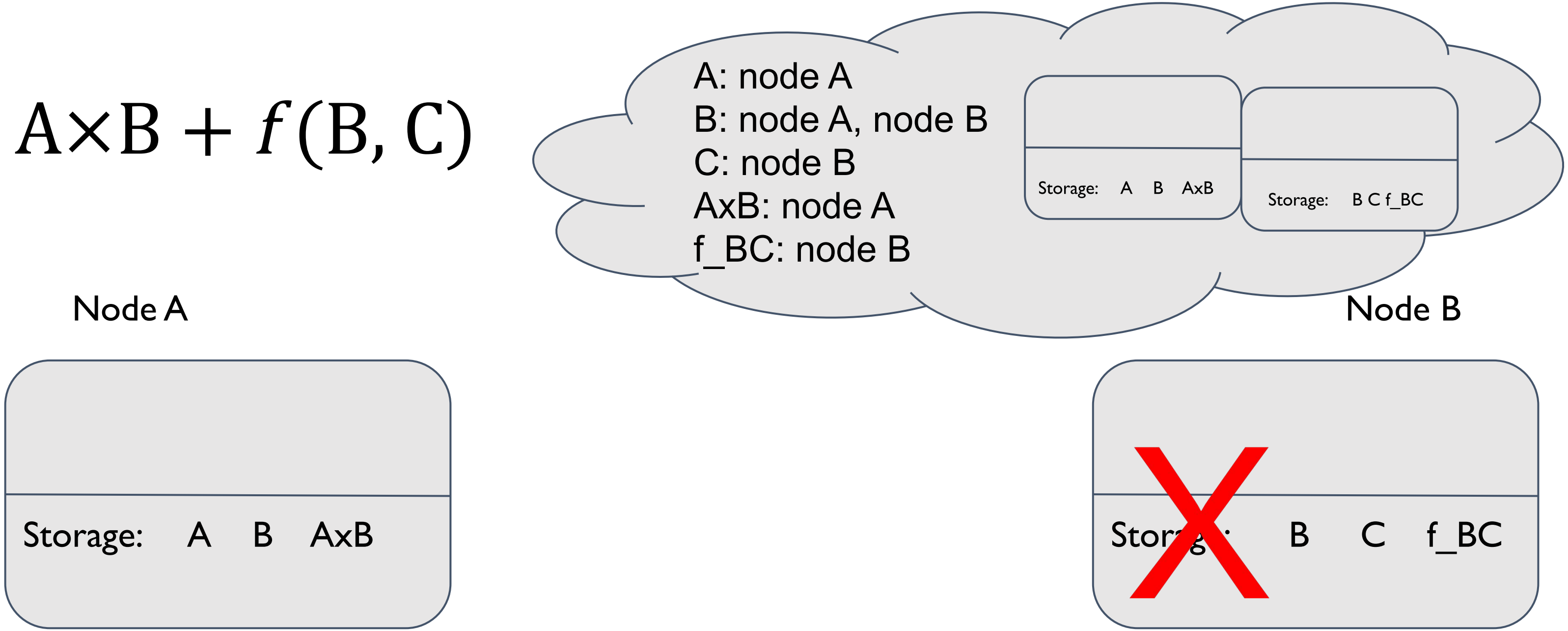
$$\mathrm{A}\times\mathrm{B} + f(\mathrm{B}, \mathrm{C})$$



Local scheduler of A: Let’s do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

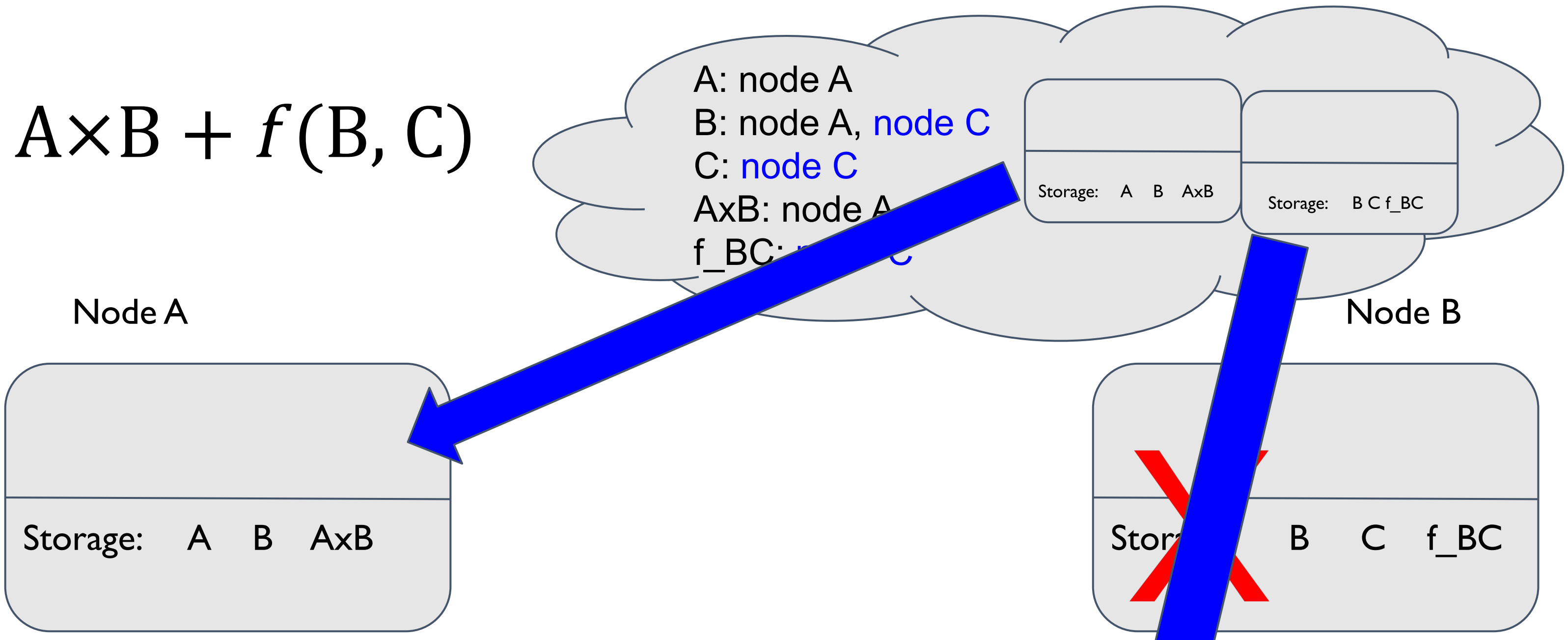
$$A\times B + f(B, C)$$



Local scheduler of A: Let's do this ourselves

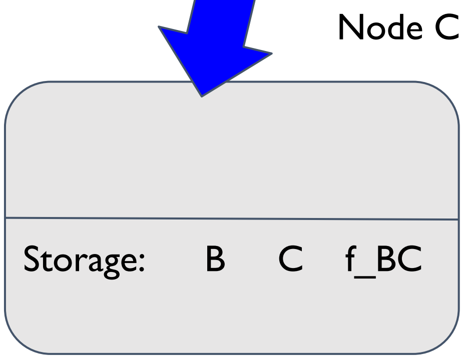
```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$\mathrm{A}{\times}\mathrm{B} + f(\mathrm{B}, \mathrm{C})$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```



Global Checkpoint

- high recovery time

Global Checkpoint

- high recovery time
- low overhead
    - State is sent in the background asynchronously

Global Checkpoint

- high recovery time
- low overhead
    - State is sent in the background asynchronously
- good for fine grained tasks (millisecond)
    - small batches
    - streams of data

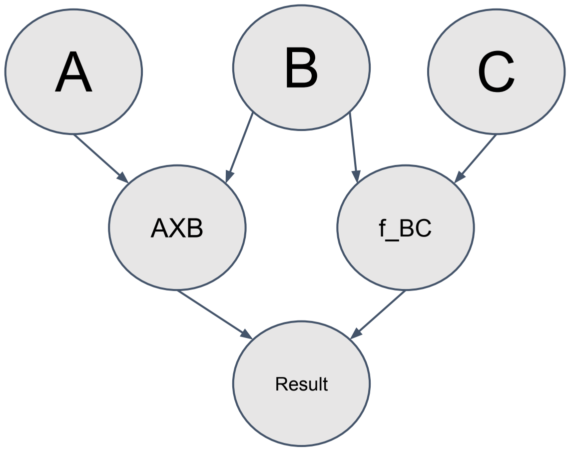
Attempt 2: Lineage Logging

- Store the lineage

Attempt 2: Lineage Logging

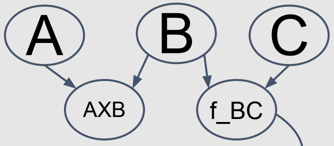
- Save the logs on what's dependent on what

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```



$$A \times B + f(B, C)$$

A: node A
B: node A, node B
C: node B
AxB: node A
f_BC: node B



Node A

Storage: A B AxB

Node B

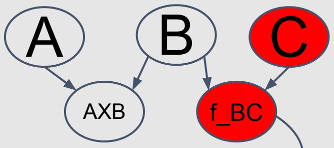
Storage: B C f_BC

Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$

A: node A
B: node A, node B
C: node B
AxB: node A
f_BC: node B



Node A

Storage: A B AxB

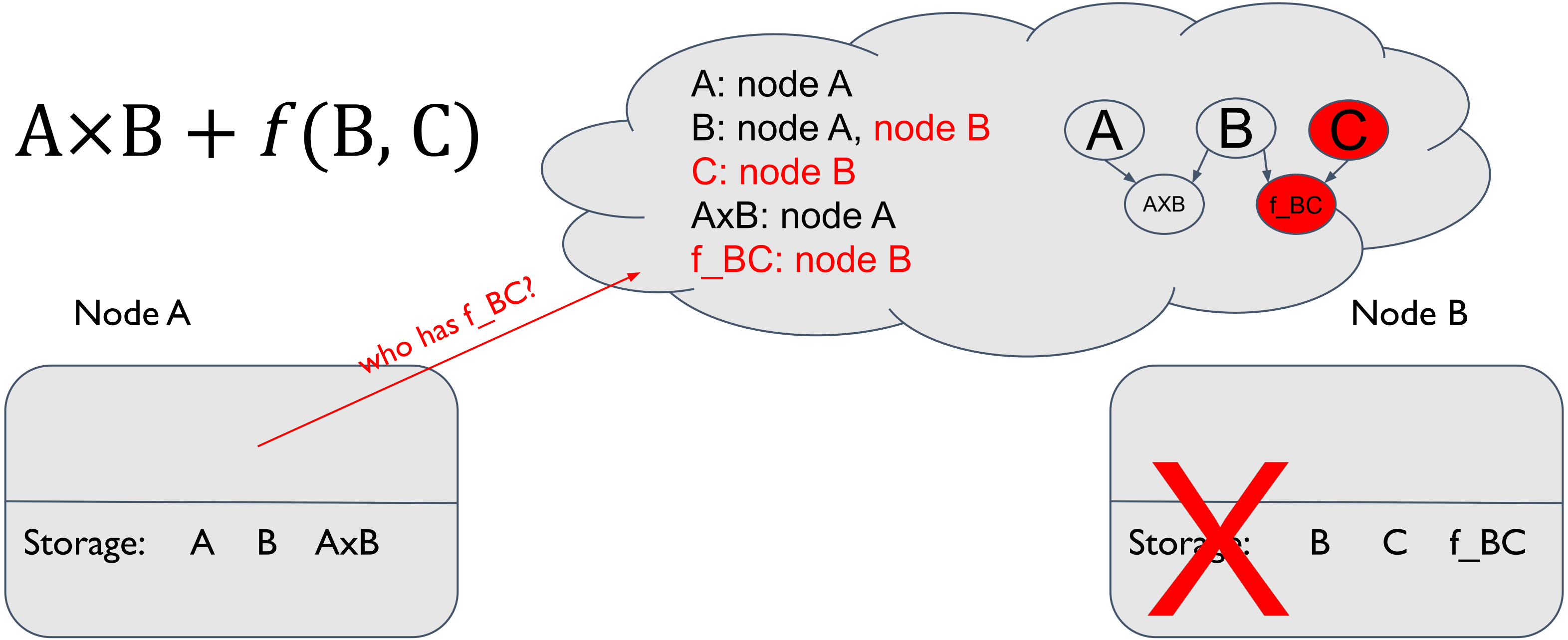
Node B

Storage: B C f_BC

Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

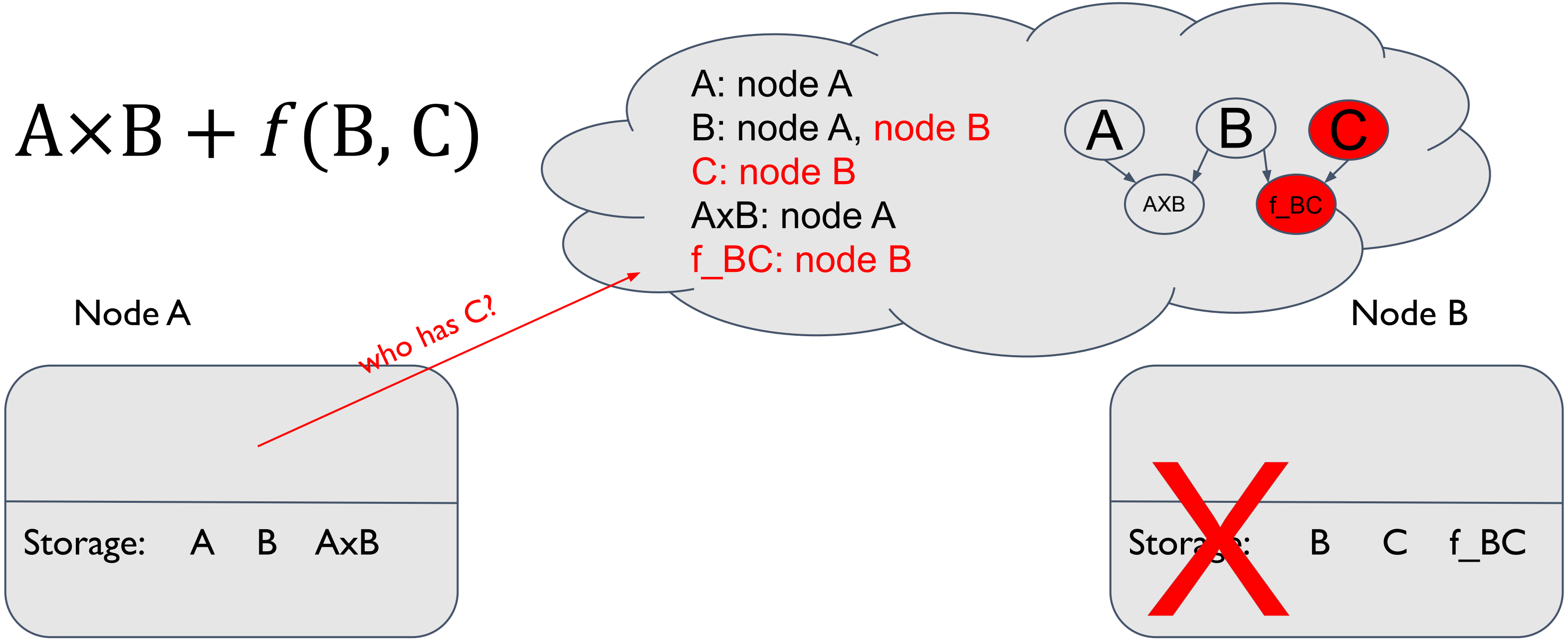
$$A \times B + f(B, C)$$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

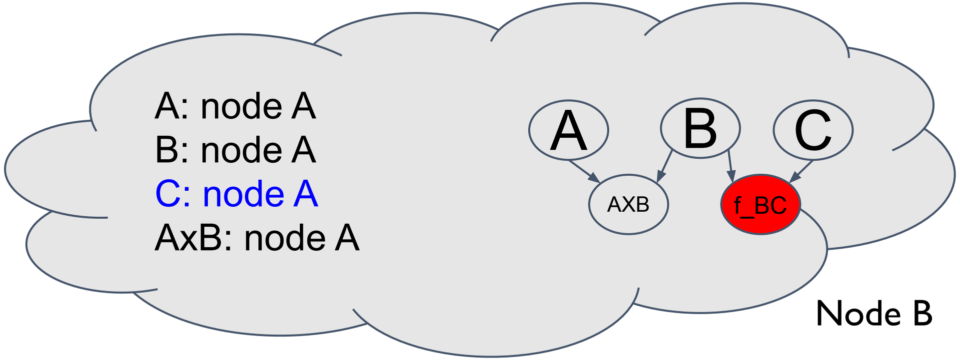
$$A \times B + f(B, C)$$



Local scheduler of A: Let's do this ourselves

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$

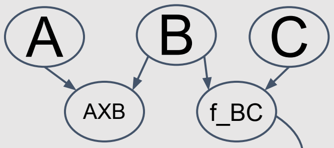


Local scheduler of A: Let's do this ourselves

```python
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$

A: node A
B: node A
C: node A
AxB: node A
f_BC: node A



Node A

Storage: A B AxB C f_BC

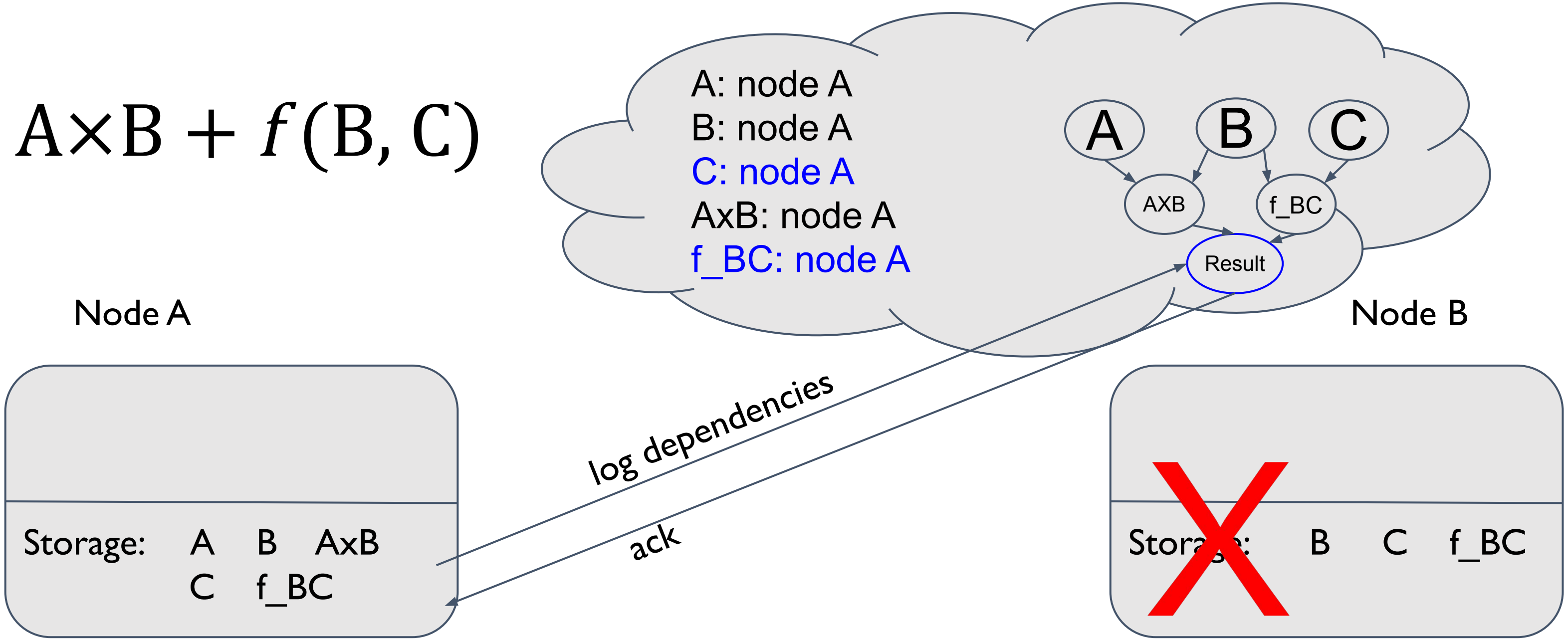
Node B

Storage: B C f_BC

Local scheduler of A: Let's do this ourselves

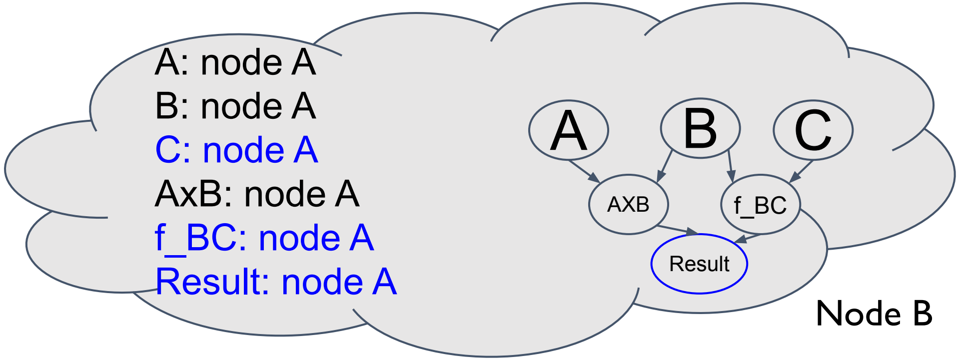
```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

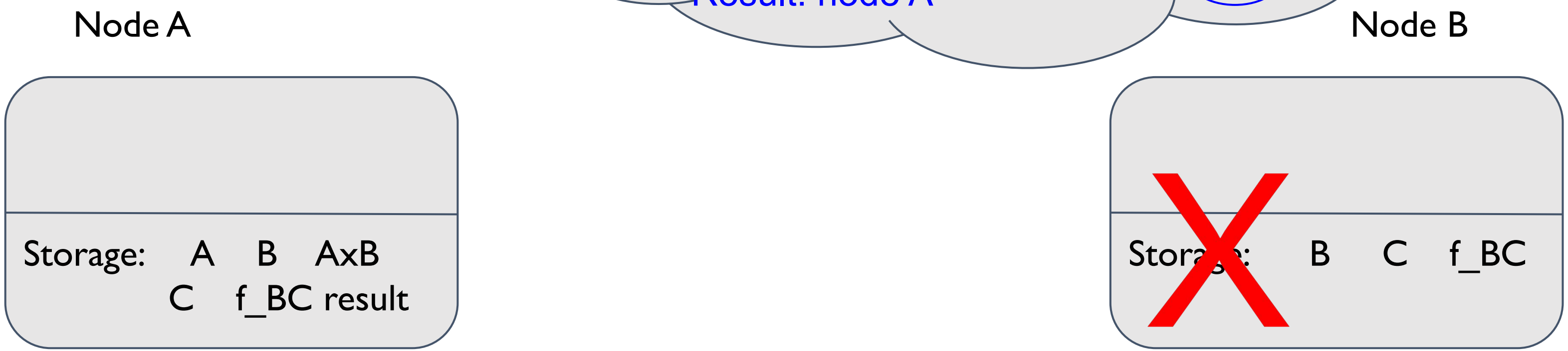
$$A \times B + f(B, C)$$




```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

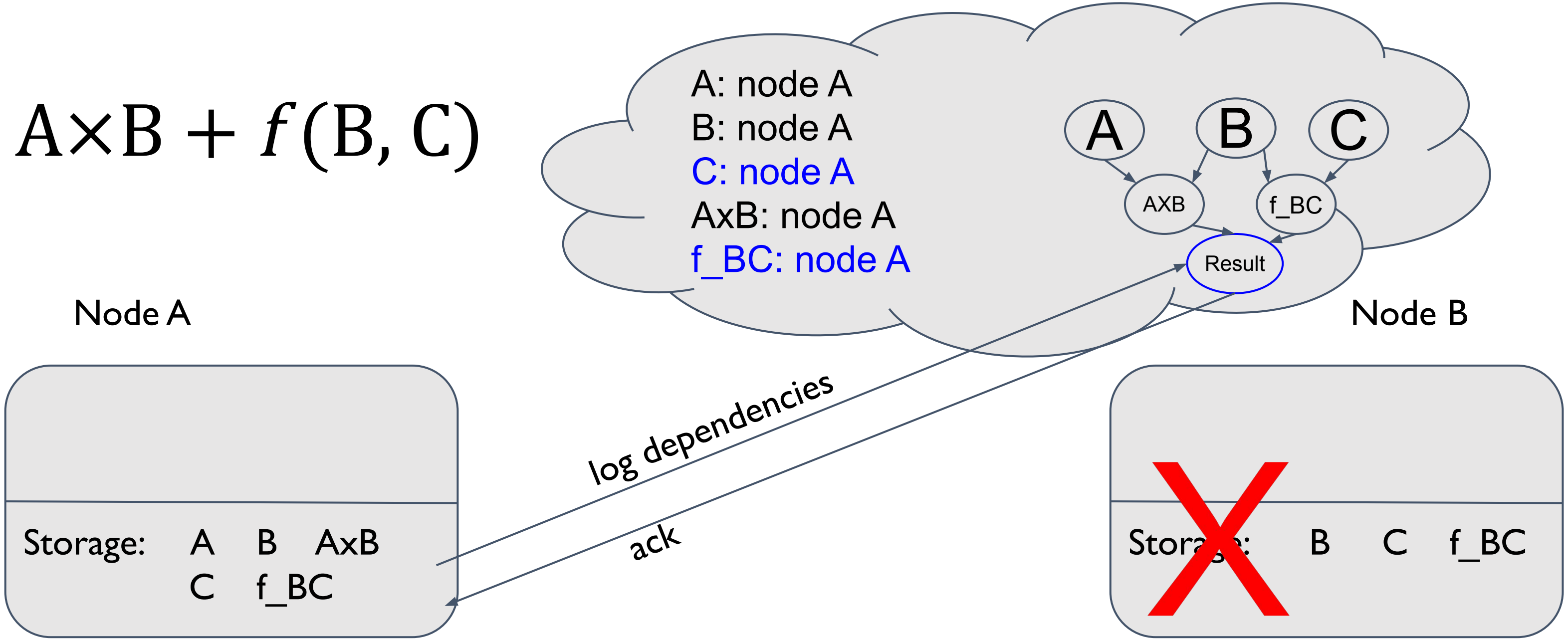
$$A \times B + f(B, C)$$






```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

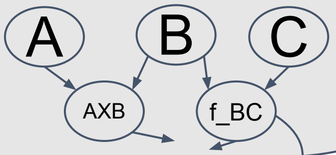
$$A \times B + f(B, C)$$




```python
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
print ray.get(result)
```

$$A \times B + f(B, C)$$

A: node A
B: node A
C: node A
AxB: node A
f_BC: node A



log dependencies

Node A

Storage: A B AxB C f_BC result

Node B

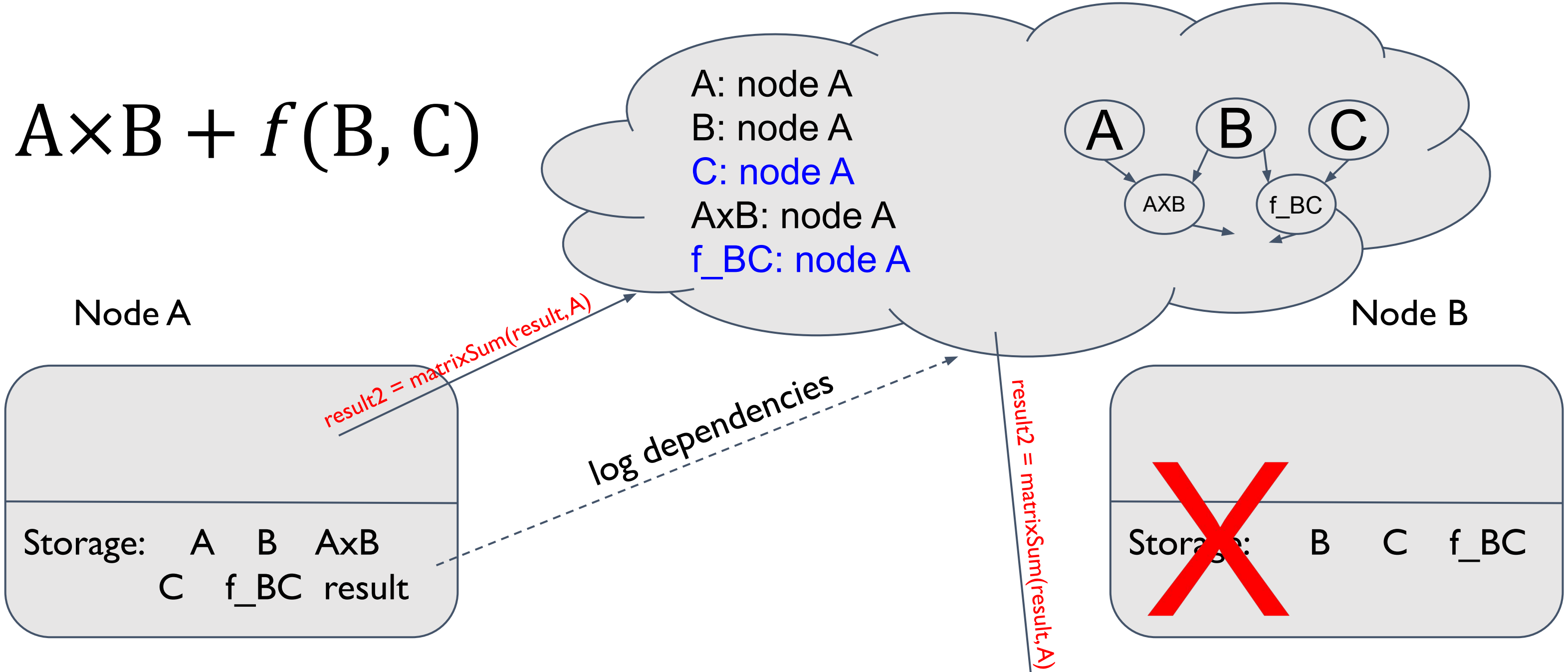
Storage: B C f_BC

Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
```

result2 = matrixSum(result, A)

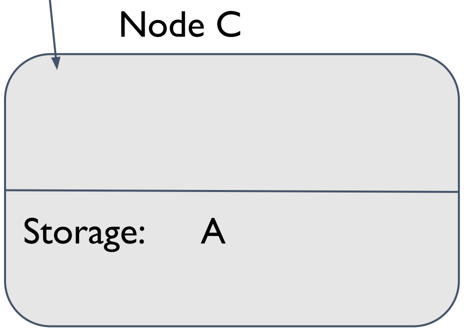
$$A \times B + f(B, C)$$



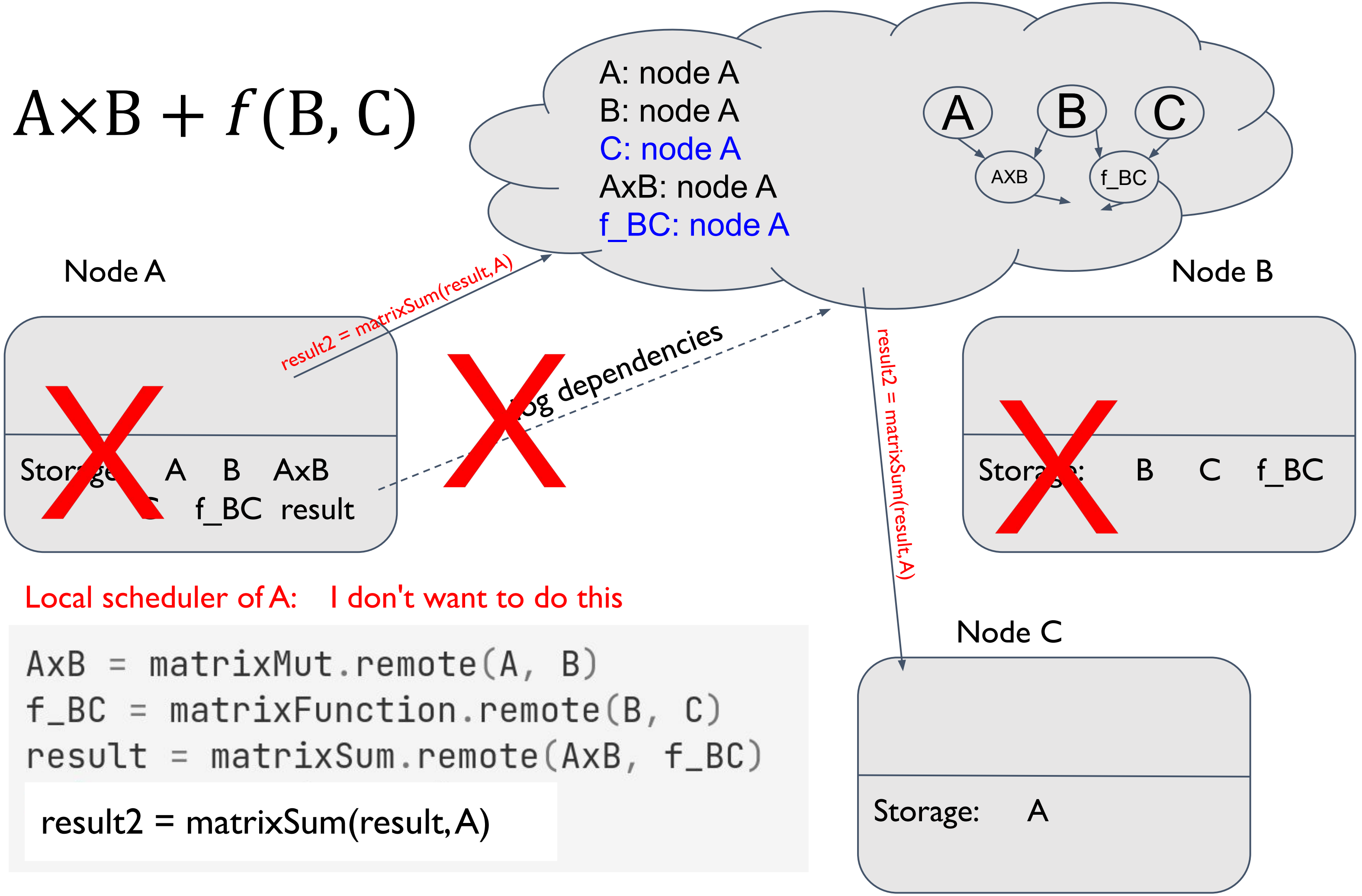
Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
```

result2 = matrixSum(result,A)



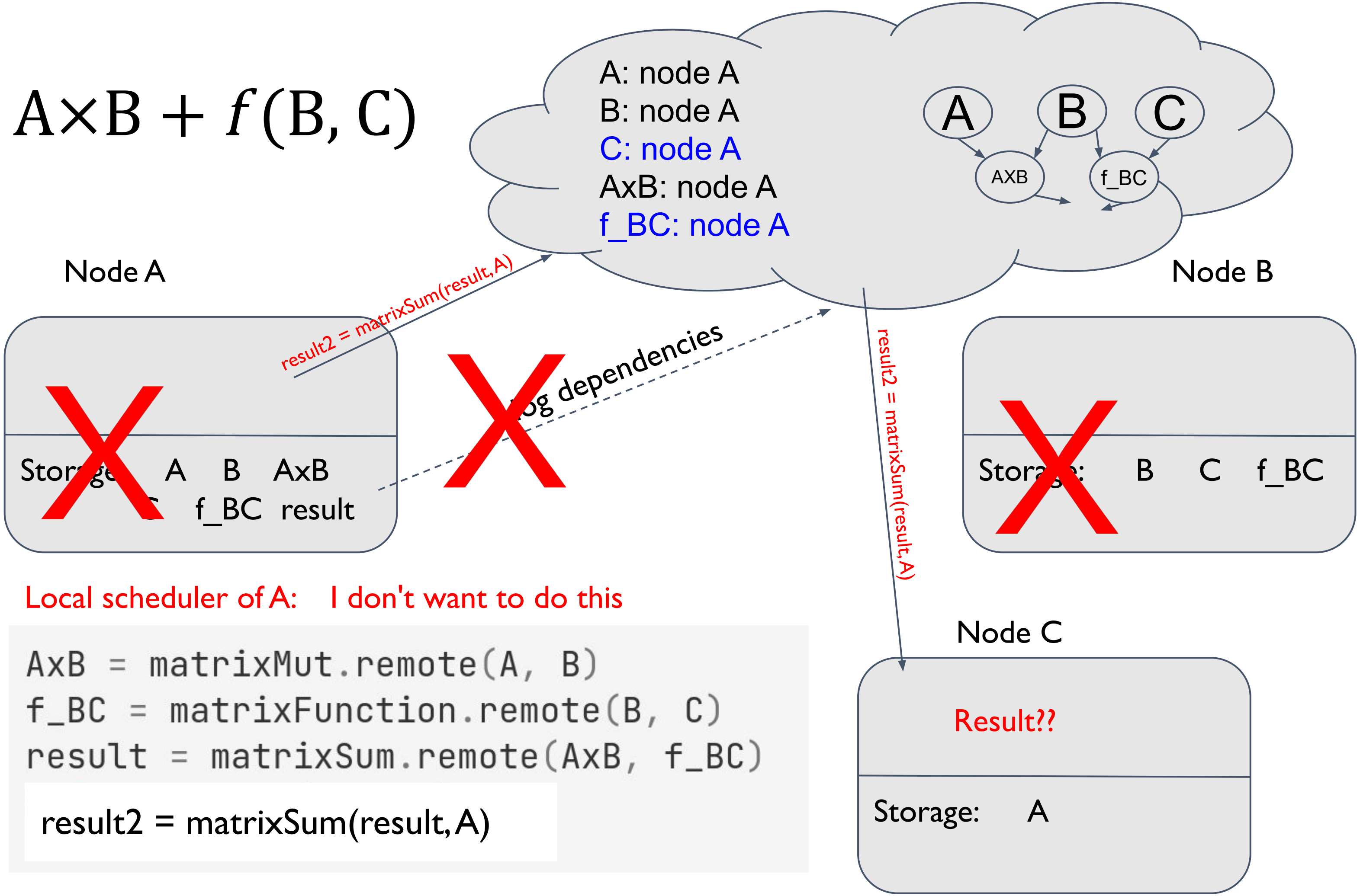
$$A \times B + f(B, C)$$



Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
result2 = matrixSum(result,A)
```

$$A \times B + f(B, C)$$



Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
```

result2 = matrixSum(result,A)

Lineage Logging

Lineage Logging

- Fast recovery time
- Large overhead
    - need to commit lineage before each task
    - need to wait for ack before starting each task

Lineage Logging

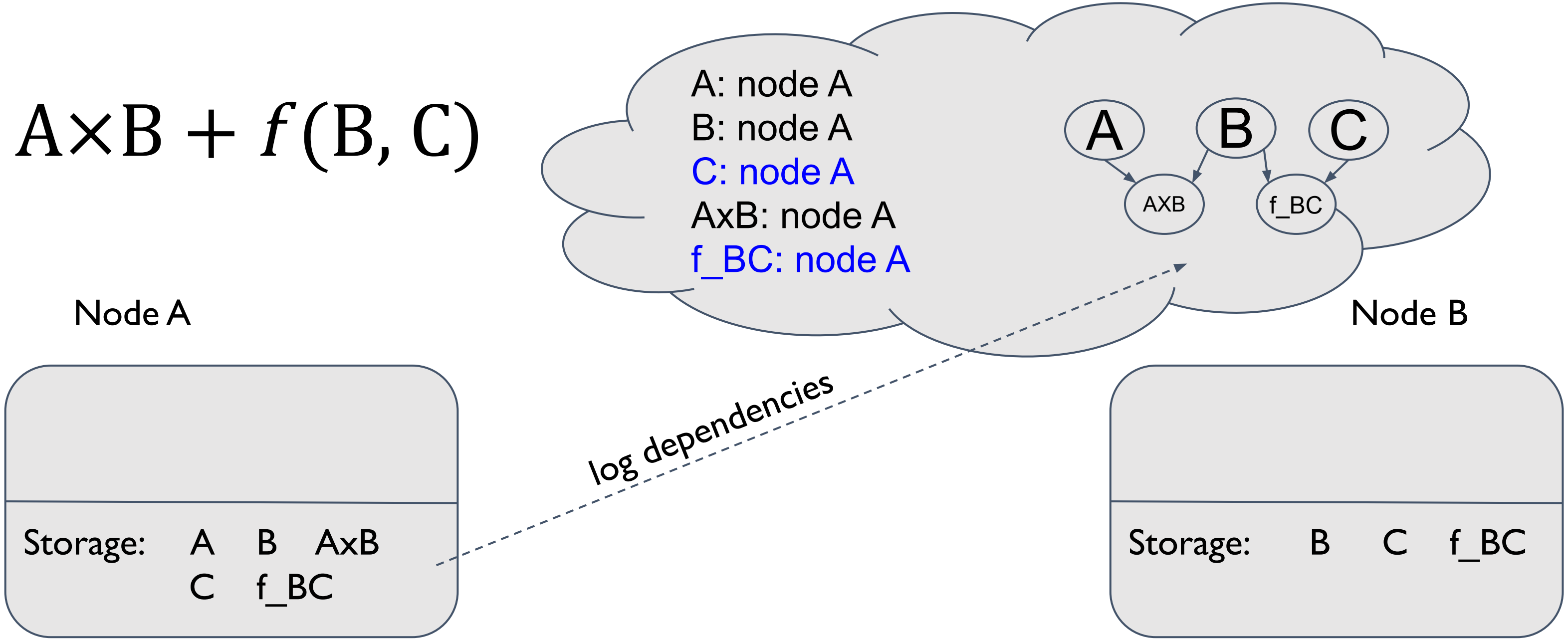
- Fast recovery time
- Large overhead
  - need to commit lineage before each task
  - need to wait for ack before starting each task
- Good for coarse grained tasks (seconds)
  - Big data

Lineage Stashing

Lineage Stashing

- low recovery time
- low overhead

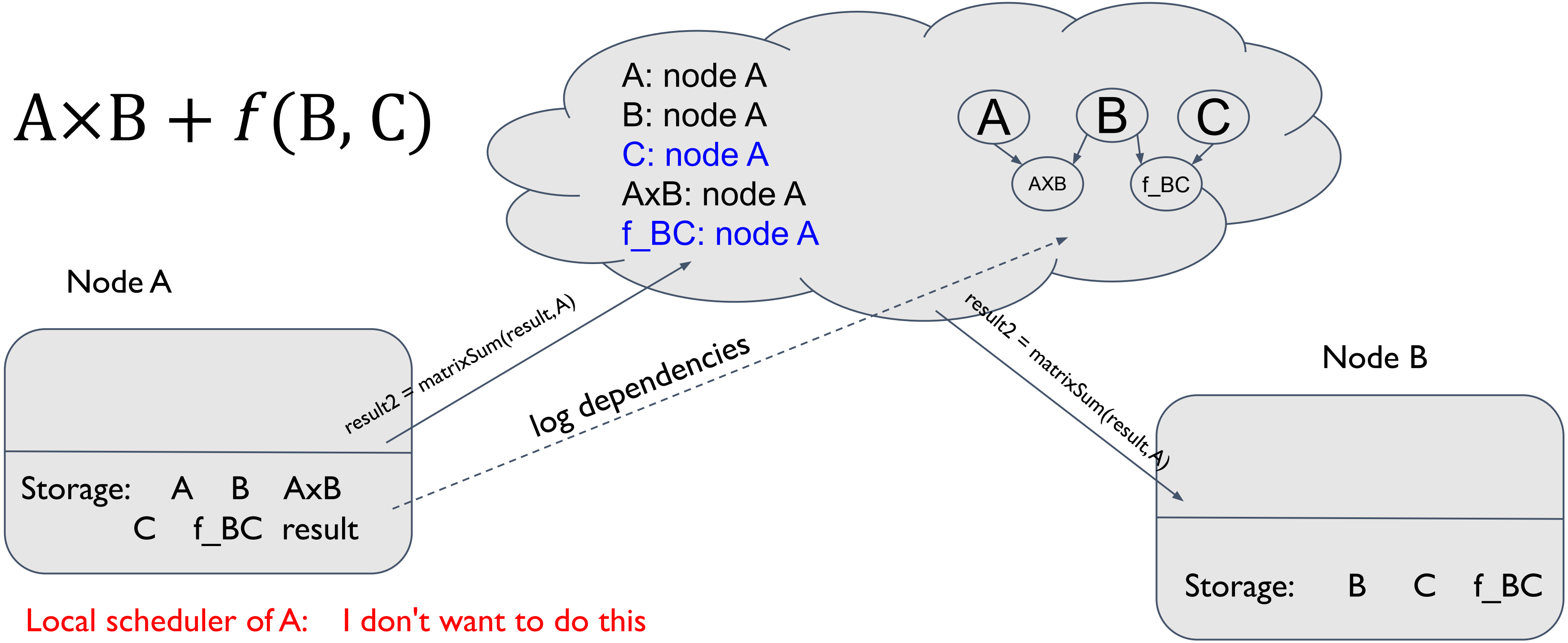
$$\mathrm{A} \times \mathrm{B} + f(\mathrm{B}, \mathrm{C})$$




```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
```

result2 = matrixSum(result, A)

$$\mathrm{A}\times\mathrm{B} + f(\mathrm{B},\mathrm{C})$$

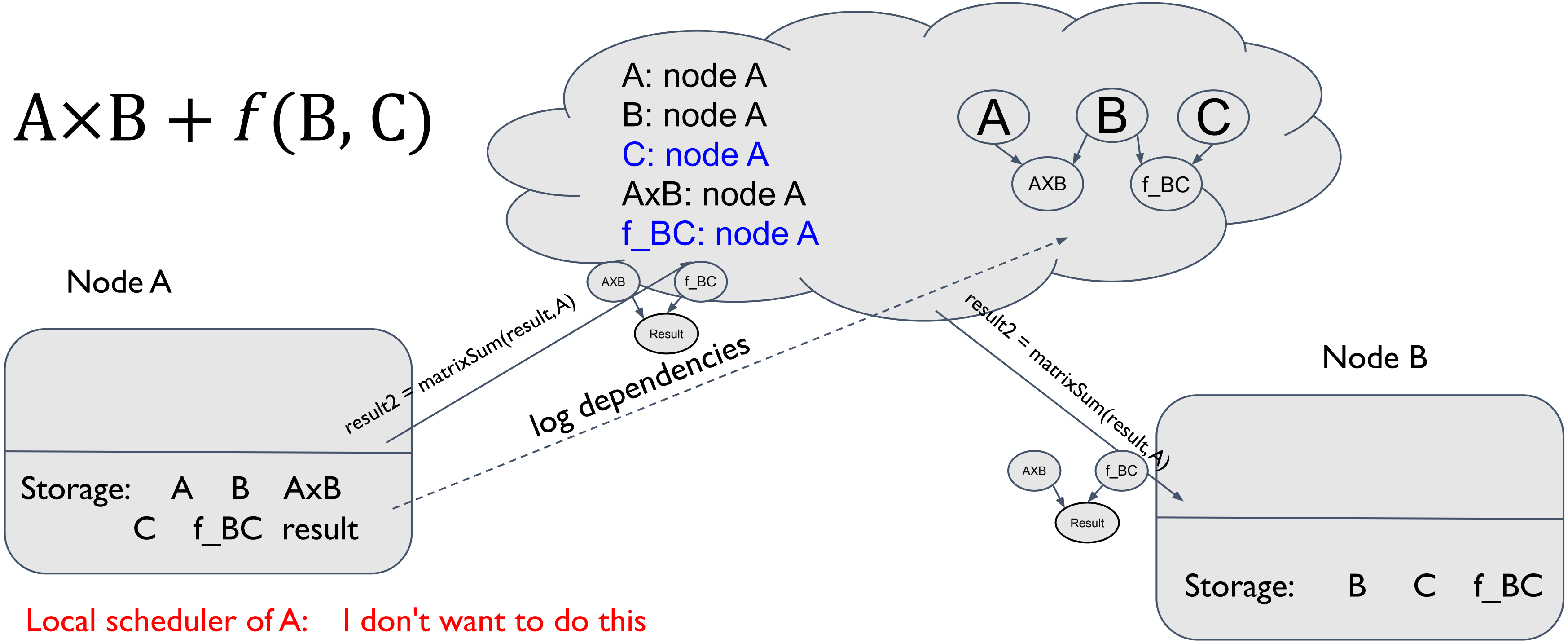


Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
```

result2 = matrixSum(result, A)

$$A \times B + f(B, C)$$

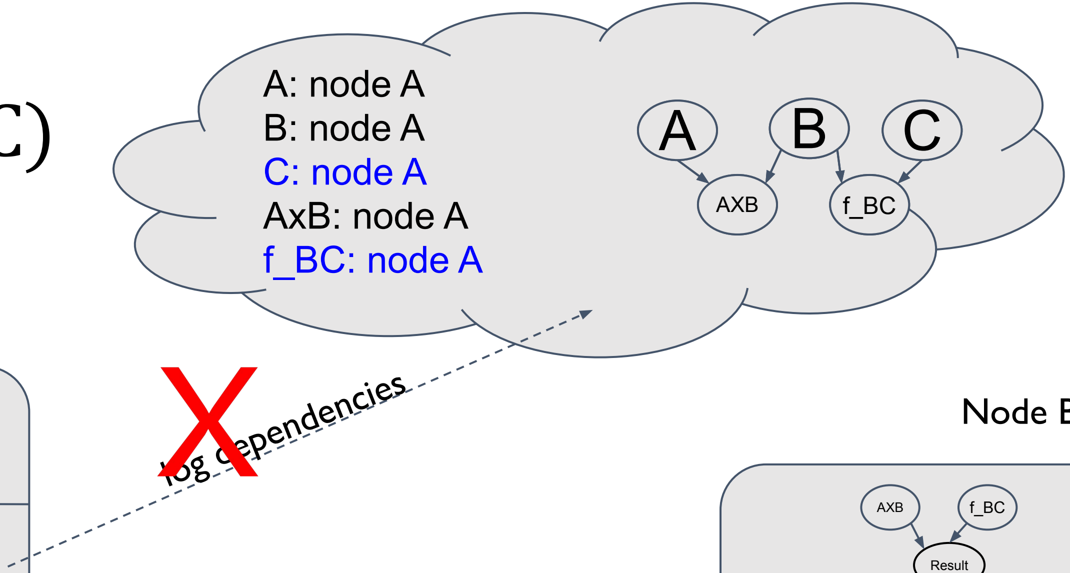


Local scheduler of A: I don't want to do this

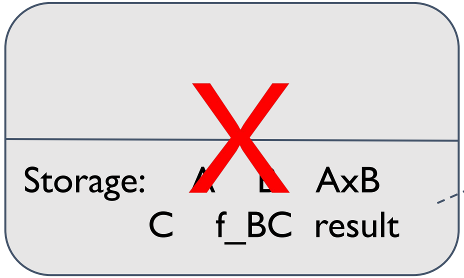
```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
```

result2 = matrixSum(result, A)

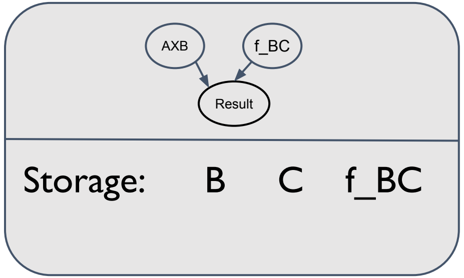
$$A\times B + f(B, C)$$



Node A



Node B



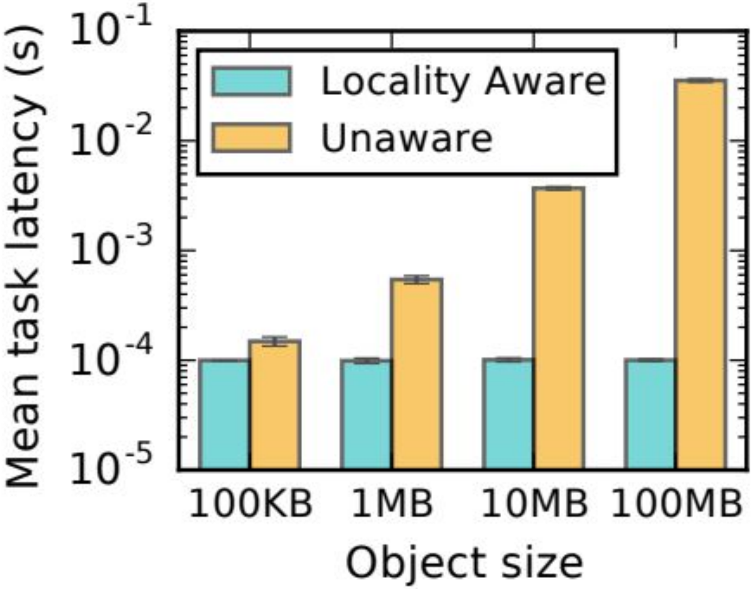
Local scheduler of A: I don't want to do this

```
AxB = matrixMut.remote(A, B)
f_BC = matrixFunction.remote(B, C)
result = matrixSum.remote(AxB, f_BC)
```

result2 = matrixSum(result,A)

Evaluation

Ray: Locality-aware task placement

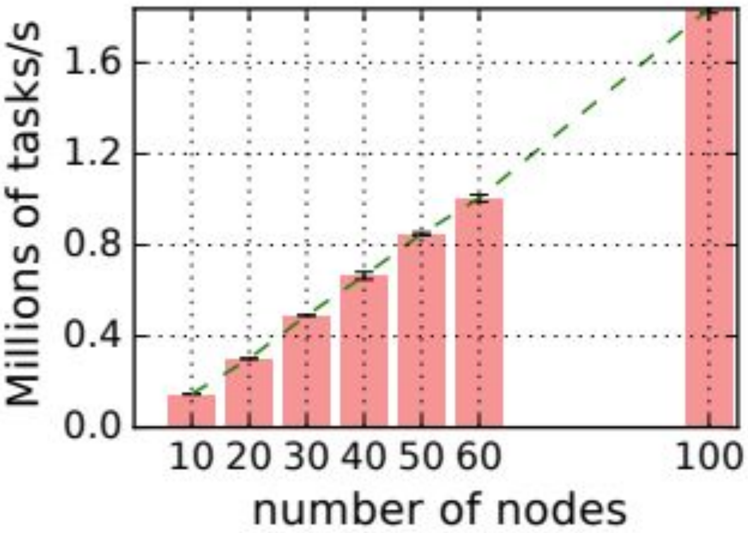


Ray: End-to-end scalability

60 nodes

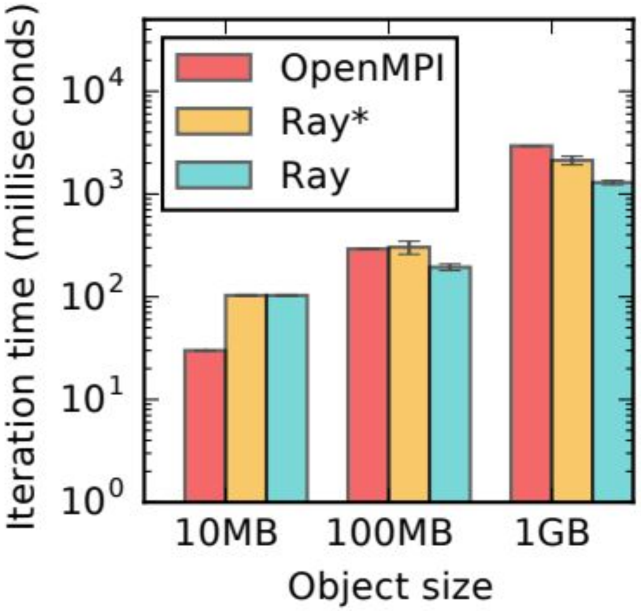
~1,000,000 tasks per sec

The design of GCS & bottom-up scheduler enables high horizontal scalability

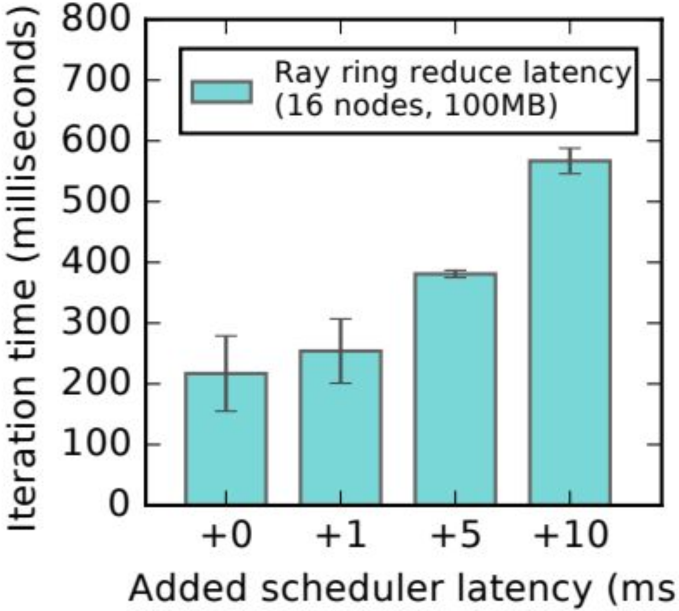


Evaluation: Allreduce

Scheduler performance is critical



Ray vs OpenMPI

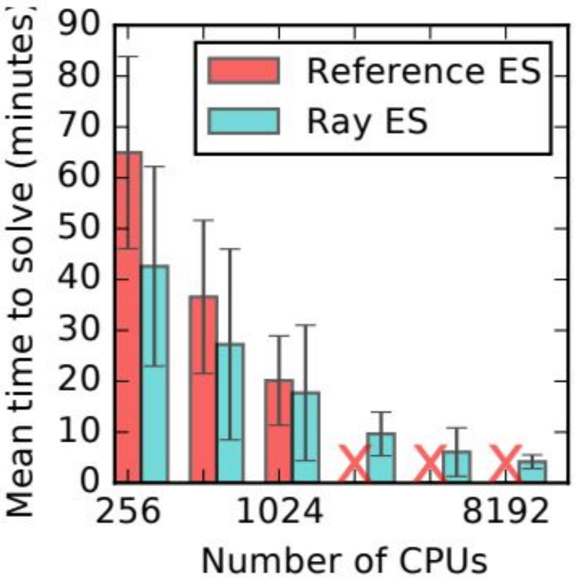


Ray scheduler ablation

Evaluation: RL Application

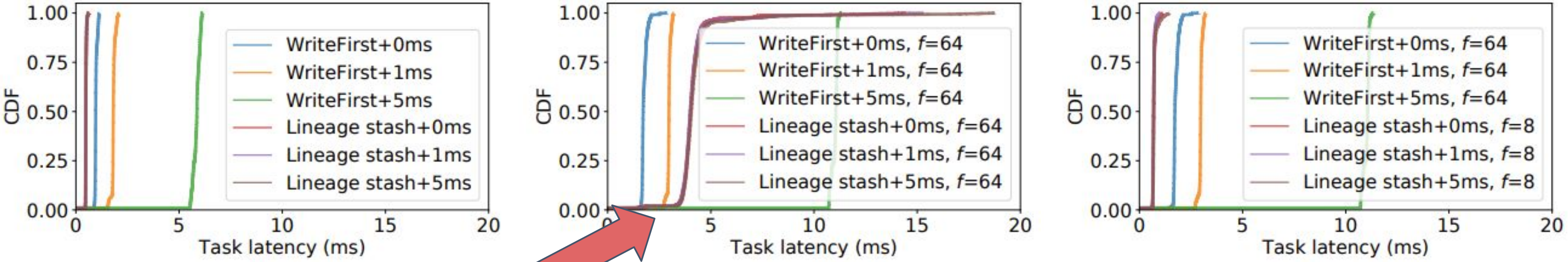
- The reference system fails to scale to 2048 cores, due the capacity of application driver
- Ray implementation uses an aggregation tree of actors

  2x cores ~ average 1.6x speedup



Evolution Strategies

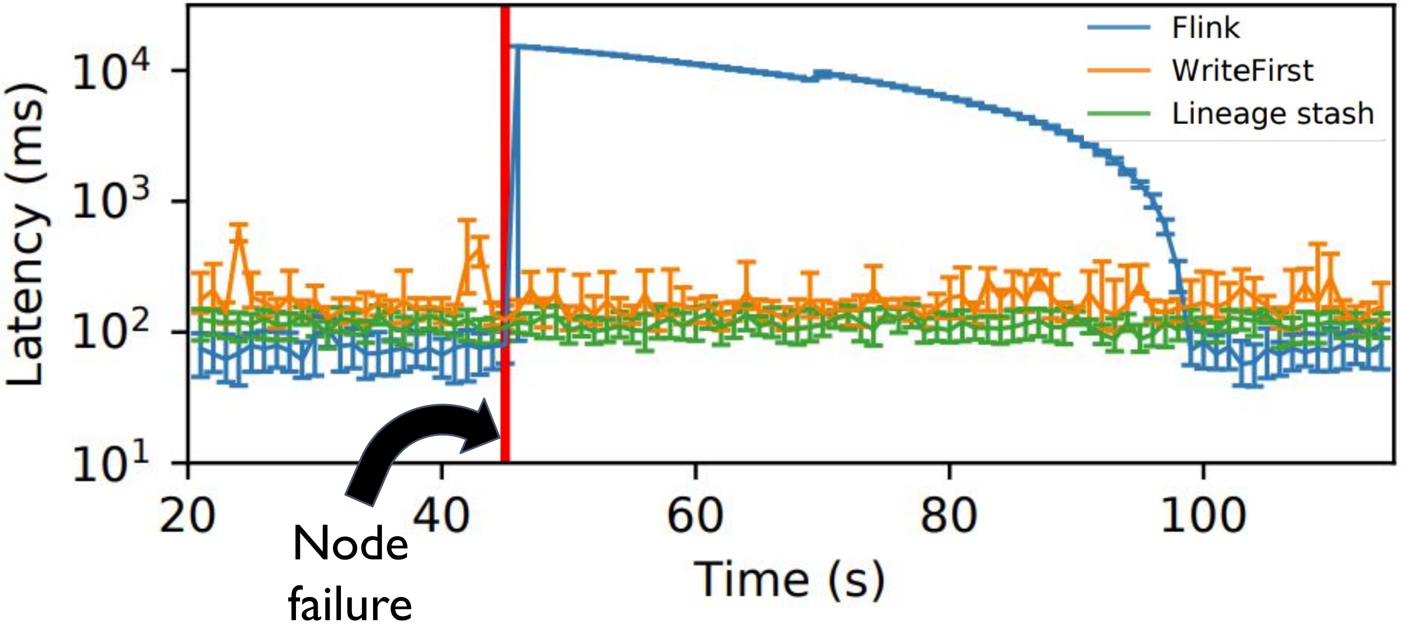
Lineage Stash: Fault tolerance for free?



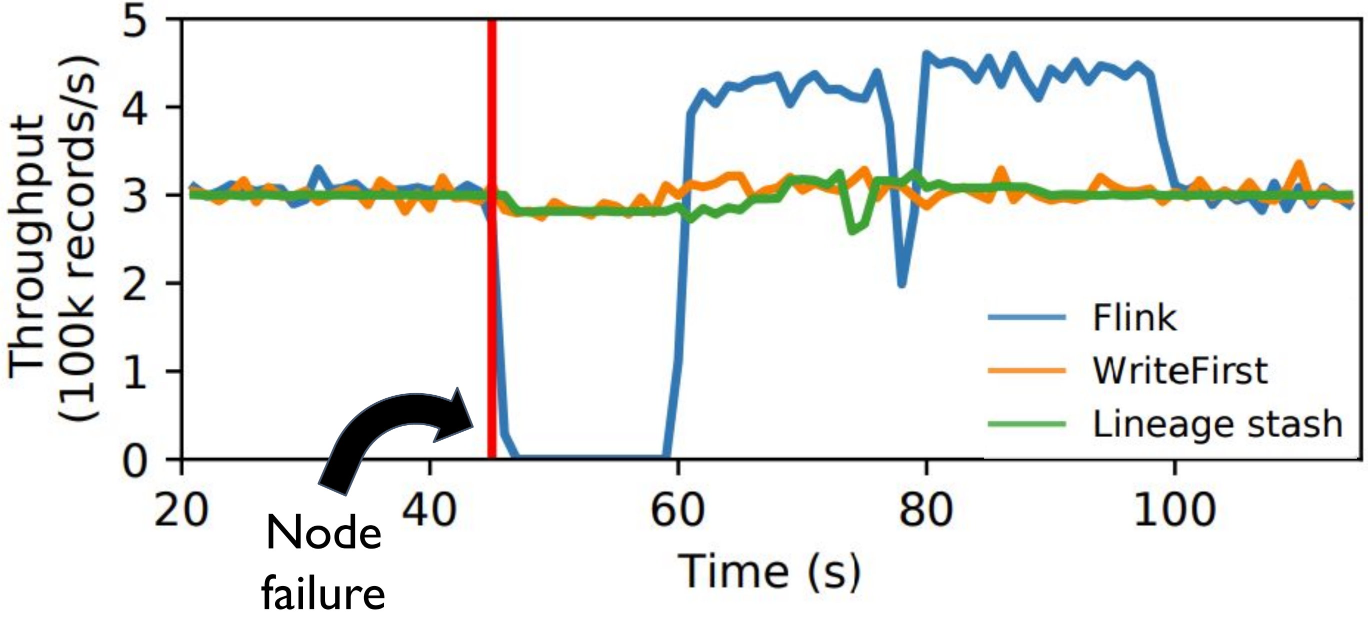
(a) Deterministic. **(b)** Nondeterministic, unlimited forwarding. **(c)** Nondeterministic, forward up to 8 nodes.

When uncommitted lineage grows too large, the performance of lineage stash will be greatly reduced

Lineage Stash: Latency during failure



Lineage Stash: Throughput during failure



Summary

Ray: a general-purpose system that

- supports training, serving and simulation efficiently
- unifies stateless (task) and stateful (actor) computations
- has high throughput, low latency and horizontal scalability

Lineage Stash:

asynchronously log the lineage and forward uncommitted lineage to guarantee recovery correctness