Command Horizons

Coalescing Data Dependencies while Maintaining Asynchronicity

Peter Thoman, Philip Salzmann
University of Innsbruck, Austria

Celerity
High-level C++ for Accelerator Clusters

DPS
Distributed and Parallel Systems

universität innsbruck

A Brief Introduction to Celerity

The Celerity Idea

- A **high-level API** designed from the ground up for **accelerator** clusters
  - Allows to constrain data structures and processing patterns to ones efficient on accelerators → less complex than fully general distributed memory programming
- Based on the SYCL Khronos industry standard
  - Can target most hardware supported by OpenCL via several SYCL platforms

- No explicit distribution, synchronization or communication
  - Derived entirely from data flow

Celerity – Jacobi Example (1/2)

```
using namespace sycl;
for(int i = 0; i < num_iterations; ++i) {
    queue.submit([=](celerity::handler& cgh) {
        auto nbr = celerity::access::neighborhood<2>{1, 1};
        auto o2o = celerity::access::one_to_one{};
        celerity::accessor in(in_buf, cgh, nbr, read_only);
        celerity::accessor out(out_buf, cgh, o2o, write_only, no_init);
        cgh.parallel_for<class Jacobi>(range<2>{N - 2, N - 2}, {1, 1},
          [=](item<2> itm) {
            const auto i = itm[0];
            const auto j = itm[1];
            out[{i, j}] = (in[{i, j - 1}] + in[{i, j + 1}] +
                           in[{i - 1, j}] + in[{i + 1, j}]) / 4.f;
          });
    });
    std::swap(in_buf, out_buf);
}
```

- **Buffers** encapsulate 1D-3D dense, typed data
  - Accesses are declared explicitly
- **Command groups** are submitted to the **distributed queue**
  - Tying kernels to the buffers they operate on
- **Kernels** execute over N-dimensional range of (virtual) threads

Celerity – Jacobi Example (2/2)

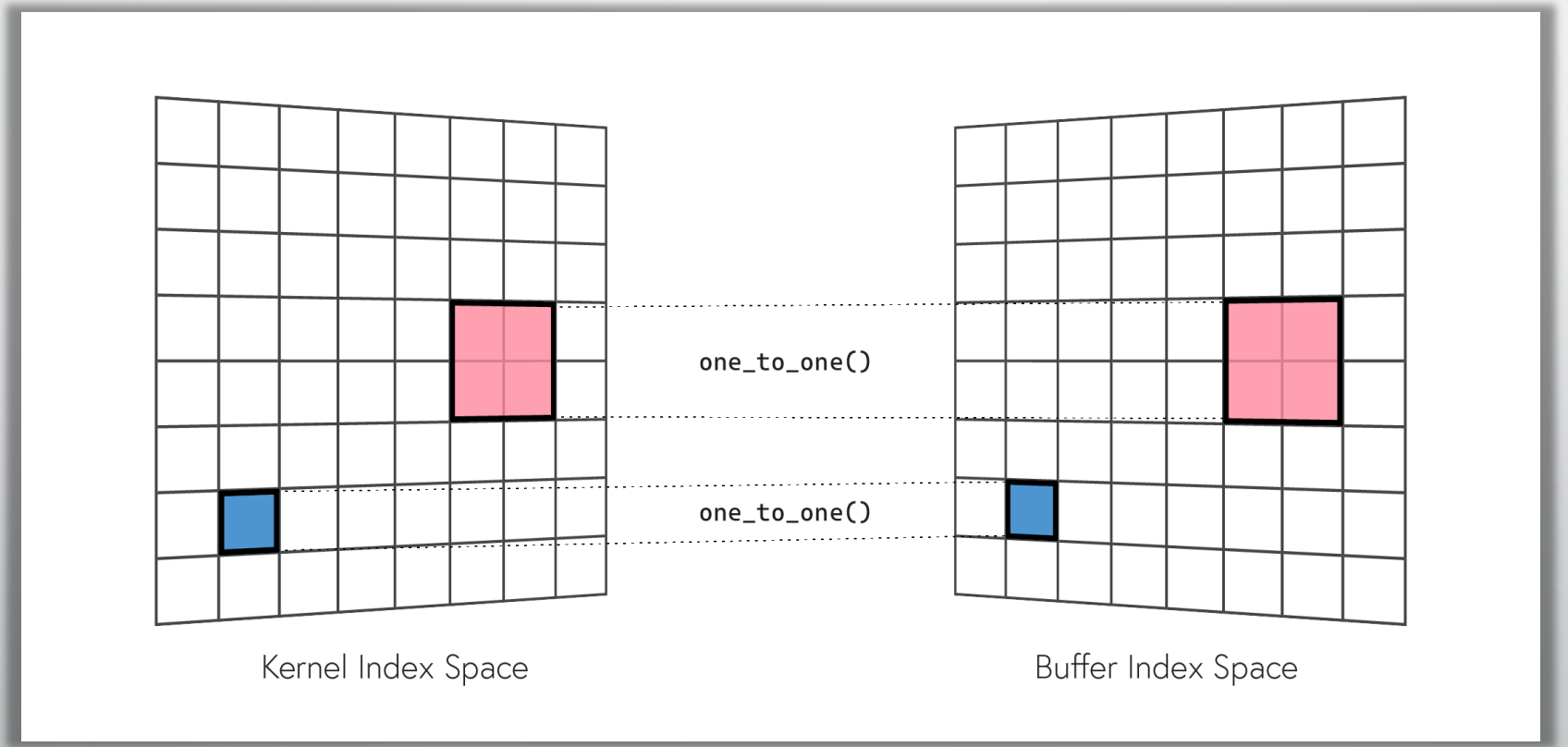
```
using namespace sycl;
for(int i = 0; i < num_iterations; ++i) {
    queue.submit([=](celerity::handler& cgh) {
        auto nbr = celerity::access::neighborhood<2>{1, 1};
        auto o2o = celerity::access::one_to_one{};
        celerity::accessor in(in_buf, cgh, nbr, read_only);
        celerity::accessor out(out_buf, cgh, o2o, write_only, no_init);
        cgh.parallel_for<class Jacobi>(range<2>{N - 2, N - 2}, {1, 1},
          [=](item<2> itm) {
            const auto i = itm[0];
            const auto j = itm[1];
            out[{i, j}] = (in[{i, j - 1}] + in[{i, j + 1}] +
                           in[{i - 1, j}] + in[{i + 1, j}]) / 4.f;
          });
    });
    std::swap(in_buf, out_buf);
}
```

- **Range mappers** declare mapping of kernel subranges to buffer subranges
  - Which data is required to compute *part* of the kernel
  - This allows **splitting** of tasks

- This program **can run on an arbitrary number of nodes**
  - Distributed memory portion is almost completely hidden from the user

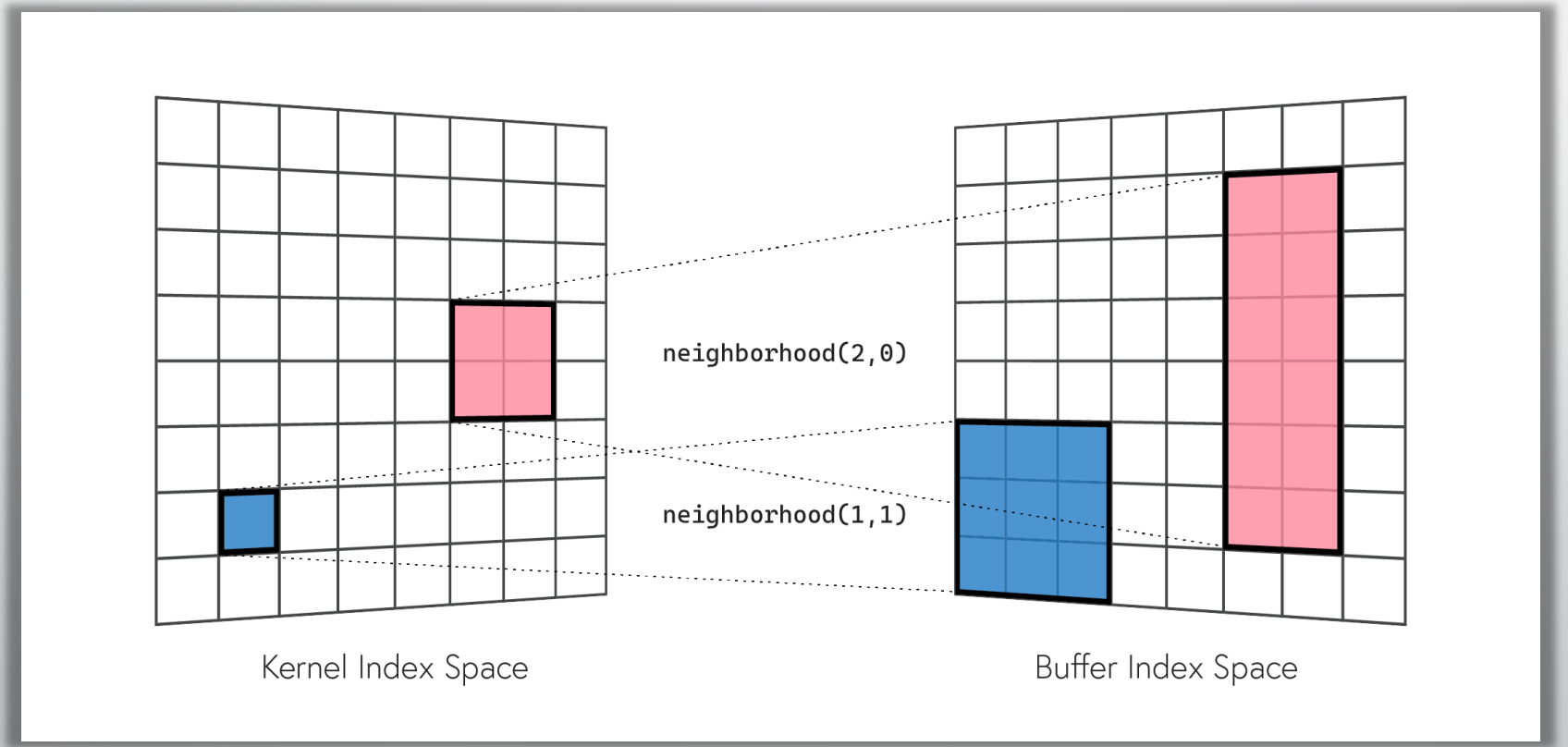
Celerity – Range Mappers

- Arbitrary functors mapping from a K-dimensional kernel index space `chunk` to a B-dimensional buffer index space `subrange`

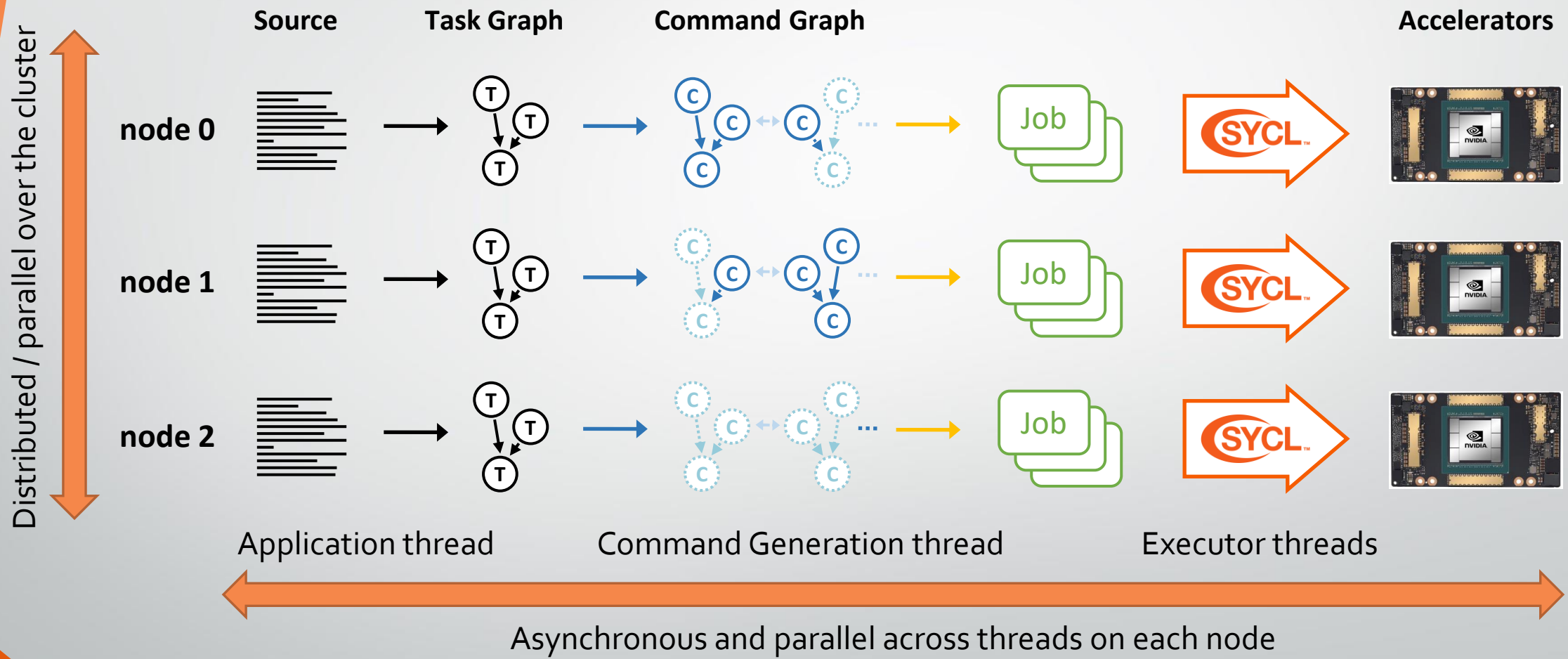


Celerity – Range Mappers

- Arbitrary functors mapping from a K-dimensional kernel index space `chunk` to a B-dimensional buffer index space `subrange`



Internal Architecture

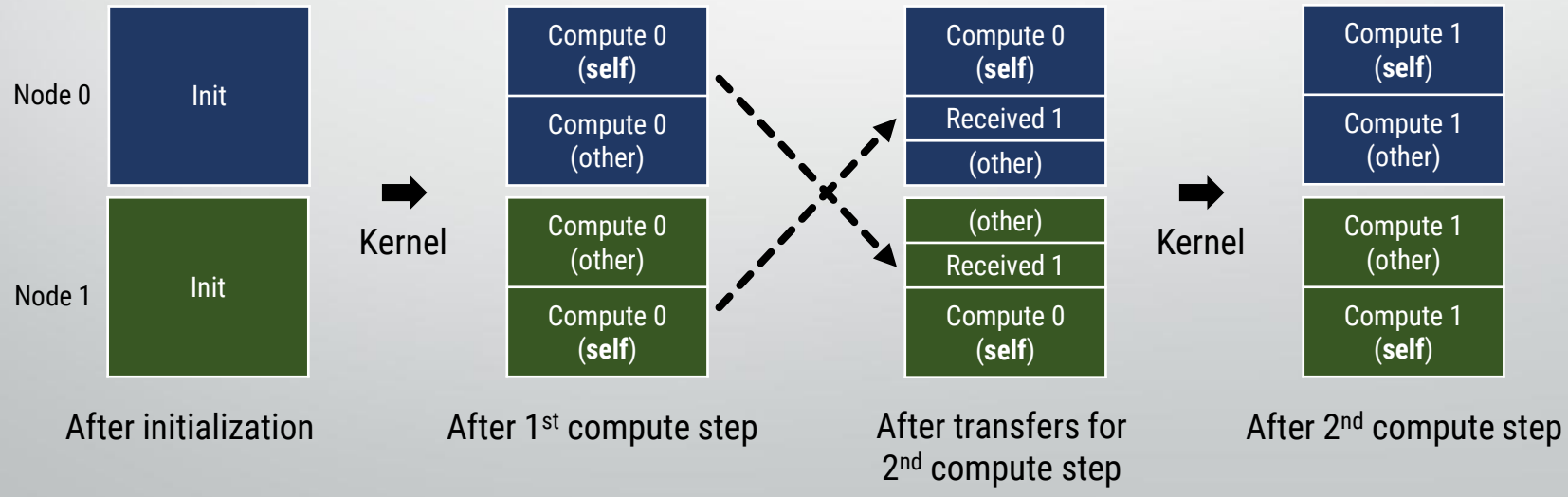


* Distributed command generation version of Celerity

Data Tracking

- In order to generate the command graph, Celerity needs to track which command last updated which location(s) for each buffer
  - More specifically, *will have updated* at the point of time currently being generated, as command generation runs significantly ahead of execution

(Simplified) example of tracking information over time for a 2D stencil on 2 nodes:

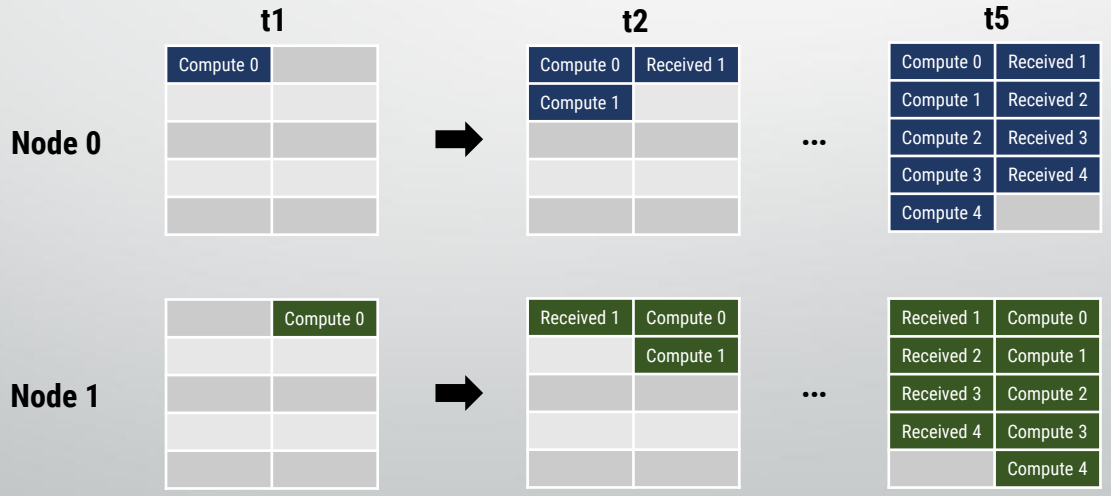


Generative Access Patterns

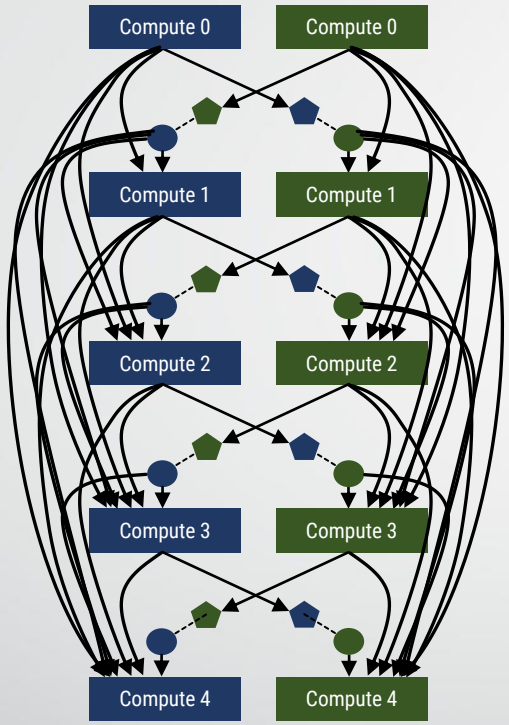
Room Response Simulator Access Pattern

- Example of a 2D generative access pattern
- **1 new row** of a 2D buffer is **written** every time step
- **All previous rows** are **read**

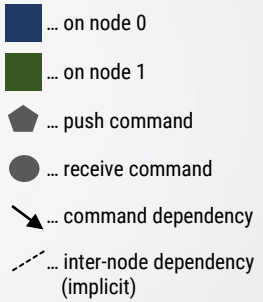
→ What does this mean for data tracking?



Impact on Command Graph Generation



Compute 0			Compute 0
Compute 0	Received 1	Received 1	Compute 0
Compute 1			Compute 1
Compute 0	Received 1	Received 1	Compute 0
Compute 1	Received 2	Received 2	Compute 1
Compute 2			Compute 2
Compute 0	Received 1	Received 1	Compute 0
Compute 1	Received 2	Received 2	Compute 1
Compute 2	Received 3	Received 3	Compute 2
Compute 3			Compute 3
Compute 0	Received 1	Received 1	Compute 0
Compute 1	Received 2	Received 2	Compute 1
Compute 2	Received 3	Received 3	Compute 2
Compute 3	Received 4	Received 4	Compute 3
Compute 4			Compute 4



→ Computational effort of dependency tracking and generation grows with algorithm iterations!

Command Horizons

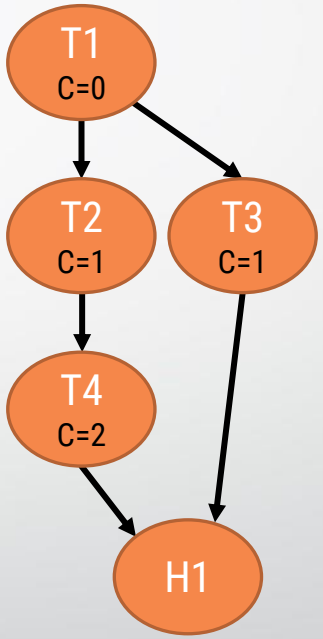
Horizons Overview

- *Goal*: solve the generative access patterns tracking issue
  - **Asynchronously**, and without additional communication
  - With a **configurable tradeoff** between tracking fidelity and overhead

- 3 important concepts:
  1. **Decision Making** – when to create a new Horizon
  2. **Horizon Generation** – what happens to the command graph when a Horizon is created
  3. **Horizon Application** – effect on tracking data structures when a Horizon is applied

Decision Making

- Simple Approach:
  - Track the critical path length while generating the task graph
    - Computationally very inexpensive
  - Every time a multiple of S is reached for the first time, generate a Horizon task
    - We call S the *Horizon Step Size*



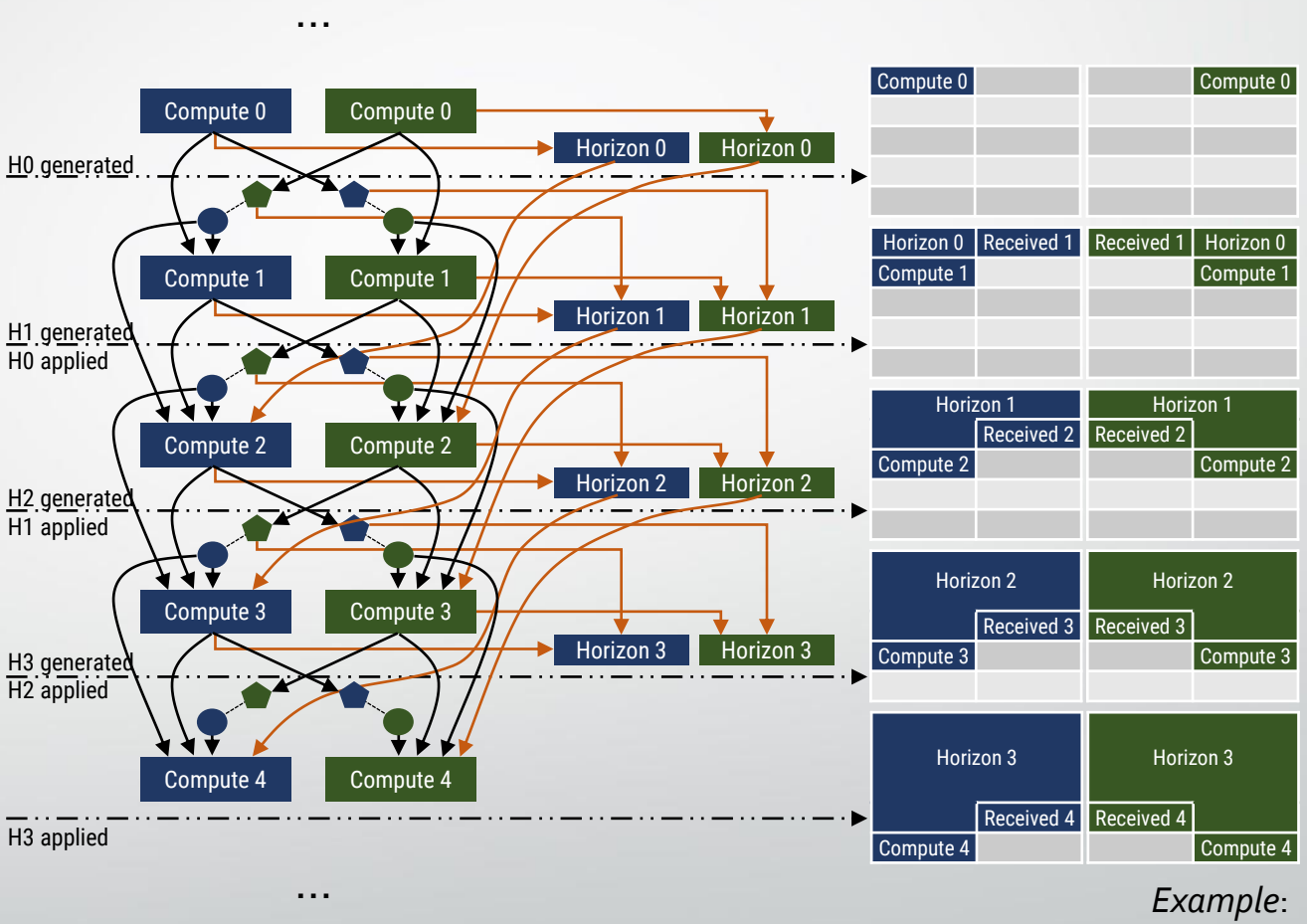
Example $S = 2$

Horizon Generation and Application

Horizon 0 ***generated***:
→ Dependencies from current command front (tracked during graph gen)

Horizon 0 ***applied***:
- Application of Hor. N-1 when Hor. N is generated

→ Subsumes all older (id < Hor.) entries in tracking data structure
→ Subsequent dependencies will be redirected to Horizon

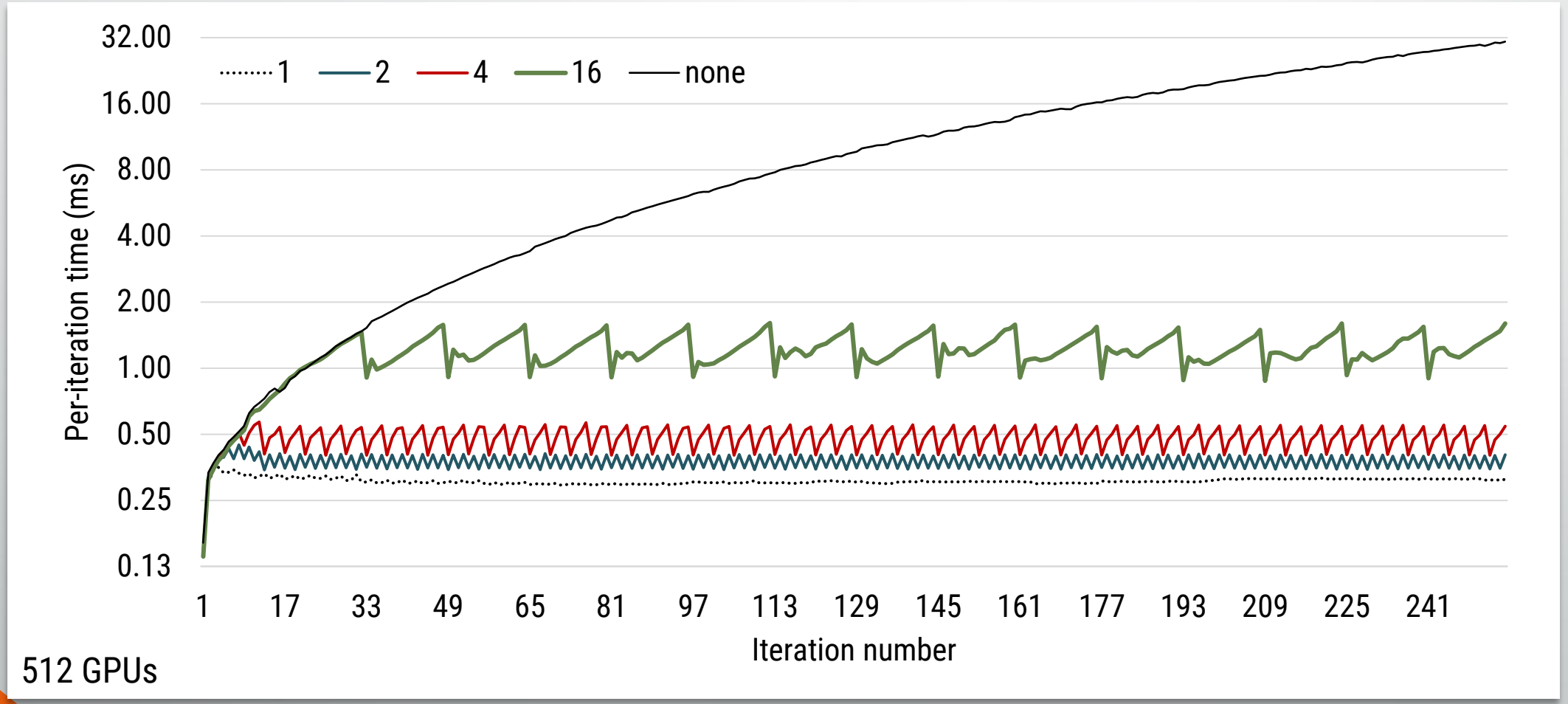


Example:
RSIM pattern
$S = 1$
Simplified

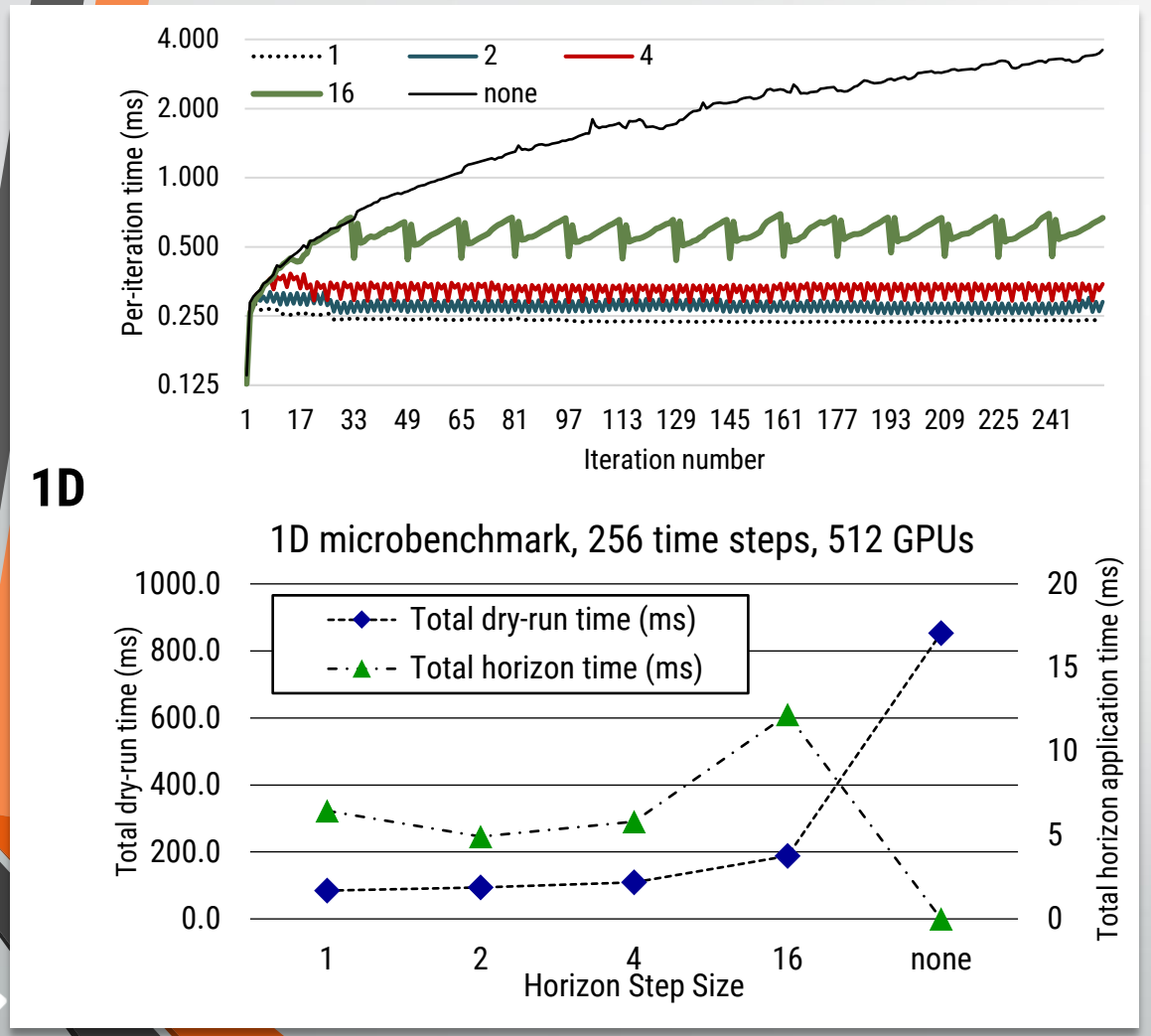
→ Fine-grained dependencies in local context
Dependencies can cross exactly one Horizon boundary

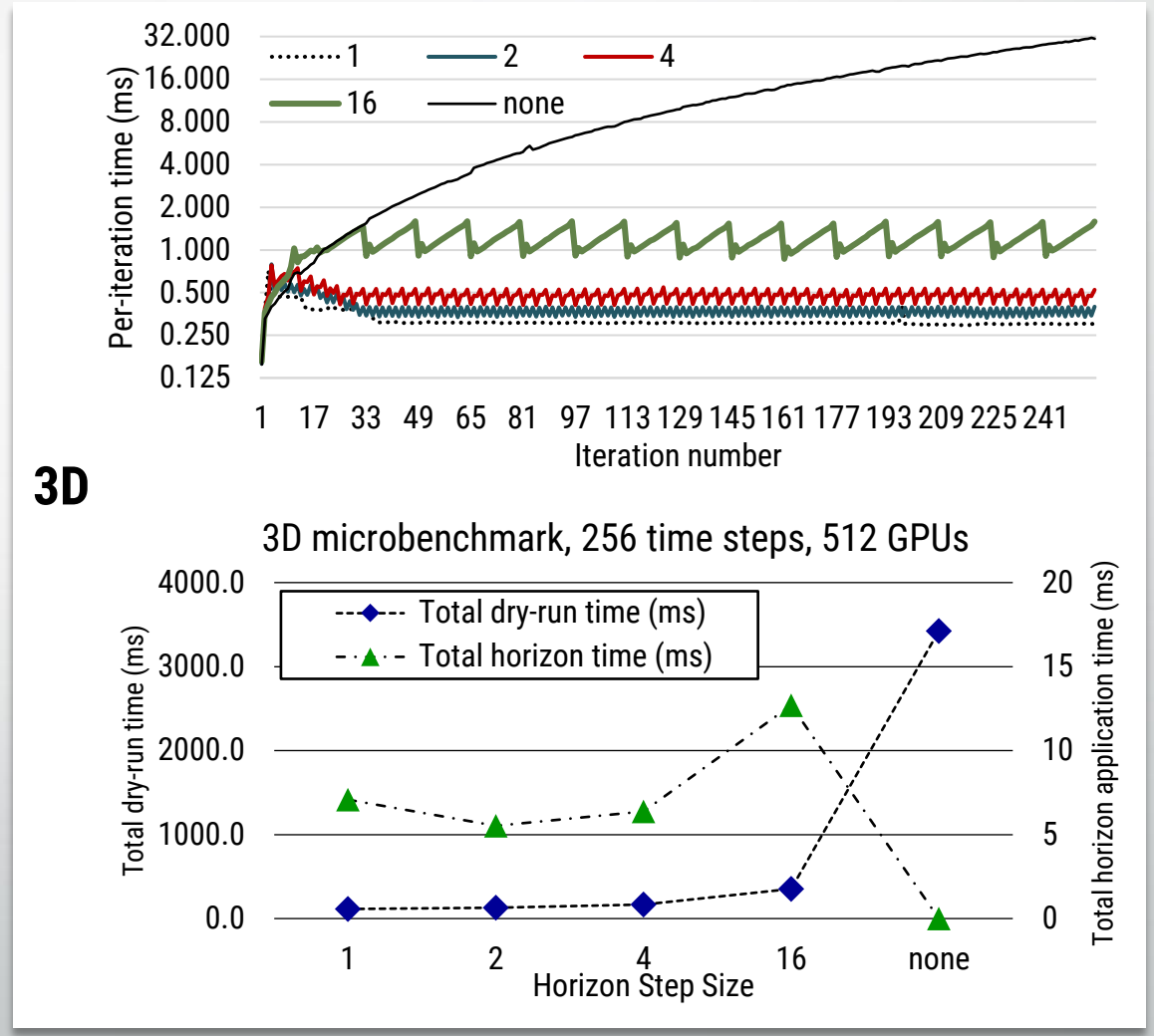
Performance Evaluation

Microbenchmarks – 2D Generative Access

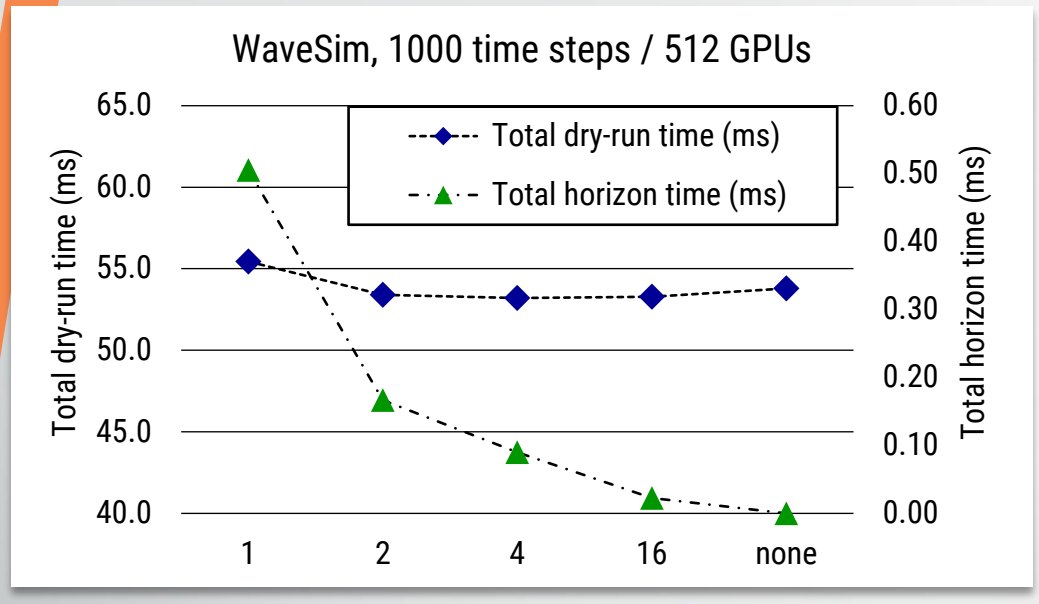


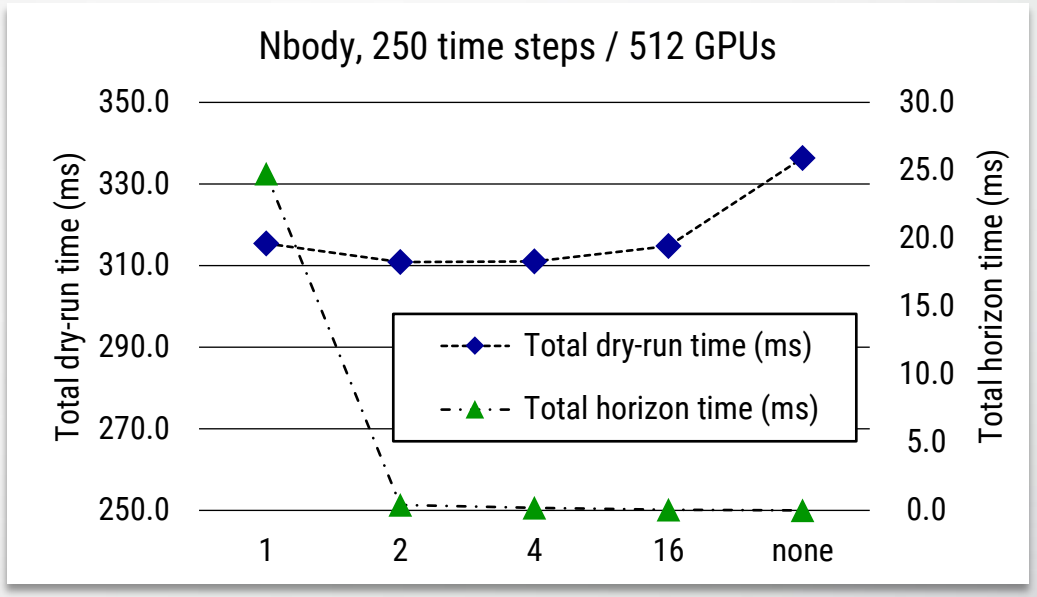
Additional Microbenchmarks





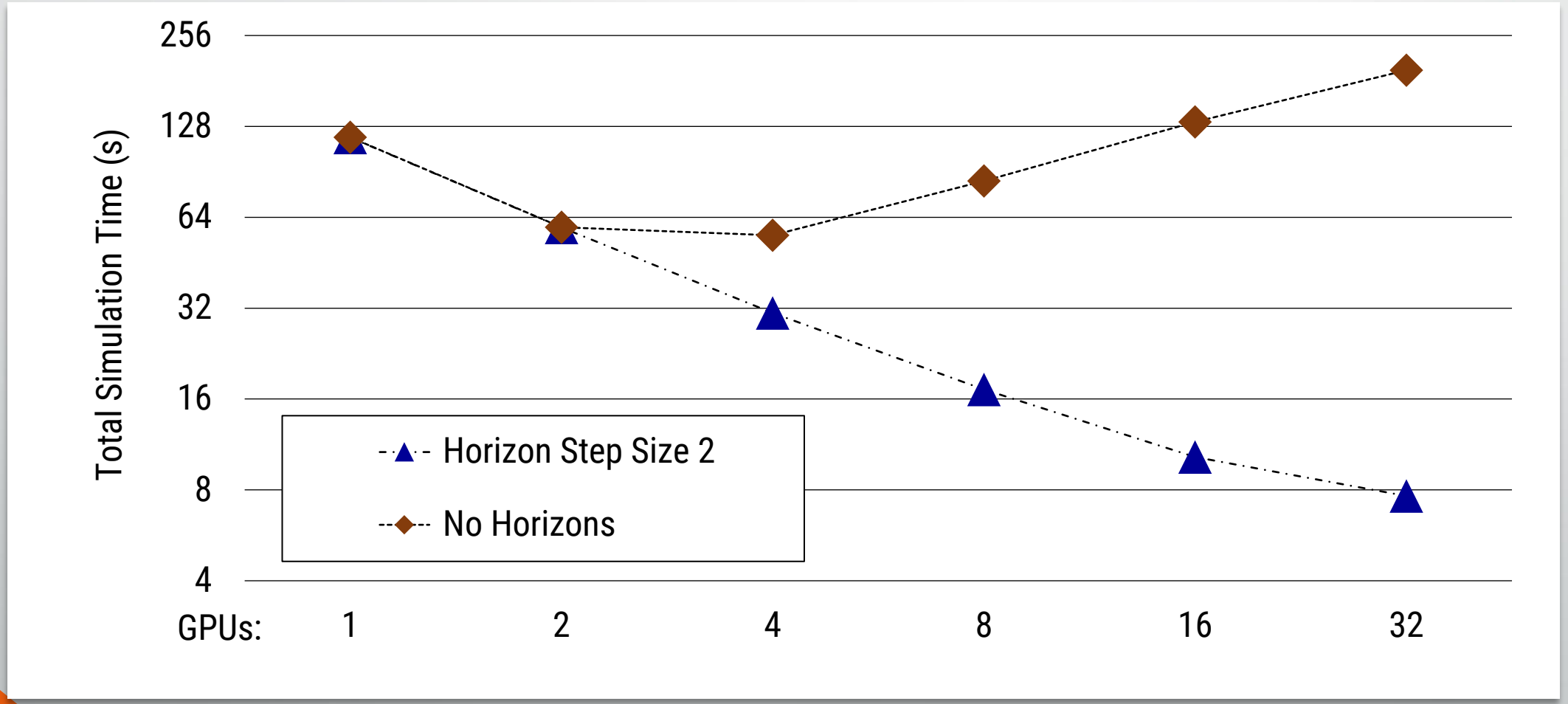
Overhead on Non-generative Applications





- Horizon overhead is negligible
  - Recall that this is entirely asynchronous to actual computation!
  - $S = 1$ for Nbody, a degenerate case
- Horizons actually have a minor positive impact even for non-generative apps
  - Related to data structure cleanup

Real-World RSIM Evaluation



Horizons Summary

- *Advantages*:
  - **Independent** of the specifics of the data **access pattern**
  - **Caps** the **per-node dependencies** which need to be tracked
  - **High-fidelity** dependency information is maintained **locally**
  - **Generation is efficient** – required information can be tracked with a small fixed overhead during command generation
  - **Application is efficient** – due to the numbering scheme of commands no graph traversal is required
  - **No additional communication** is required
- *Potential downside*:
  - Independent commands might be sequentialized → No impact in practice with $S \geq 2$

Thank you for your attention!

Questions?

https://celerity.github.io

https://discord.gg/k8vWTPB

Celerity

High-level C++ for Accelerator Clusters

DPS

Distributed and Parallel Systems

universität innsbruck