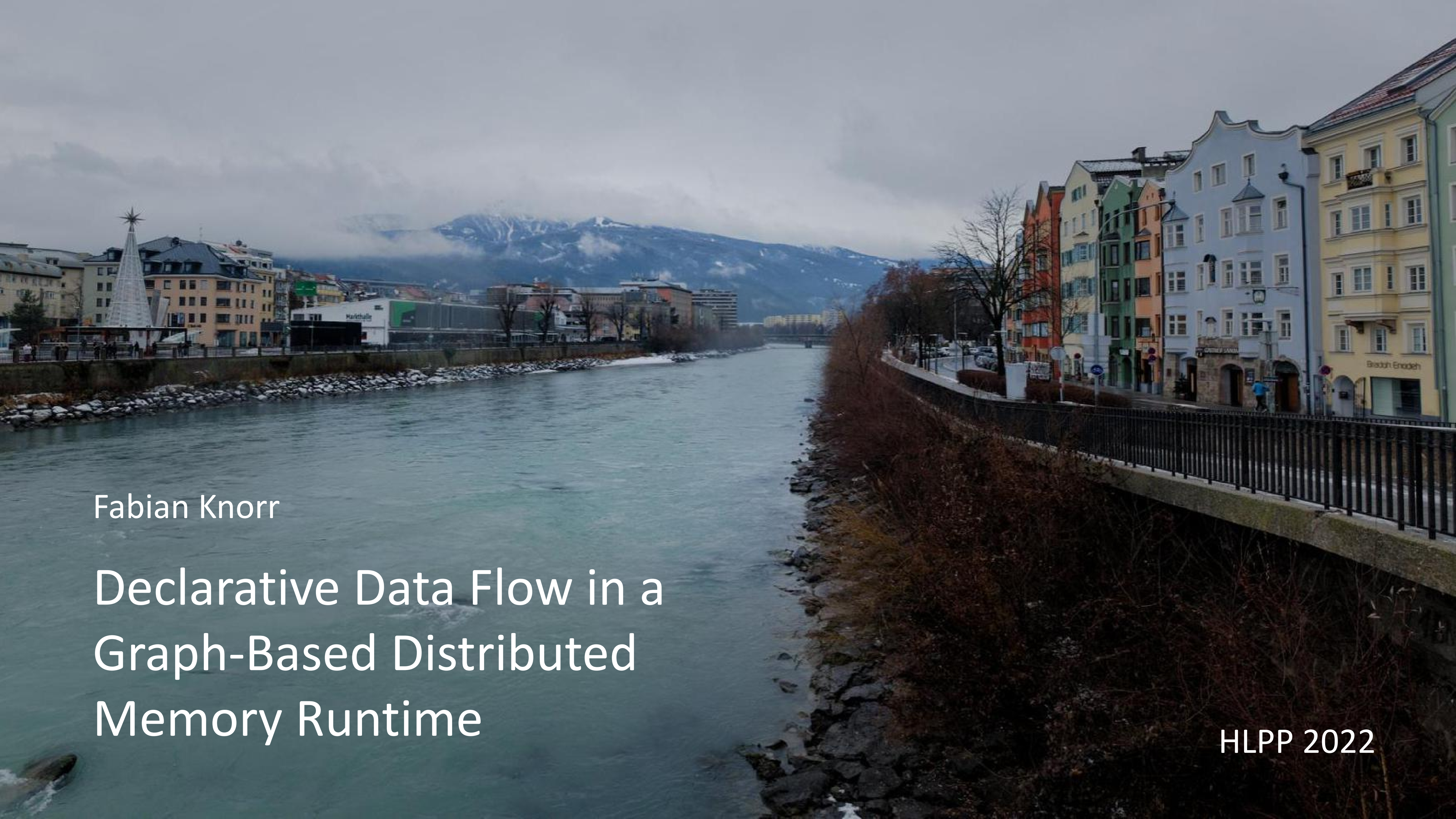


Fabian Knorr

Declarative Data Flow in a Graph-Based Distributed Memory Runtime

HLPP 2022

SYCL™ ... in High Performance Computing

SYCL is an established C++ DSL for accelerator programming with a **high-level dataflow API**.

```
int main() {
    sycl::buffer<float, 2> buf{range<2>{1024, 1024}};          ①
    sycl::queue{}.submit([&](handler &chg) {
         sycl::accessor a{buf, cgh, write_only, no_init};              ②
         cgh.parallel_for(range<2>{1024, 1024}, [=](item<2> it) {
             a[it] = sin(it[0] / 100) * sin(it[1] / 100);              ③
         });
     });
}
```

① Buffers manage host and device allocations

② Accessors restrict buffer access to inside ③ kernels and enable dependency tracking

⟹ SYCL runtime automates data migration and scheduling based on dependency information

Celerity: Extending the SYCL API to Accelerator Clusters

High-level C++ for Accelerator Clusters

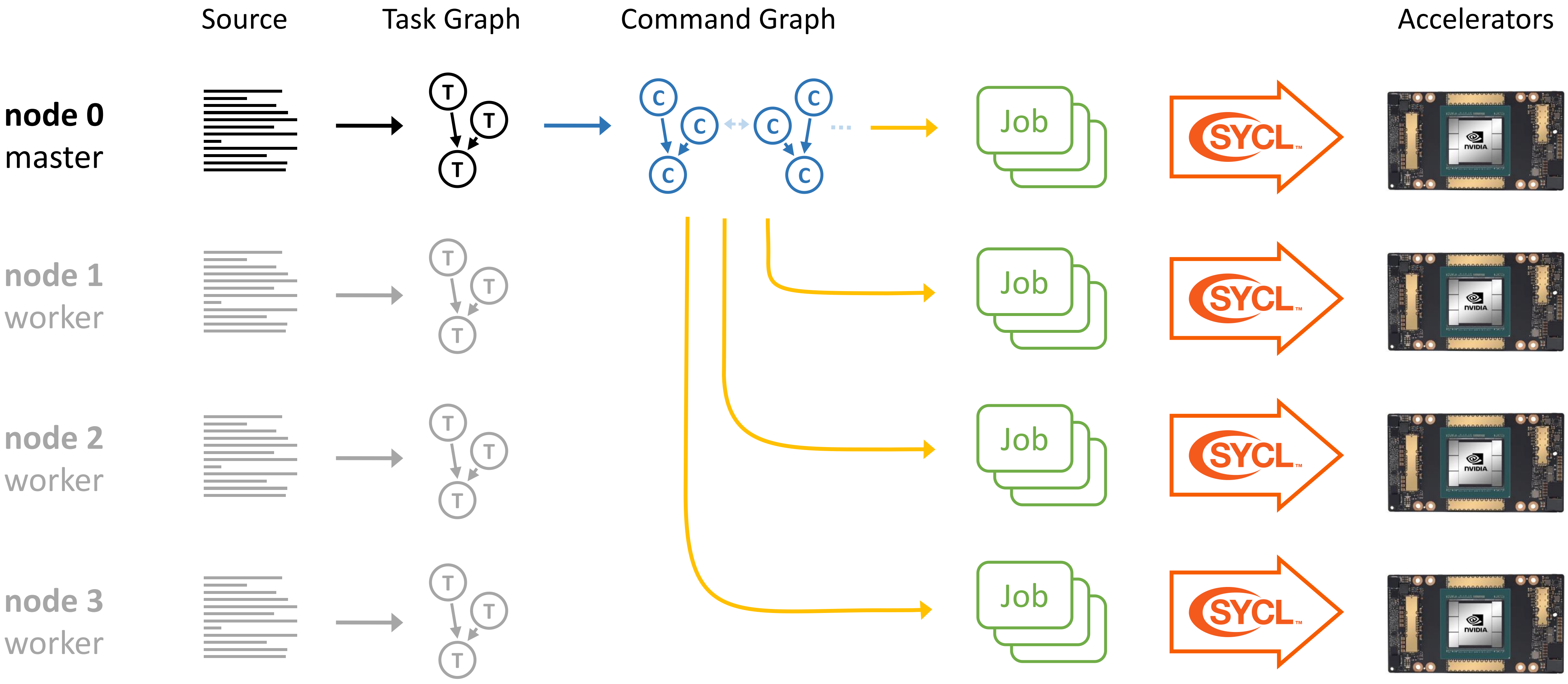
Kernels are transparently distributed onto MPI ranks by splitting ① execution ranges.

```
int main() {
    celerity::buffer<float, 2> buf{range<2>{1024, 1024}};
    celerity::distr_queue{}.submit([&](handler &chg) {                       ③
   /    celerity::accessor a{buf, cgh, write_only, celerity::one_to_one{}, no_init};
 ②      cgh.parallel_for(range<2>{1024, 1024}, [=](item<2> it) {
            a[it] = sin(it[0] / 100) * sin(it[1] / 100);
        });
     });                                             ①
}
```

Switching to the ② Celerity API requires augmenting accessors with ③ Range Mappers.
They inform Celerity which buffer subrange is accessed by which chunk of the iteration space.

From this, the runtime generates **MPI data transfers** to satisfy data requirements.

Excursion: Celerity Architecture



Source

Task Graph

Command Graph

Accelerators

node 0
master

node 1
worker

node 2
worker

node 3
worker

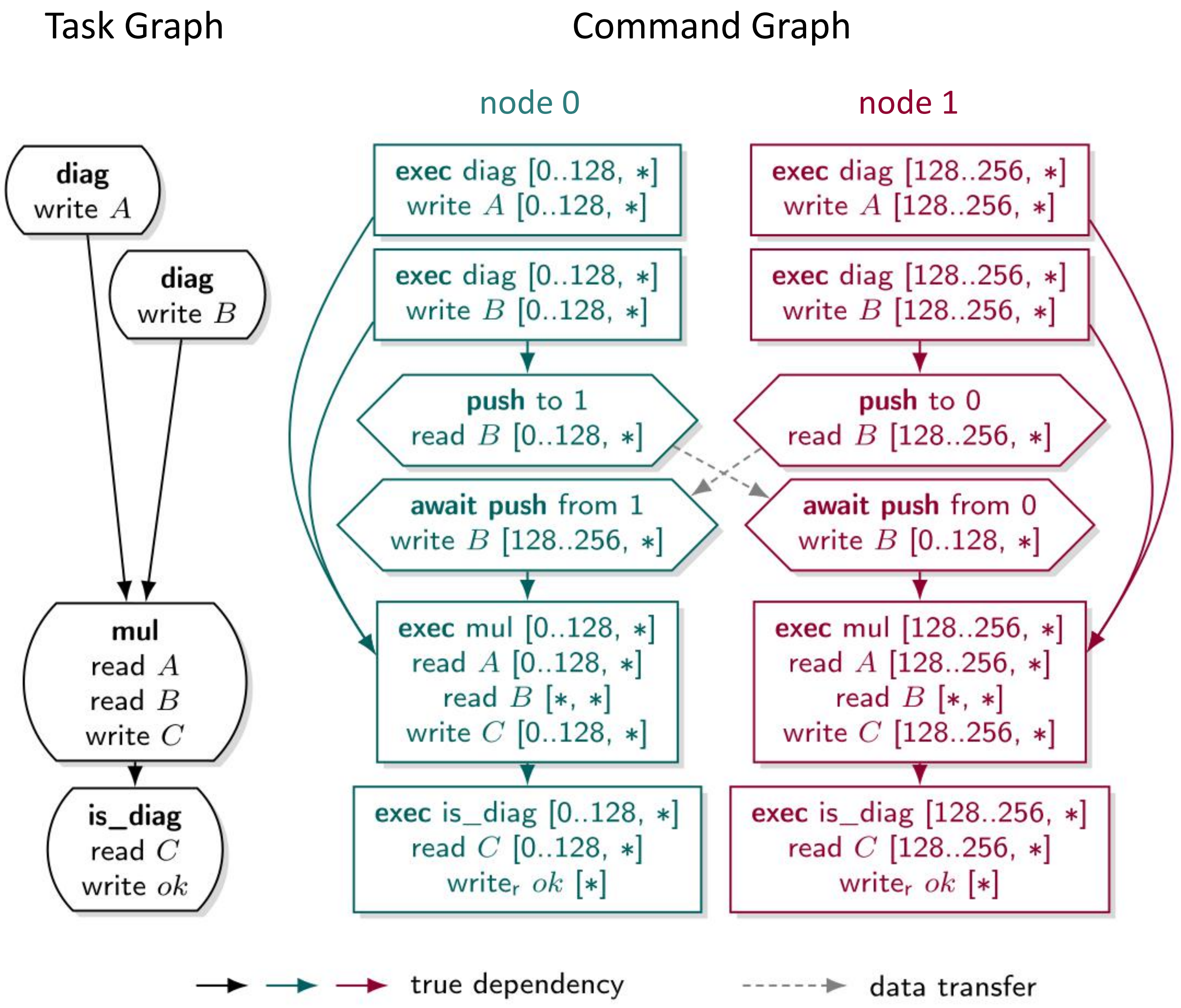
Graph-Based Execution Model

Example with 4 kernels:

```
distr_queue q;
buffer<float, 2> A, B, C;
buffer<bool> ok;

q.submit([=](handler &cgh) {
    diag(cgh, A, 2); });
q.submit([=](handler &cgh) {
    diag(cgh, B, 3); });
q.submit([=](handler &cgh) {
    mul(cgh, A, B, C); });
q.submit([=](handler &cgh) {
    is_diag(cgh, C, 6, ok); });

return /* ok[0] is true */;
```



Task Graph

diag write A

diag write B

mul read A read B write C

is_diag read C write ok

Command Graph

node 0

exec diag [0..128, ∗] write A [0..128, ∗]

exec diag [0..128, ∗] write B [0..128, ∗]

push to 1 read B [0..128, ∗]

await push from 1 write B [128..256, ∗]

exec mul [0..128, ∗] read A [0..128, ∗] read B [∗, ∗] write C [0..128, ∗]

exec is_diag [0..128, ∗] read C [0..128, ∗] write_r ok [∗]

node 1

exec diag [128..256, ∗] write A [128..256, ∗]

exec diag [128..256, ∗] write B [128..256, ∗]

push to 0 read B [128..256, ∗]

await push from 0 write B [0..128, ∗]

exec mul [128..256, ∗] read A [128..256, ∗] read B [∗, ∗] write C [128..256, ∗]

exec is_diag [128..256, ∗] read C [128..256, ∗] write_r ok [∗]

true dependency

data transfer

Difficult: Accessing Buffers Data from outside the Runtime

Runtime-managed data lives in an **asynchronous execution** context, requires manual synchronization with the main thread

① compute value in kernel

② copy value to outer scope
(bug-prone reference capture)

③ implicitly synchronize
on queue shutdown

④ use value in
main thread

```
int main() {
    bool host_ok;
    {
        distr_queue q;
        // ...
        buffer<bool> ok{1};
        q.submit([=](handler& cgh) { is_diag(cgh, C, ok); });

        q.submit(allow_by_ref, [=, &host_ok](handler& cgh) {
            accessor passed_acc{ok, cgh, access::all{},
                     read_only_host_task};
            cgh.host_task(on_master_node, [=, &host_ok] {
                host_ok = passed_acc[0];
            });
        });
    }
    return host_ok ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

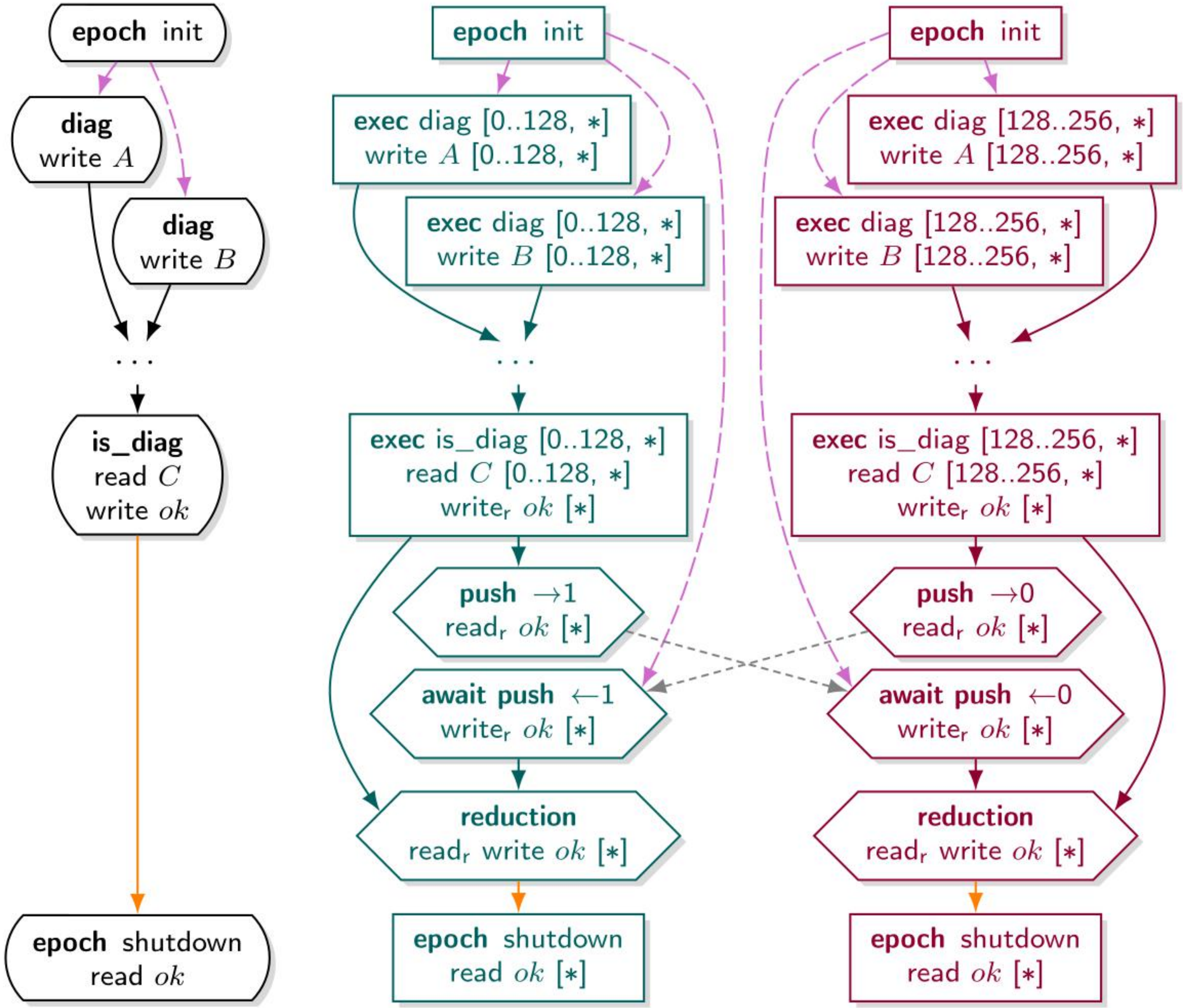
New: Epoch Nodes

Epochs **serialize execution** and carry data requirements like any other graph node:

```
distr_queue q;
buffer<float, 2> A, B, C;
buffer<bool> ok;

q.submit([=](handler &cgh) {
    diag(cgh, A, 2); });
q.submit([=](handler &cgh) {
    diag(cgh, B, 3); });
q.submit([=](handler &cgh) {
    mul(cgh, A, B, C); });
q.submit([=](handler &cgh) {
    is_diag(cgh, C, 6, ok); });

return q.drain(capture{ok})[0];
```



New in the Frontend: Buffer Captures API

Free-standing buffer data representation

```
template <typename T, int Dims>
class buffer_data {
    decltype(auto) operator[](size_t idx);
};
```

Descriptor for capture of a single buffer subrange

```
template <typename T, int Dims>
class capture<buffer<T, Dims>> {
    using value_type = buffer_data<T, Dims>;
    explicit capture(buffer<T, Dims> buf);
};
```

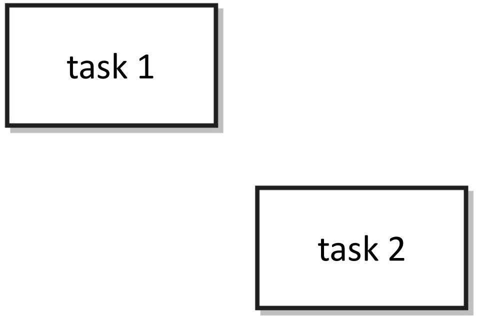
Synchronization function extracting data for one or more captures

```
class distr_queue {
    template <typename T> typename capture<T>::value_type
        drain(const capture<T>& cap);
    template <typename... Ts> std::tuple<capture<Ts>::value_type...>
        drain(const std::tuple<capture<Ts>...>& caps);
};
```

Node-Local Side Effects

Problem: Effects on global resources or reference-captured objects do not generate dependencies

```
int main() {
    distr_queue q;
    std::ofstream ofs("file.txt");
    q.submit(allow_by_ref, [&](handler& cgh) {
        cgh.host_task(on_master_node, [&] { ofs << "Hello "; });
    });
    q.submit(allow_by_ref, [&](handler& cgh) {
        cgh.host_task(on_master_node, [&] { ofs << "world!"; });
    });
}
```

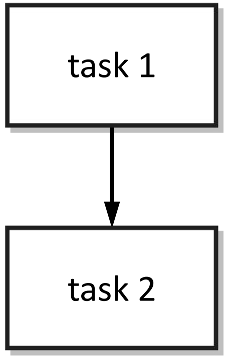


Celerity will regard Task 1 and Task 2 are concurrent, yet there exists a dataflow dependency

⟹ **Data Race / Undefined Behavior!**

New: Modelling Local Dependencies through Side Effects

```
int main() {
    distr_queue q;
    host_object<std::ofstream> ofs("file.txt");
    q.submit([=](handler &cgh) {
        side_effect e{ofs, cgh};
        cgh.host_task(on_master_node, [=] { *e << "Hello "; });
    });
    q.submit([=](handler &cgh) {
        side_effect e{ofs, cgh, sequential_order};
        cgh.host_task(on_master_node, [=] { *e << "world!"; });
    });
}
```



① Wrap shared resource in a **host object**, transferring ownership to the runtime

② Capture host-objects in host tasks **by value**

③ Obtain access inside a host task through a **side effect**, specifying side-effect order

New: Host Objects and Side Effect API

Buffer-like container
for a user-defined type

```
template <typename T>
class host_object {
    host_object(T&& obj);
};
```

Concurrency constraint

```
enum class side_effect_order { relaxed, exclusive, sequential };
```

Accessor-like reference
to a host object

```
template <typename T, side_effect_order Order = sequential>
class side_effect {
    side_effect(const host_object<T>& object, handler& cgh,
            order_tag<Order> = {} /* for deduction only */);
    T& operator*() const;
    T* operator->() const;
};
```

CTAD deduction tags

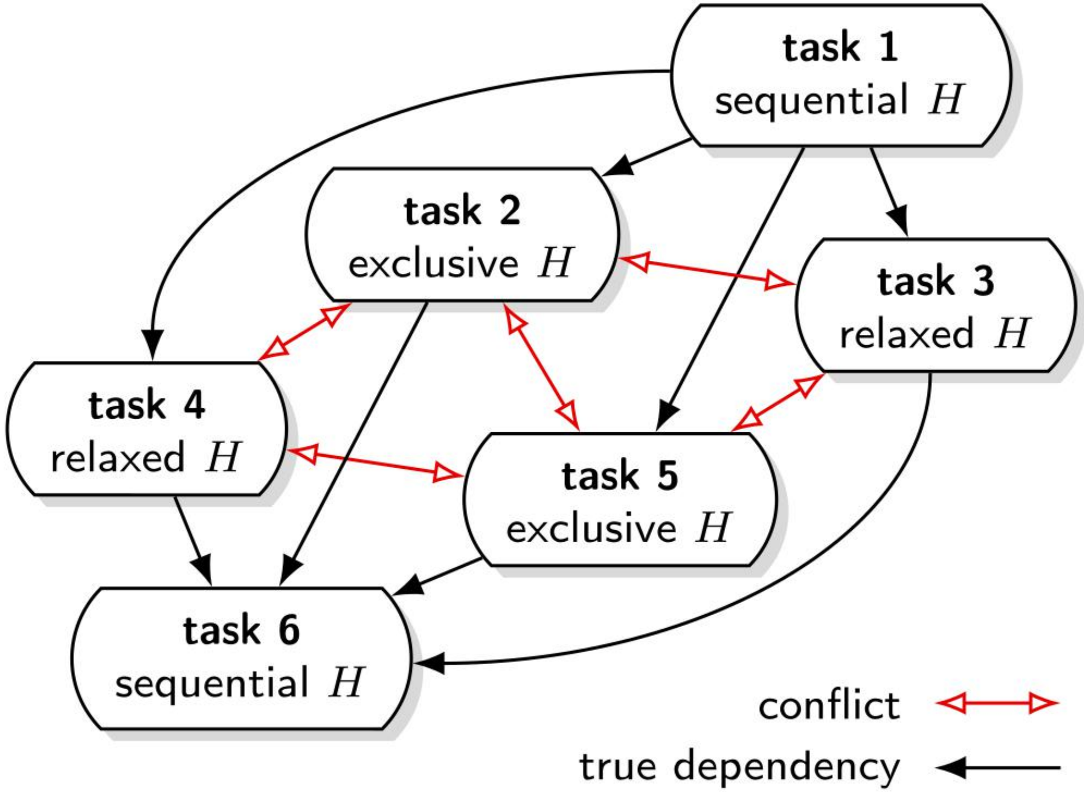
```
constexpr order_tag<side_effect_order::relaxed> relaxed_order;
constexpr order_tag<side_effect_order::relaxed> exclusive_order;
constexpr order_tag<side_effect_order::relaxed> sequential_order;
```

Side Effect Orders

Similar to read-write access modes on buffers, we can **increase potential concurrency** by specifying synchronization requirements on host objects:

side effect order	concurrency	reordering
sequential	✗	✗
exclusive	✗	✓
relaxed	✓	✓

Concurrency restrictions add **undirected edges** to task and command graphs, which become mixed **conflict graphs.**



Conflict Graph for one host object *H*

Opportunistic Scheduling of Conflict Graphs on Workers

Commands are streamed to worker nodes, no future commands are known at scheduling time.
A command is *eligible* if all its dependencies are met.

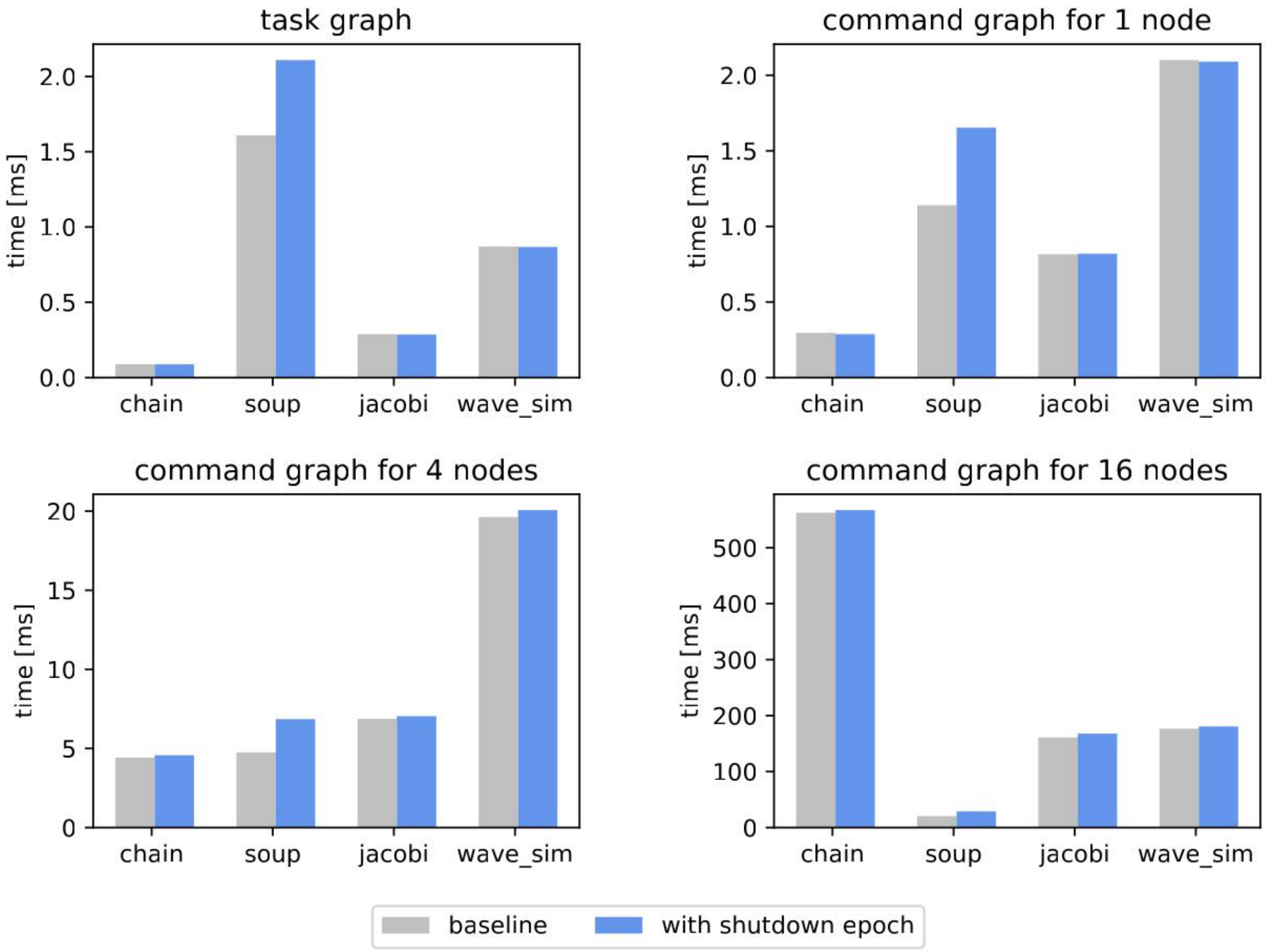
In **pure Directed Acyclic Graphs** (only dependency edges):

- Greedily start executing all eligible commands as soon as possible.

In mixed **Conflict Graphs** (directed dependency edges + undirected conflict edges):

- find the largest subset where no command has a conflict with
    a) any other command in the same subset
    b) any currently executing command
- Solve for the **Maximum Independent Set** with regard to conflict egdes.
  NP-complete $\Longrightarrow$ approximate by **backtracking** with a limited the number of steps.

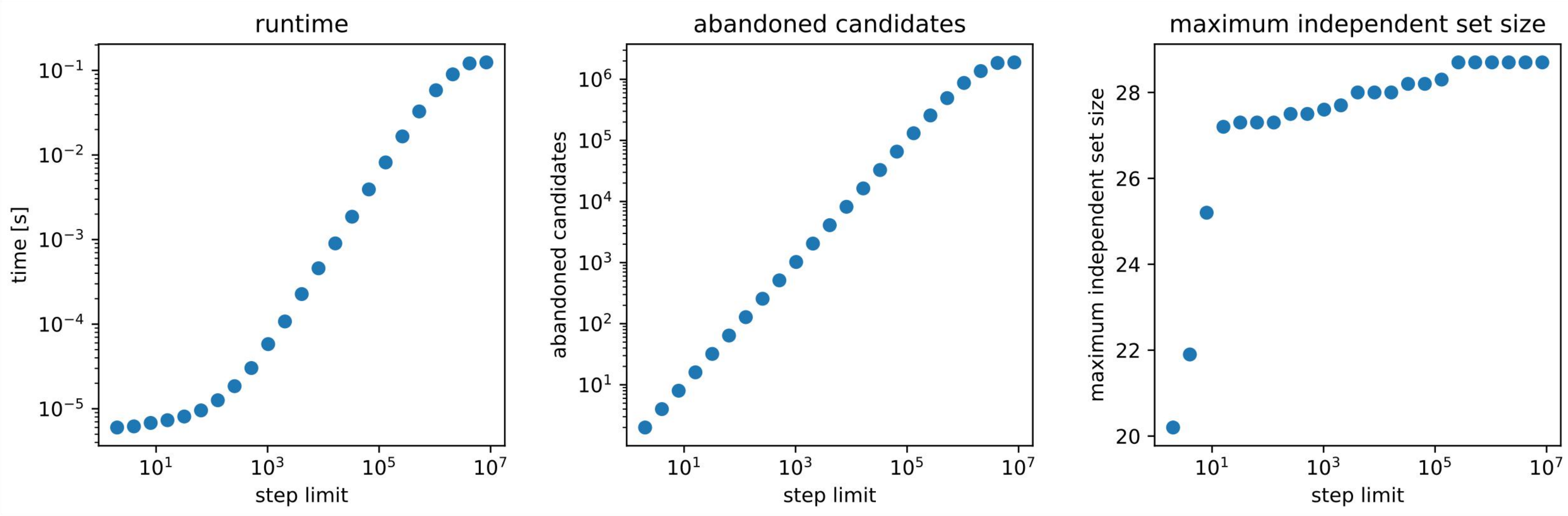
Benchmark: DAG Generation Overhead for Inserting a Shutdown Epoch



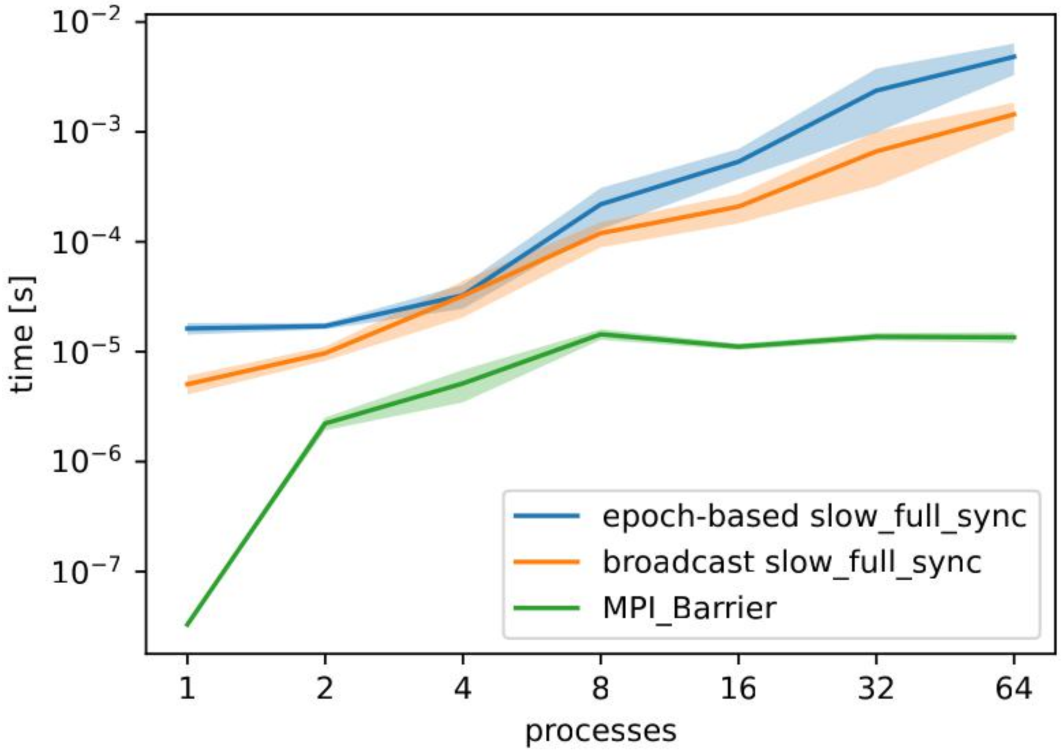
For extremely broad graphs (*soup*), dependency generation for an epoch has measurable additional cost. For long dependency chains (*chain* and *wave_sim)*, overhead is much less pronounced.

Benchmark: Backtracking approximation for Conflict Graph Scheduling

The number of backtracking steps must be limited to mitigate the exponential runtime behavior.
With 40 eligible commands and 20 conflicts:



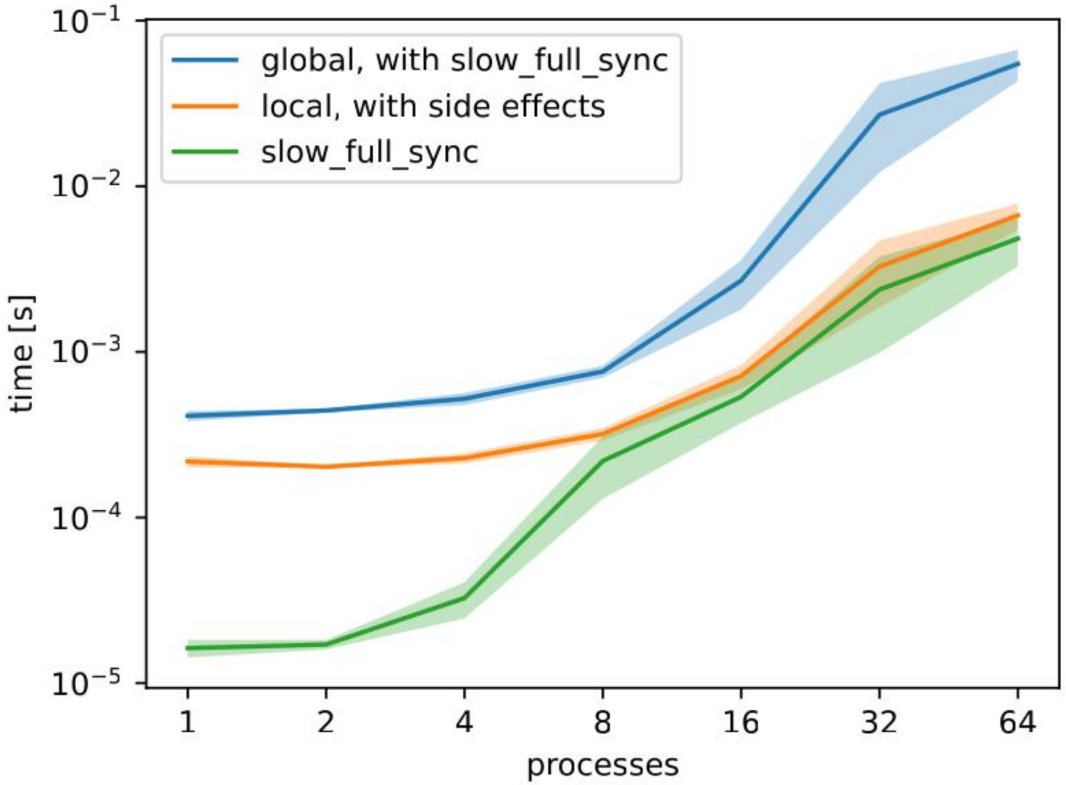
Benchmark: Epoch Based vs Ad-Hoc Synchronization



Epoch-based synchronization has measurable overhead, but is intended to be used scarcely.

In this chart, `slow_full_sync()` is a queue barrier operation similar to `drain()`, but allows the program to resume afterwards.

Benchmark: Side Effects vs. Barrier Synchronization Between Dependent Kernels



Barriers (blue) are too coarse to achieve good performance for fine-grained synchronization. Side effects on the other hand have negligible overhead.

Here, `slow_full_sync()` is the baseline used internally for timing both implementation candidates.

Thank you!

Check out Celerity: https://celerity.github.io

Contact me: fabian.knorr@uibk.ac.at

DPS
Distributed and Parallel Systems

universität innsbruck

This project has received funding from the
European High Performance Computing Joint Undertaking (JU)
under grant agreement **No 956137.**