International Journal of Parallel Programming (2023) 51:150–171
https://doi.org/10.1007/s10766-022-00743-4

Declarative Data Flow in a Graph-Based Distributed Memory Runtime System

Fabian Knorr[1] · Peter Thoman[1] · Thomas Fahringer[1]

Received: 8 September 2022 / Accepted: 21 November 2022 / Published online: 26 December 2022

Abstract

Runtime systems can significantly reduce the cognitive complexity of scientific applications, narrowing the gap between systems engineering and domain science in HPC. One of the most important angles in this is automating data migration in a cluster. Traditional approaches require the application developer to model communication explicitly, for example through MPI primitives. Celerity, a runtime system for accelerator clusters heavily inspired by the SYCL programming model, instead provides a purely declarative approach focused around access patterns. In addition to eliminating the need for explicit data transfer operations, it provides a basis for efficient and dynamic scheduling at runtime. However, it is currently only suitable for accessing array-like data from runtime-controlled tasks, while real programs often need to interact with opaque data local to each host, such as handles or database connections, and also need a defined way of transporting data into and out of the virtualised buffers of the runtime. In this paper, we introduce a graph-based approach and declarative API for expressing side-effect dependencies between tasks and moving data from the runtime context to the application space.

Keywords Runtime system · DAG · Accelerator · Data flow · API

1 Introduction

Modern scientific and High Performance Computing (HPC) is a challenging environment for software engineering. In order to increase compute throughput despite the ever tighter constraints on power efficiency, modern supercomputer

✉ Fabian Knorr
fabian@dps.uibk.ac.at

Peter Thoman
petert@dps.uibk.ac.at

Thomas Fahringer
tf@dps.uibk.ac.at

[1] University of Innsbruck, Innsbruck, Austria

hardware embraces heterogeneous processor architectures, deep memory hierarchies with non-uniform access characteristics and specialized network topologies. Most of the increasing complexity is directly passed onto the application developer in the form of intricate APIs and in some cases entirely disjoint programming models allowing optimal utilization of the available technologies in every use case. While the resulting increase in up-front development cost can be acceptable for large-scale applications such as general-purpose simulation toolkits, specialized single-use codes for novel discovery will not have the development budget required to test a research hypothesis that might turn out to be a dead-end.

Distributed Memory Runtime Systems are an established concept for easing select aspects of the complexity in these heterogeneous systems, such as performance portability, optimizing execution schedules with unbalanced loads or automatic data migration between computation steps. They typically incur a trade-off between expressiveness, correctness guarantees, and the level of permitted user control.

The mission statement of Celerity [14], a task-based distributed memory runtime system for accelerator clusters, is to make programming heterogeneous HPC systems more accessible and time-efficient by facilitating low-effort porting of single-node SYCL [11] accelerator programs. The Celerity model decomposes a problem into compute tasks and their data dependencies, using subdivision of the computational index spaces to transparently distribute work onto a cluster. Celerity exposes a declarative, data-flow-based API operating on virtualized buffers, inferring dependencies and necessary data transfers in the distributed program and relieving the programmer of manual scheduling decisions and data migration.

Celerity's APIs allow it to statically guard against unmanaged buffer accesses and race conditions between tasks, greatly reducing the potential for programming errors. The runtime implementation benefits from an information-dense API that supports the generation of efficient execution schedules, while the user is assured of their code's correctness by an expressive programming paradigm, allowing them to focus on core algorithm development instead.

A notable use of Celerity is the Cluster-accelerated magnetohydrodynamics simulation CRONOS [9], which demonstrates the viability of the Celerity model for real-world applications. It is also sufficiently generic to serve as the basis for further abstractions like the Celerity High-level API [15], a programming model exposing data transformations using composable functional operator pipelines similar to the C++20 *ranges* library.

While domain-specific problems can be fully described by compute tasks and data dependencies between them, real codes need additional features to perform I/O operations with side effects. Incremental porting from single-node SYCL applications, an important development goal of Celerity, further requires data movement between the legacy host application and runtime-controlled virtual buffers.

In this paper, we present an approach to augmenting the Celerity execution model with declarative mechanisms for tracking I/O side effects and safely moving data out of the managed context on pre-existing synchronization points.

2 Related Work

We compare our novel developments in Celerity to state-of-the-art runtime systems based on their coherence model and synchronization behavior.

SYCL [11] is an industry-standard, single-source programming model for parallel software targeting hardware accelerators. A multitude of implementations exist, with backends for GPUs [1], multi-core CPUs, and application-specific FPGAs [10]. Its execution model is fundamentally asynchronous, and scheduling is constrained by implicit and explicit data dependencies on buffers. SYCL is the primary influence on the API of Celerity, which aims to ease porting from single-node SYCL programs to distributed-memory applications.

Legion [3] is a runtime system for distributed heterogeneous architectures including GPU clusters. It models task parallelism through manual subdivision of programs into hierarchical tasks in accordance with user-controlled data partitioning. Legion tasks are spawned and awaited asynchronously based on *futures*, giving the runtime's out-of-order scheduler the freedom to migrate tasks between nodes. Unlike other systems, there is no notion of a "main thread" driving the execution flow, instead, any task (starting with a single *top-level task*) has the freedom to issue more parallel work as it executes.

SkePU [6, 7] is a skeleton programming system targeting single-node execution on CPUs or GPUs or distributed execution on an MPI-based backend. Skeletons are higher-order constructs such as *Map*, *Reduce* or *Scan* that can be efficiently implemented on all target backends. SkePU follows a synchronous model where skeleton computations are performed in lock-step with the main program flow. Memory coherence between host and device containers (and in a distributed setting, within a container partitioned between MPI ranks) must be established manually using *flush* commands.

Kokkos [5, 16] is a single-source programming model targeting various high-performance computing architectures. It optimizes performance portability by building abstractions on both the compute and memory hierarchy of modern hardware. Kokkos has both synchronous and asynchronous APIs for dispatching work, depending on how output data is passed back to the caller. The user explicitly controls in which memory space data resides in and for which access pattern the data layout is optimized, e.g. with row-major or column-major matrix layouts.

3 The Celerity Runtime System

Celerity is a high-level C++ API and runtime system bringing the SYCL [11] accelerator programming model to distributed-memory clusters. Using an enhanced declarative description of data requirements, it transparently distributes compute kernels onto the nodes of a cluster while maintaining an API very close to its single-node ancestor. Celerity has evolved significantly beyond what has previously been published [13, 14], so we give a broad overview of the interface and execution model.

```
using mat = buffer<float, 2>;
const range<2> size{256, 256};

void diag(handler& cgh, mat& M, float d) {
    accessor m{M, cgh, access::one_to_one{}, write_only, no_init};
    cgh.parallel_for(size, [=](item<2> i) {
        m[i] = i[0] == i[1] ? d : 0;
    });
}

void mul(handler& cgh, mat& A, mat& B, mat& C) {
    accessor a{A, cgh, access::slice<2>{1}, read_only};
    accessor b{B, cgh, access::slice<2>{0}, read_only};
    accessor c{C, cgh, access::one_to_one{}, write_only, no_init};
    cgh.parallel_for(size, [=](item<2> i) {
        c[i] = 0;
        for(size_t k = 0; k < i.get_range(0); ++k) {
            c[i] += a[i[0]][k] * b[k][i[1]];
        }
    });
}

void is_diag(handler& cgh, mat& C, float d, buffer<bool>& ok_buf) {
    accessor c{C, cgh, access::one_to_one{}, read_only};
    auto ok_r = reduction(ok_buf, cgh, sycl::logical_and<bool>{},
            property::reduction::initialize_to_identity{});
    cgh.parallel_for(size, ok_r, [=](item<2> i, auto &ok) {
        ok.combine(c[i] == (i[0] == i[1] ? d : 0));
    });
}

int main() {
    distr_queue q;
    mat A{size}, B{size}, C{size};
    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); });
    buffer<bool> ok{1};
    q.submit([=](handler& cgh) { is_diag(cgh, C, 6, ok); });
    return /* ok[0] is true */ ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing 1: Simple Celerity program computing the product of two diagonal matrices.

Listing 1 exemplifies the source code of a typical Celerity application. The `main` function allocates three two-dimensional buffers for square matrices and instantiates a *distributed queue*. It then launches a sequence of kernels that initialize A and B as diagonal matrices (`diag` function) and compute the naïve matrix product $C := A \cdot B$ (`mul` function). Finally, the result is verified by launching a fourth kernel that computes the expected value of each c_{ij} and combines the results using a distributed reduction over the `&&` operator.

Work is submitted to the asynchronous distributed queue in the form of *command group functions*, which are implemented as lambdas receiving a `command group` handler called cgh in the example. A command group declares a set of buffer requirements and specifies the work to be executed.

Buffer access is guarded by *accessors*, which bind buffers to the command group handler and inform Celerity of the mode of access and the access ranges through *range mappers* (here one_to_one and slice). Captured inside the kernel function that is passed on to parallel_for, these accessors facilitate reading and writing of the actual buffer contents.

All submissions to the distributed queue happen asynchronously and instruct Celerity to build an internal representation of data requirements and execution ranges. The actual scheduling, distribution and execution of the submitted kernels within the cluster is transparently managed by the runtime. The completion of all submitted command groups is finally awaited implicitly by the ~distr_queue() destructor.

As indicated by the comment in the last line of main, Celerity does not have a designated mechanism for transporting data managed by the runtime back to the host application. Closing this gap is non-trivial and a core contribution of this work, for which workarounds need to be inserted currently.

3.1 Celerity's Graph-Based Execution Model

Execution of a Celerity program is distributed unto *nodes*, where a designated *master node* creates the execution schedule for the entire cluster and determines how data and computational load is distributed. This centralized approach has the potential to incorporate dynamic scheduling decisions such as load balancing at runtime without requiring costly synchronization between equal nodes in a distributed scheduling setting. By relying on fully asynchronous work assignment, Celerity is able to avoid the scalability problems that a more traditional lock-step implementation of centralized scheduling would be certain to encounter.

As command groups are submitted from the application thread of a Celerity program, a coarse-grained, directed acyclic graph (DAG) called the *task graph* is constructed. Each command group creates a corresponding task node, and data dependencies between command groups manifest as true- or anti-dependencies as if the entire program was executed on a single node.

On the master node, the *scheduler* then constructs a fine-grained *command graph* that models the distributed executions and all data transfers that arise with it. Commands are always bound to a particular node, but the precise projection of tasks onto commands varies with the task type. For example, device execution tasks, which are generated from command groups invoking handler::parallel_for(), may be split such that each worker node receives one part of the total execution range.

Figure 1 shows possible task and command graphs for the program in Listing 1. While the task graph reflects the high-level dependency structure visible in the source code, the command graph contains only dependencies induced by the subranges executed on each node.

Within task and command graph, dependencies are assigned based on the access modes of buffer accesses and the submission order. For example, a command group

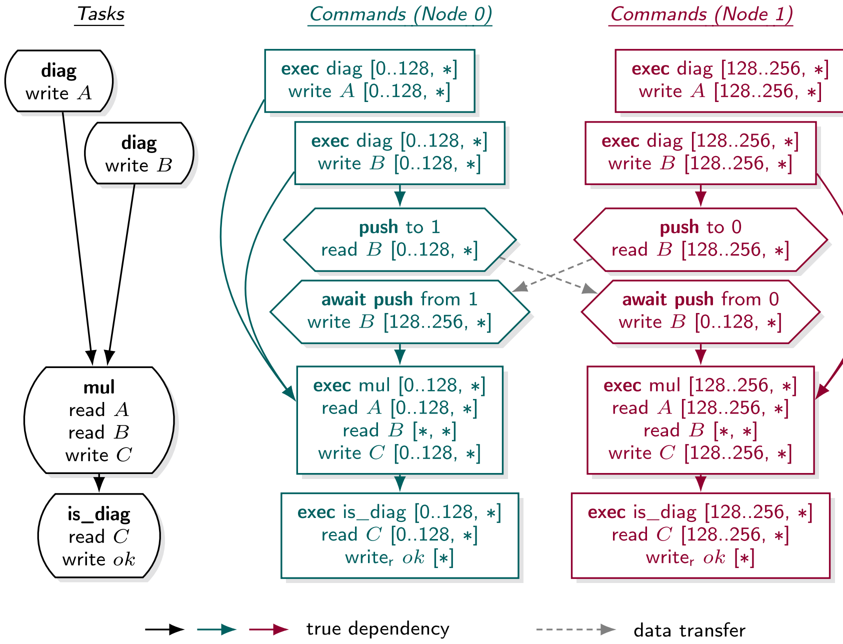
Tasks
Commands (Node 0)
Commands (Node 1)
diag write A
diag write B
mul read A read B write C
is_diag read C write ok
exec diag [0..128, ∗] write A [0..128, ∗]
exec diag [0..128, ∗] write B [0..128, ∗]
push to 1 read B [0..128, ∗]
await push from 1 write B [128..256, ∗]
exec mul [0..128, ∗] read A [0..128, ∗] read B [∗, ∗] write C [0..128, ∗]
exec is_diag [0..128, ∗] read C [0..128, ∗] write_r ok [∗]
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

Fig. 1 Task graph (left) and command graph (right) arising from Listing 1 for two nodes in stable Celerity. Kernel execution commands show the 2-dimensional iteration sub-range and the resulting data requirements as assigned by the scheduler. In each dimension, the interval a..b includes a but excludes b, and $*$ denotes the entire range. The necessary inter-node data exchange generates auxiliary *push* / *await push* command pairs

with write access followed by a command group with read access to the same buffer region will generate a true dependency, while the inverse order will generate an anti-dependency.

A unique concept in Celerity, and one of the fundamental points where its API differs from SYCL, are *range mappers*. These projections, required on each accessor, inform the runtime which portions of each buffer an arbitrary subdivision of the execution space will access.

The stream of serialized commands is forwarded to the respective worker nodes, which place them into their *executor queue*. The executor of each worker node will then make its own local scheduling decisions to best allocate its resources to the pending commands. While all nodes construct identical task graphs in parallel, the command graph structure only exists on the master node in its full form. Pure worker nodes only reconstruct the relevant dependency graph locally from the serialized commands.

4 Modeling Node-Local Side Effects

SYCL and Celerity share the concept of *host tasks* that asynchronously schedule the execution of arbitrary code on the host, avoiding host-device synchronization and scheduler stalls. Similar to device tasks, host tasks can read and write buffers through the accessor mechanism. Additionally, they are able interact with operating system APIs such as file I/O and reference objects allocated in the main thread, since they operate in the same address space. As soon as multiple host tasks references a single resource, the resulting synchronization or ordering constraints need to be enforced during execution.

The only synchronization primitive offered by Celerity are cluster-wide barriers that can be inserted between command groups through the aptly-named `distr_queue::slow_full_sync()` API. These barriers additionally serialize the execution on each node and synchronize between the main and executor threads of the runtime.

In order to avoid race conditions around node-local state, the application developer must currently insert such a barrier in any place where an invisible node-local dependency exists between two tasks. This “sledgehammer synchronization” is not only error-prone, but also detrimental to application performance due to the subsequent stalling of work submission.

In the following, we want to explore how to establish ordering on node-local state while conserving as much scheduling freedom as possible through an in-graph mechanism.

4.1 Node-Local Side Effects and Dependencies in Related Work

SYCL offers host tasks for asynchronously executing arbitrary C++ code. In addition to implicit data dependencies arising from buffer accesses, a user is free to add control-flow dependency edges using the `handler::depends_on()` API. These dependencies ensure correct ordering around side effects.

Legion forbids side-effects inside task code since its scheduler will dynamically migrate tasks between nodes. To perform I/O work, Legion offers specialized *Launchers* that permit attaching global resources to a task.

SkePU forbids side effects inside skeleton user functions to ensure protability between CPU and accelerator backends. Since it uses lock-step execution, code containing side effects can be freely interspersed with skeleton calls as long as the necessary memory coherence is established using *flush* commands. In the distributed setting, SkePU offers the `external` facility for constraining code with cluster-global side effects to a single MPI rank.

Kokkos has support for light-weight task parallelism using the `host_spawn` facility. Spawing a task will yield a *future* which can be named as a prerequisite to a successor task, introducing a scheduling dependency. Aside from the naming, this approach is identical to SYCL.

4.2 Dataflow-Centric: Host Objects and Declarative Side Effects

Even though the closely-related SYCL sets a precedent for explicit control-flow dependencies, the `depends_on` API is primarily intended for the alternative, explicit memory management added in SYCL 2020 a feature that is fundamentally at odds with the transparent coherence model of Celerity.

To the contrary, adopting this approach would introduce room for user error that does not exist for buffer data dependencies since no connection could be made between a dependency declaration and the actual side effect.

As a novel data-flow centric API, we introduce the concept of *host objects* and *side effects* as shown in Listing 2. Similar to how buffers and accessors manage distributed data, they provide an expressive and safe interface for creating data-flow dependencies between command groups.

A **host object** is a wrapper to a reference or value type with semantics that are entirely user-defined, but for which access is guarded by the runtime. Any host object is guaranteed to outlive its last observing host task, so no dangling reference problems arise from deferred kernel execution.

A **side effect**, when defined in a command group, grants the host task access to a host object and communicates the resulting local ordering constraints to the runtime. The host object–side effect duality is deliberately similar to the one between buffers and accessors, both in SYCL and Celerity.

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

template <typename T>
class host_object<T&> {
    host_object(std::reference_wrapper<T> obj);
};

template <>
class host_object<void> {
    host_object();
};

enum class side_effect_order { relaxed, exclusive, sequential };
template<side_effect_order> struct /* exposition only */ order_tag {};
inline constexpr order_tag<side_effect_order::relaxed> relaxed_order;
inline constexpr order_tag<side_effect_order::exclusive> exclusive_order;
inline constexpr order_tag<side_effect_order::sequential> sequential_order;

template <typename T, side_effect_order Order = sequential>
class side_effect {
    side_effect(const host_object<T>& object, handler& cgh,
        order_tag<Order> = {} /* for class template argument deduction */);
    /* reference-type */ operator*() const; // when T is not void
    /* pointer-type */ operator->() const; // when T is not void
};
```

Listing 2: Host Object and Side Effect API

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

Listing 3: Using side effects to serialize writes to a shared file handle

The example in Listing 3 shows how a file handle is wrapped in a host object to capture it in a host task. Thereafter, accessing the handle itself is only possible by constructing a side effect. This statically guarantees that the object state can only be observed inside host tasks and resulting ordering constraints are always known to the runtime.

To guard against the accidental observation of non-managed state, we assert at compile time that a command group function does not capture by reference[1] unless it is passed with the `allow_by_ref` tag. Since buffers and host objects have shared-pointer semantics internally, by-value captures are always sufficient in kernels interacting with them.

4.3 Accurate Scheduling Constraints through Side Effect Orders

By default, side effects as proposed above will always serialize execution between host tasks observing the same object. Since host objects are opaque and the precise semantics of interactions within the host task cannot be further inspected by the runtime, this can be overly restrictive. For example, incrementing an atomic counter from multiple host tasks does not need to introduce any scheduling or synchronization constraints, but the user should still be able to rely on the runtime for the liveness guarantees on the host object.

Choosing between different scheduling guarantees for side effects is reminiscent of access modes on buffer access. However, the read–write dichotomy itself is not a good fit for this new use case: First of all, whether two “writing” side effects can be scheduled concurrently or not depends on the level of synchronization employed by the object itself, which is outside of Celerity’s control. Also, for

[1] In C++, references and types transitively containing references are not considered *standard layout types*, so this property can be conservatively verified using `std::is_standard_layout_v<>`.

buffers, the access modes are instructive of implicit data movement by the runtime, which does not apply to host objects either.

We therefore propose three distinct *side effect orders* that can optionally be specified when a side effect is declared:

- *sequential order*: The task cannot be re-ordered against or executed concurrently with any other task affecting the same host object.
- *exclusive order*: The task may be re-ordered, but not executed concurrently with any other task affecting the same host object.
- *relaxed order*: The task may be executed concurrently with and freely re-ordered against other tasks affecting the same host object.

Relaxed-order side effects are sufficient if the contained object provides synchronization internally, or if the task only performs inherently thread-safe non-mutating accesses while any mutating operations in other tasks occur in the context of a sequential-order side effect.

```
procedure AddSideEffect($t, h$)
    if $s_h$ exists $\wedge$ ($r(t,h) \neq$ sequential $\vee$ $A_h = \emptyset$) then
        $D \leftarrow D \cup \{(t \rightarrow s_h)\}$
    end if
    if $r(t,h) =$ sequential then
        $D \leftarrow D \cup \{(t \rightarrow t') \mid t' \in A_h\}$
        $A_h \leftarrow \emptyset$
        $s_h \leftarrow t$
    else
        $C \leftarrow C \cup \{(t \leftrightarrow t') \mid t' \in A_h$
                $: r(t,h) =$ exclusive $\vee$ $r(t',h) =$ exclusive$\}$
        $A_h \leftarrow A_h \cup \{t\}$
    end if
end procedure
```

Legend

t	task
h	host object
s_h	last task with sequential side effect on h
$r(t,h)$	side effect order of task t on host object h
A_h	active conflict set of tasks on h
D	set of dependenciesd (directed edges)
C	set of conflicts (undirected edges)

Algorithm 1: Generating dependency and conflict edges for side effects on the task graph. This algorithm also applies to the command graph, where states (D, C, A, s) are tracked separately per worker instead.

An exclusive-order side effect is indicated when execution order is irrelevant, but concurrent accesses would violate synchronization requirements. This is superior to a relaxed-order side effect combined with manual locking if the lock would have to be held for any significant amount of time. Instead of stalling executor threads, each worker node is able to generate efficient local schedules around the resulting constraints ahead of time.

A sequential-order side effect must be used when re-ordering would change the semantics of the node-local state in a way that invalidates results, or concurrency on execution would violate synchronization requirements. This is the strongest guarantee and also the default behavior.

Note that between a pair of tasks affecting the same host object, the more restrictive side effect order decides the level of freedom with respect to re-ordering and concurrency. As a consequence, relaxed side effects give a stronger

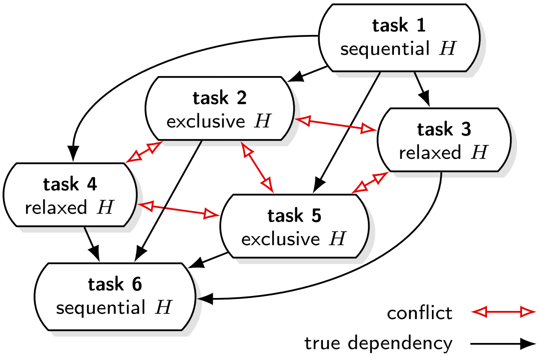

Fig. 2 Mixed task graph originating from side effects with different orders on a single host object H. Sequential-order side effects serialize against other tasks using temporal dependencies, whereas exclusive-order side effects introduce conflict edges to otherwise concurrent tasks. No edge arises between the two relaxed tasks 3 and 4, so this pair remains concurrent. The associated command graph (not shown here) will have an equivalent structure

guarantee than an unmanaged reference-capture of the raw object would, since they are guaranteed to not be re-ordered against sequential effects.

To implement re-ordering constraints, we augment the task and command graph structures to track undirected conflict edges between tasks in addition to the existing directed dependency edges. Conflict edges indicate mutual exclusion between tasks, a strictly weaker requirement than the serializing dependencies impose. Task and command graphs thus become mixed graphs as seen in Fig. 2 Algorithm 1 shows how dependencies and conflicts are derived from side effects.

As evaluating the necessary concurrency constraints of arbitrary operations on a host object requires intricate knowledge of its API guarantees, we consider the explicit specification of side effect orders an advanced feature. The sequential default guarantees scheduling correctness until an exact set of constraints proves beneficial for a specific problem.

4.4 Opportunistic Scheduling of Mixed Command Graphs

The output of the existing Celerity scheduler is a stream of commands per node consisting of kernel execution ranges, metadata, and an list of prior command identifiers that it depends on. These commands are serialized to worker nodes in a topological order of the directed dependency graph. Executors do not need to reconstruct the command graph from this stream, but can instead maintain a set of *eligible commands* which contains all those that have no remaining unmet dependencies. The executor can then perform local scheduling on the eligible set to dynamically optimize resource utilization.

With the addition of conflict edges to the command graph, we extend the local scheduler to handle mutual exclusions between commands. The theory behind efficient scheduling around conflict graphs has been studied in the context of scheduling tasks with known completion times on a fixed number of general-purpose processors [2]. For certain classes of graphs, optimal solutions can be found efficiently [4].

Because Celerity has no *a priori* knowledge of kernel execution times and aims to minimize latencies by intentionally leaving low-level allocation of resources like GPU cycles to the operating system scheduler, the scheduling target is to maximize the number of active concurrent tasks.

A correct but sub-optimal implementation could execute all eligible conflicting commands sequentially in receiving order. This however misses potential concurrency between tasks, and to properly harness the increased scheduler freedom, we instead find the largest conflict-free set of eligible commands.

As a classic NP-hard graph theory problem, the Maximum Independent Set can be found in exponential time through backtracking [8], although other, more efficient algorithms exist [12, 17]. Since we expect the eligible set to be rather small most of the time, we implement a simple backtracking solution that will yield sufficient performance in the common case. Independent of the algorithm, the exponential growth of run time can thwart potential efficiency gains of the scheduler, so we stop backtracking early after rejecting 100 candidate solutions to limit evaluation time to a constant on degenerate graphs.

This method is opportunistic as the full set of eligible commands may not be known at the time a scheduling decision is made. Commands should begin execution as soon as they arrive to minimize latency, so waiting for a certain filling degree is infeasible. However, since we expect most commands to have an execution time that greatly exceeds that of command generation, executors will have a well-filled command queue and thus the full set of eligible commands for one earlier time step most of the time.

5 Data Extraction from Runtime-Managed Structures

Although the Celerity runtime mostly concerns itself with distributing work while keeping actively managed buffer data coherent between nodes, real-world applications must be able to convert existing in-memory data into Celerity data structures on startup and extract buffer contents and host object state once execution has completed.

The former is already available in Celerity today: like in SYCL, buffers can be initialized from a pointer to host memory on construction, assuming that all nodes pass identical initialization data. In the same fashion, host objects can be constructed from arbitrary values.

There is however no native way for the application to observe buffer data or host object state in the main thread after their construction. Instead, host tasks must be used to export data through the file system or copy them to a user-controlled data structure that can be accessed once the asynchronous task has finished executing.

Stalling the main thread for synchronization with such a host task interrupts the asynchronous submission of more work, negatively impacting performance by starving workers until the barrier is cleared. However, Celerity already has explicit synchronization points where this performance impact is anticipated:

The non-recurring implicit shutdown on queue destruction, where each node awaits all currently pending commands, and explicit barriers issued through `distr_queue::slow_full_sync()`.

```
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];
            });
        });
    } // await implicit synchronization shutdown from ~distr_queue()
    return host_ok ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing 4: Reference-capture workaround for retrieving buffer data. Necessary data transfers are requested through a host task accessor and awaited in the queue destructor.

Both of these synchronization points currently serve as a workaround to manually extract managed data using a host task. Listing 4 shows how the verification result from Listing 1 can be observed from the application thread by reference-capturing a result value and relying on the implicit shutdown as a synchronization point.

While functionally correct, this method is non-obvious, requires significant boilerplate, and can easily lead to undefined behavior if the application developer does not ensure that the reference-captured object outlives the synchronization point. In the following, we present a programming model allowing the extraction of arbitrary managed data data by-value and without the aforementioned hazards using existing synchronization points.

5.1 Data Extraction in Related Work

SYCL knows three ways of accessing buffer data outside of asynchronous tasks: By constructing a `host_accessor`, by explicitly synchronizing a host-coherent buffer via `handler::update_host()`, and by issuing a copy operation to a user-managed host data pointer via `handler::copy()`. Constructing a host accesser stalls the submitting thread until dependencies are satisified and memory coherence is established. Similarly, explicit copying must be followed up with a call too `event::wait()` to synchronize with the main threads. The latencies caused by both of these approaches is often more acceptable in SYCL's single-node context than it would be in Celerity's distributed setting.

In Legion, any task can access data produced by its sub-tasks without additional synchronization by awaiting the corresponding future. This execution model has no

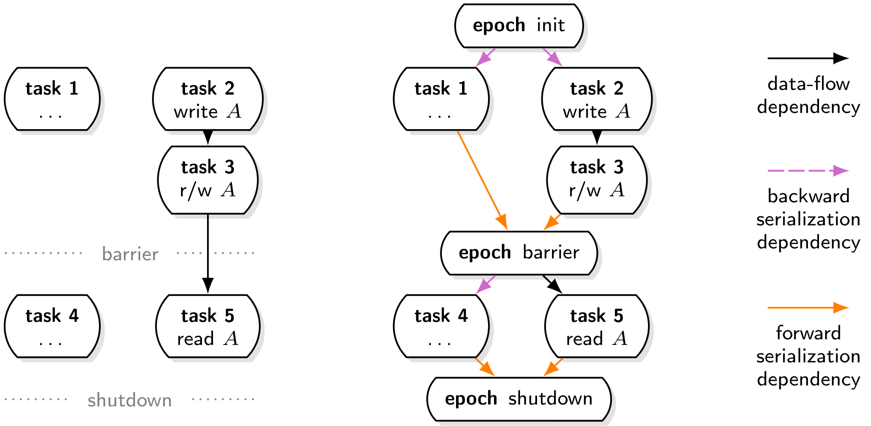

Fig. 3 Ad-hoc synchronization with broadcast commands (left, implied) and in-graph synchronization with epoch tasks (right). The barrier epoch becomes the effective producer of A, so task 5 receives a data-flow dependency on it. Serialization dependencies are inserted whenever no other transitive dependencies exist to the preceding or succeeding epoch to enforce correct temporal ordering

direct correspondence to Celerity, since Legion has no notion of a main thread of execution.

In Kokkos, some operations such as `parallel_reduce` will implicitly synchronize with the main thread when the output argument is a user-defined scalar variable. In all other cases, the user is expected to issue a *fence* operation in order to perform explicit synchronization, and/or establish memory coherence by constructing a data view that is accessible on the host side.

SkePU allows accessing data inside containers on the host side after manually flushing it to establish coherence within its lock-step execution.

5.2 Attaching Data Requirements to Synchronization Points with Epochs

In stable Celerity, barrier synchronization and convergence on runtime shutdown and is orchestrated using ad-hoc *control commands* which are sent to workers like regular commands, but are not part of the command graph.

While this enables a less involved implementation, it is not compatible with Celerity's graph-based mechanisms of orchestrating and tracking the necessary data migrations ahead of any synchronization point that wants to extract buffer data. The first step is therefore to integrate these synchronization points into the task and command graphs.

To that end, we introduce the concept of *epoch* tasks and commands that fully serialize execution on each node by placing appropriate dependencies in the graphs. In this model, each task or command (except for the first epoch) has exactly one preceding epoch, and no task or command can ever depend on an ancestor of its preceding epoch.

Figure 3 illustrates the approach. We begin by inserting an epoch task in to the task graph, from which the scheduler generates exactly one epoch command per node. To ensure correct temporal ordering, each epoch graph node receives a *forward serialization* true-dependency on the entire previous execution front, and all nodes without other true-dependencies (pure producers) receive a *backward serialization* true-dependency on the preceding epoch.

On each worker node, all synchronizing API calls block the application thread until the local executor reaches the epoch command.

Since dependency information from before an epoch is irrelevant for generating future command dependencies, as an optimization, all commands preceding an epoch can be eliminated from the graph once the epoch command has been issued to executors and the epoch can be regarded as the producer of any value currently available on that node.

5.3 Extracting Buffer Data and Host Object State with the Captures API

With epoch-based synchronization in place, the runtime can attach data dependencies onto synchronization commands and thus automatically generate data migrations for reading up-to-date buffer contents on every node.

To safely inspect buffer contents and host objects without introducing unnecessary additional submission stalls, we propose *captures*, a declarative API for attaching data requirements to shutdown and barrier epochs, which will be returned to the caller as snapshots by value.

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

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

template <typename T>
class capture<host_object<T>> {
    using value_type = T;
    explicit capture(host_object<T> ho);
};

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

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

Listing 5: Capture API around `celerity::distr_queue` (excerpt)

Listing 5 shows how the `distr_queue` class is extended to allow data extraction at existing synchronization points. The existing `slow_full_sync()` barrier primitive gains additional optional parameters, and shutdown convergence can be triggered explicitly using the `drain()` function. Both functions either accept a single *capture* or a tuple of captures and returns a single value or tuple of values as a result.

Each capture adds the necessary dependencies and data transfers to the generated epoch nodes and creates a snapshot of the data once the epoch has executed. As Celerity requires all MPI processes to perform the same sequence of API calls in order to allow centralized scheduling without worker-to-master communication, all nodes must currently request identical captures.

Listing 6 shows how the verification result from Listing 1 can be inspected in the application thread on the shutdown convergence explicitly triggered by `distr_queue::drain()`.

```
int main() {
    // ...
    buffer<bool> ok{1};
    q.submit([=](handler& cgh) { is_diag(cgh, C, ok); });
    return q.drain(capture{ok})[0] ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Listing 6: Data retrieval through the high-level `capture` construct. Data transfers are generated and awaited inside the `drain()` function.

Figure 4 shows the DAGs resulting from the capture-augmented Listing 6. With the switch to epoch-based synchronization, the graphs first shown in Fig. 1 now explicitly include the data requirement on the result buffer *ok*.

The introduction of the side-effect and capture–drain APIs eliminate all strictly necessary uses of by-reference captures in kernels that have been encountered during Celerity development so far.

6 Evaluation

While work focuses primarily on API expressiveness and programmability, the introduction of declarative side effects promises a performance improvement. Conversely, the introduction of epoch-based synchronization increases internal complexity, so the proposed changes demand further assessment.

We evaluated Celerity's performance on the Marconi 100 supercomputer in Bologna, Italy, which holds rank 18 of the TOP500 list as of November 2021[2]. Each node is powered by dual-socket IBM POWER9 AC922s and 256 GB of RAM, while inter-node communication is handled by dual-channel Infiniband EDR with a unidirectional bandwidth of 12.5 Gbit/s.

[2] https://www.top500.org/lists/top500/list/2021/11

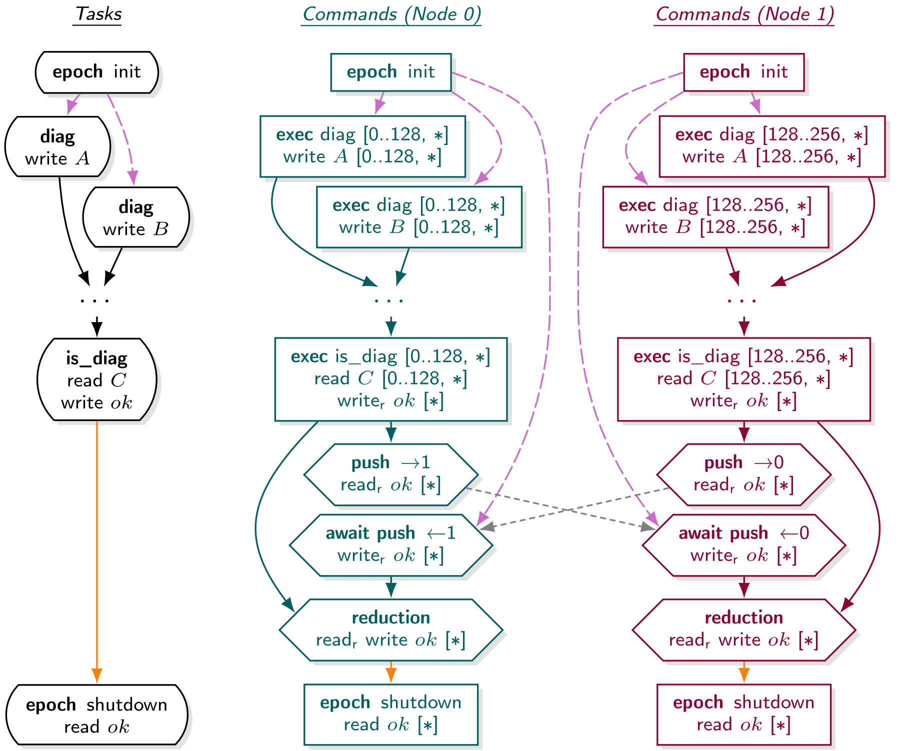

Fig. 4 The updated task and command graph, first seen in Fig. 1, after the introduction of epochs and capture-based data extraction following Listing 6. The reduction operation in `verify()` places the *ok* buffer in the *pending reduction state* indicated by the subscript in read$_r$ and write$_r$. A *reduction* command is generated as the result of the data requirement in the shutdown epoch which reverts the buffer back to the *distributed state*

Although this system is GPU-accelerated and Celerity is built around accelerator computation, no device kernels are executed as part of the benchmarks. Celerity unconditionally depends on a SYCL implementation for type definitions such as `sycl::range`, but results are expected to be independent of the backend choice. For the following evaluation, we compiled against the most recent development version of hipSYCL[3] on with Clang 12.0.1 as the host compiler and IBM Spectrum MPI 10.4.0 as recommended on Marconi 100.

For all multi-process benchmarks, we allocated 4 Celerity processes per cluster node through SLURM except for the 1- and 2-process case, where all processes were mapped to a single node. Since Celerity currently requires one process per compute device, this matches the typical configuration on a system with 4 GPUs per node. Each measurement was repeated 10 times.

[3] https://github.com/illuhad/hipSYCL/commit/1046a787

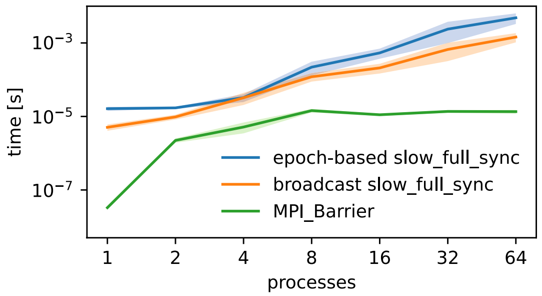

Fig. 5 Latency of barrier synchronization primitives (95% confidence intervals). `slow_full_sync` (blue and orange curves) has additional communication cost compared to the MPI baseline (green curve). Epoch-based synchronization (blue curve) further adds a constant overhead for graph generation that is amortized for higher node counts (Color figure online)

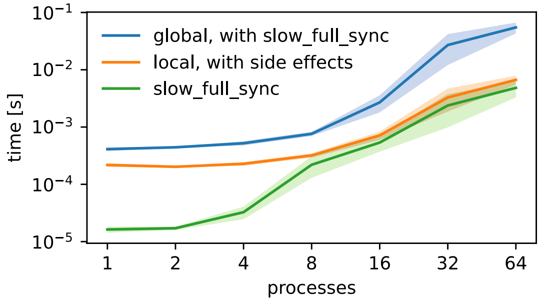

Fig. 6 Efficiency gains from replacing global barrier synchronization (blue curve) with side-effect dependencies (orange curve) to serialize a chain of 10 host tasks (95% confidence intervals). The local method does not require communication between worker nodes. Timings are measured using a single `slow_full_sync` barrier per run, which is included as a baseline (green curve) (Color figure online)

Figure 5 compares the latency of Celerity's `slow_full_sync` synchronization primitive against a synchronous `MPI_Barrier`. The latency of the Celerity implementation is elevated compared to the explicit MPI call as the broadcast-synchronization command or epoch command has to be sent to each worker before they can initiate their own `MPI_barriers`,. The epoch-based version is additionally delayed by graph generation overhead with a polynomial factor.

Figure 6 compares the overhead of serializing host tasks through barrier synchronization (the necessary workaround in stable Celerity) to the novel, local method using side effects. The benchmark measures a chain of 10 empty host tasks, serialized either through calls to `slow_full_sync` or side effects on a common host object. The local method, which only requires the introduction of scheduling dependencies, has much lower latency than the global barrier method, which introduces unnecessary synchronization between nodes.

Figure 7 shows the performance implications of introducing shutdown epochs on graph generation in the master node. We measured the time required to construct task and command graphs for 4 synthetic topologies: *chain*, an artificial

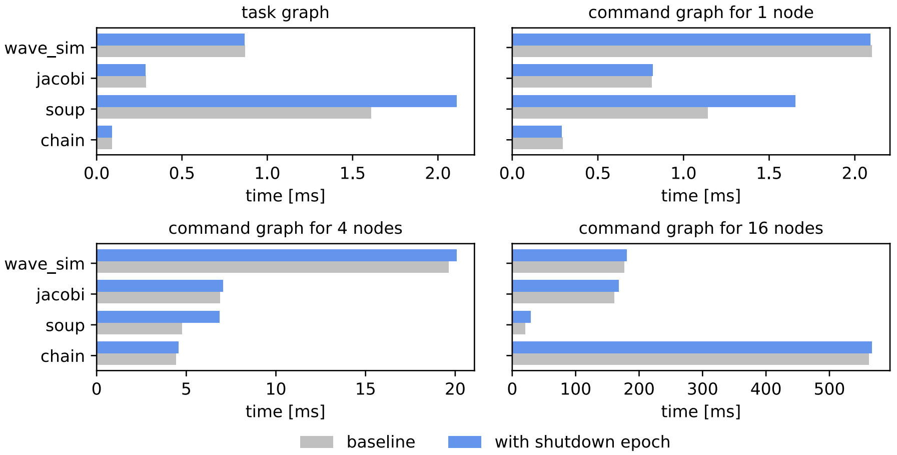

Fig. 7 Isolated time measurements for task and command graph generation on the master node. Introducing a shutdown epoch requires forward serialization dependencies which cause measurable overhead if the execution front is large. This is pronounced for the artificial and degenerate *soup* topology of a set of disconnected tasks

chain of command groups that require all-to-all communication between worker nodes; *soup*, an artificial, loose collection of disconnected tasks; *jacobi*, the task chain resulting from a 2D Jacobian solver; and *wave_sim*, the graph of a wave propagation stencil.

While accepting the extra work of generating a shutdown epoch will increase runtime unconditionally, this is especially pronounced for graphs with a large execution front, such as the artificial and degenerate *soup* topology. As expected, generating a forward serialization dependency from each task in the execution front and subsequently updating tracking structures has a measurable overhead. Graphs that more closely resemble real-world applications, which typically manifest as a chain of time steps, have a much smaller execution front and are therefore affected to a much smaller degree. As the number of nodes increases, scheduling is dominated by satisfying data dependencies instead. For adverse patterns such as the all-to-all communication required by the *chain* topology, this increase can be superlinear.

The approach to finding an optimal schedule on conflict graphs introduced in 4.4 has a worst-case runtime dependent on the number of allowed backtracking candidates. We measure the effect of this limit on synthetic conflict graphs which are generated by adding uniformly-sampled conflict edges to a set of disconnected command nodes. Fig. 8 visualizes the effects of varying the candidate limit, which confirms our choice of 100 as a reasonable trade-off.

To summarize, the introduction of declarative side effects has a net-positive performance impact, which will help overall system performance as we expect their use to arise repeatedly during application life cycle. As data extraction from runtime-managed structures is usually only relevant on shutdown, we argue that the demonstrated increase in synchronization latency has minimal impact on overall runtime and is justified by the improved programmability.

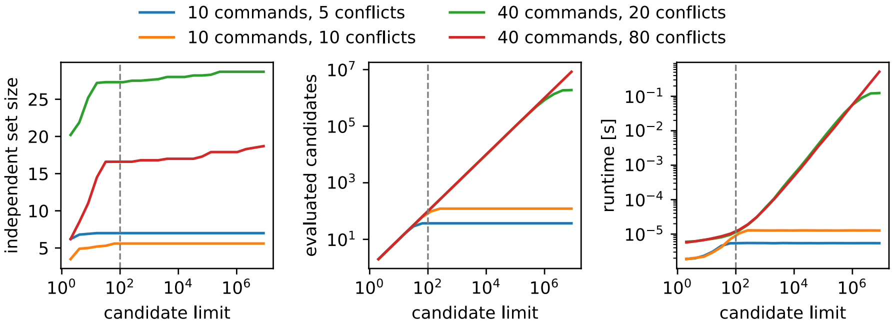

Fig. 8 Backtracking search for the largest conflict-free command set has exponential runtime behavior which must be cut short to cap scheduling latency in the executor. For a wide range of conflict-graph configurations, a limit of 100 backtracking candidates reduces worst-case execution time to the order of tens of microseconds, while backtracking beyond that limit will only yield diminishing returns in independent set size (Color figure online)

7 Conclusion

In this work, we have investigated how a graph-based distributed-memory runtime system can be extended with safe, declarative APIs to track dependencies on opaque node-local objects and transfer runtime-managed data back to the application thread to ease porting of legacy applications.

Specifically, we added the concept of *host objects* and *side effects* to the Celerity runtime system, a declarative mechanism for guarding access to and generating scheduling constraints around arbitrary node-local objects.

We further introduced the *captures* mechanism that allows observing snapshots of Celerity-managed data in the application thread without introducing unnecessary stalls in the asynchronous execution flow. In order to model the required data movements, existing synchronization points were fully integrated into the task and command graphs as *epochs*, which allow the expression of captured ranges as ordinary dependencies.

Experimentally, we confirmed that declarative node-local side effects are much more efficient than the previously necessary workaround employing barrier synchronization. While the epoch-based execution model required for data extraction can incur measurable overhead for command generation, this time is quickly amortized in a highly parallel setting.

Since evaluation was performed purely on synthetic benchmarks, the practical effects of the proposed extensions on programming effort and runtime performance of real-world applications remain to be seen.

7.1 Future Work

There is further potential in exploring the design space of the *captures* and *side effects* APIs. A mechanism to capture different buffer subranges on different nodes would allow a non-Celerity portion of the user program to continue operating in a distributed-memory fashion. Further, side effects are currently node-local by definition, but an application might also introduce cluster-wide side effects as well by writing to a parallel file system. Such global side effects should introduce edges in the Celerity graph model as well.

Acknowledgements This research is supported by the European High-Performance Computing Joint Undertaking (JU) project LIGATE under grant agreement No 956137.

Author Contributions Fabian Knorr wrote the manuscript text and produced all figures and code listings. All authors reviewed the manuscript.

Funding Open access funding provided by University of Innsbruck and Medical University of Innsbruck.

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Alpay, A., Heuveline, V.: SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL. In: International Workshop on OpenCL, pp. 1(2020)
2. Baker, B.S., Coffman, E.G., Jr.: Mutual exclusion scheduling. Theoret. Comput. Sci. **162**(2), 225–243 (1996)
3. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE (2012)
4. Bodlaender, H.L., Jansen, K.: On the complexity of scheduling incompatible jobs with unit-times. In: International Symposium on Mathematical Foundations of Computer Science, pp. 291–300. Springer (1993)
5. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distribut. Comput. **74**(12), 3202–3216 (2014)
6. Ernstsson, A., Ahlqvist, J., Zouzoula, S., Kessler, C.: Skepu 3: Portable high-level programming of heterogeneous systems and hpc clusters. Int. J. Parallel Prog. **49**(6), 846–866 (2021)
7. Ernstsson, A., Li, L., Kessler, C.: Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. Int. J. Parallel Prog. **46**(1), 62–80 (2018)
8. Golomb, S.W., Baumert, L.D.: Backtrack programming. J. ACM (JACM) **12**(4), 516–524 (1965)
9. Gschwandtner, P., Kissmann, R., Huber, D., et al.: Porting Real-World Applications to GPU Clusters: A Celerity and Cronos Case Study. In: 2021 IEEE 17th International Conference on eScience (eScience), pp. 90–98. IEEE (2021)
10. Keryell, R., Yu, L.Y.: Early experiments using SYCL single-source modern C++ on Xilinx FPGA: Extended abstract of technical presentation. In: Proceedings of the International Workshop on OpenCL, pp. 1–8 (2018)

11. Khronos Group: SYCL^{TM} 2020 Specification (revision 4). https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html (2021)
12. Robson, J.M.: Algorithms for maximum independent sets. J. Algorithm. **7**(3), 425–440 (1986)
13. Thoman, P., Jordan, H., et al.: CELERITY: Towards an Effective Programming Interface for GPU Clusters. In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp. 18–28 (2018)
14. Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: High-Level C++ for Accelerator Clusters. In: European Conference on Parallel Processing, pp. 291–303. Springer (2019)
15. Thoman, P., Tischler, F., et al.: The Celerity High-level API: C++20 for Accelerator Clusters. Int. J. Parallel Program. (accepted, to appear in 2022)
16. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., et al.: Kokkos 3: Programming model extensions for the exascale era. IEEE Trans. Parallel Distrib. Syst. **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
17. Xiao, M., Nagamochi, H.: Exact algorithms for maximum independent set. Inf. Comput. **255**, 126–146 (2017)