SN Computer Science (2024) 5:409
https://doi.org/10.1007/s42979-024-02749-w

ORIGINAL RESEARCH

Balancing Tracking Granularity and Parallelism in Many-Task Systems: The Horizons Approach

Peter Thoman[1] · **Philip Salzmann**[1]

Received: 29 September 2023 / Accepted: 24 February 2024
© The Author(s) 2024

Abstract
Reducing the need for users to manually manage the details of work and data distribution is an important goal of high-level many-task runtime systems. For distributed memory platforms this means that the runtime system has to keep track of both fine-grained task dependencies and data residency meta-information. The amount of such meta-information is proportional to the granularity of parallelism which needs to be managed, introducing a trade-off. More precise tracking of data state allows leveraging more opportunities for compute and transfer parallelism, while also introducing more overhead. As such, the fidelity of the information being tracked needs to be managed carefully, ideally without introducing additional latency, communication or substantial compute overhead. We present the "Horizons" approach, designed to fulfill these goals. Specifically, horizons allow for the effective and efficient management of parallelism and the coalescing of previous fine-grained tracking information while maintaining an easily configurable scheduling window with full information precision. As an additional benefit, they provide consistent cluster-wide decision points without requiring any inter-node communication, and effectively cap the size of state tracking data structures even in the presence of problematic access patterns. Experimental evaluation on microbenchmarks and dry runs demonstrates that horizons are effective in keeping the scheduling complexity constant, while their own overhead is negligible—below 10 µs per horizon when building a command graph for 512 GPUs. We additionally demonstrate the performance impact of horizons—as well as their low overhead—on a real-world application.

Keywords Dependency tracking · Task graph · Asynchronicity · Command generation · Gpu cluster

Introduction and Related Work

Modern high performance computing (HPC) hardware platforms feature many layers of parallelism, memory and communication. While they employ state-of-the-art methods to keep latencies as low as possible, the increase in computational throughput and bandwidth outpaces reductions in latency. Communication latency is thus an important limiting factor for performance, particularly at larger scales. As such, software for HPC systems is frequently designed to leverage asynchronicity as much as possible, enabling e.g. communication and computation overlapping.

Developing software which implements these techniques is a challenging endeavor, particularly while relying on the established de-facto standard approach to developing distributed GPU applications: "MPI + X", where the Message Passing Interface (MPI) [1] is combined with a data parallel programming model such as OpenMP, CUDA or OpenCL. Together with other factors such as lacking funding, this complexity leads to the development of new software for HPC typically being left to a select few HPC experts, and research is frequently performed using a small set of domain-specific software packages.

Parallel Runtime Systems

One promising avenue for improving programmability or enabling more flexible development of and experimentation

This article is part of the topical collection "Applications and Frameworks using the Asynchronous Many Task Paradigm" guest edited by Patrick Diehl, Hartmut Kaiser, Peter Thoman, Steven R. Brandt and "Ram" Ramanujam.

✉ Peter Thoman
peter.thoman@uibk.ac.at

Philip Salzmann
philip.salzmann@uibk.ac.at

[1] Distributed and Parallel Systems Group, University of Innsbruck, Technikerstraße 21a, Innsbruck 6020, Tirol, Austria

Published online: 06 April 2024

with high performance code for distributed memory GPU clusters are higher-level *runtime systems*. These typically introduce an API and custom terminology, as well as enabling ecosystems of tooling and derived software projects.

A notable example is StarPU [2], an extensible runtime system for programming heterogeneous systems. It offers a wide array of scheduling approaches, from simple FCFS policies, over work-stealing and heuristics, to dedicated schedulers for dense linear algebra on heterogeneous architectures [3, 4]. Nevertheless, StarPU's C API is rather low level and requires the explicit handling of data distribution when executing in cluster environments.

Legion [5] is a runtime system designed to make efficient use of heterogeneous hardware through highly configurable and efficient work splitting and mapping to the available resources. Its C++API is intricate and precise, with the explicit intent of putting performance first, before any programmability considerations, making it unsuitable for non-expert users.

HPX [6] is a C++ runtime system for parallel and distributed applications of any scale with a particular focus on enabling asynchronous data transfers and computation. Its heterogeneous compute backend supports targeting both CUDA and SYCL [7].

PaRSEC [8] uses a custom graph representation language called JDF to describe the dataflow of an application [9]. Either automatically generated or written by hand, this representation enables a fully decentralized scheduling model and automatic handling of data dependencies across a distributed system, although the initial distribution of data needs to be provided by the user.

The Celerity programming model [10] was designed to minimally extend the SYCL programming standard [11] while enabling automated distributed memory execution, specifically for clusters of GPU-like accelerators. It asynchronously generates and executes a distributed command graph from an implicit task graph derived from data access patterns [12].

While all the projects mentioned so far are primarily library-based runtime systems, a related category comprises those approaches which extend the grammar of existing programming languages, for example the pragma-based OmpSs [13]. Some related projects introduce entirely new languages altogether, such as Chapel [14], X10 [15] or Regent [16]. However, they, too, require an associated runtime system.

Finally, skeleton-based approaches trade some generality for a more tailored and user-friendly API, and this focus can also allow for simpler or higher-level data management and dependency tracking. Early implementations include the Quaff library by Falcou et al. [17], as well as the Orléans Skeleton Library by Noman and Loulergue [18]. Shigeyuki et al. [19] developed an early application of algorithmic skeletons to GPU computing. More recently, Ernstsson et al. [20] developed an extension to the SkePU skeleton programming model which lazily records the lineage of skeleton invocations, and applies tiling once partial results are actually required by the program.

What is clear from this broad and sustained interest is that ways to quickly develop distributed applications and efficiently experiment with different work and data distribution patterns are widely desired. Depending on the level of abstraction targeted by a system, data distribution and synchronization is either manual, semi-automatic or fully automatic.

Tracking Data State

For those systems which transparently manage distributed memory transfers and/or derive their task and command graphs from memory access patterns, *tracking* the state of data in the system at any given point in time is a significant challenge. On the one hand, all opportunities for asynchronous compute and transfer operations should be leveraged, but on the other hand, in an HPC context, scheduling and command generation also need to be sufficiently fast to scale to potentially thousands of cluster nodes. This is particular relevant when targeting strong scaling, as the time available to schedule individual tasks while maintaining efficient throughput shrinks even for fully asynchronous systems.

Tracking for some data access patterns—e.g. stencil-like computations—is quite manageable with a relatively simple approach, which means that some skeleton-based systems can largely circumvent these issues. However, a general high-level runtime system may be presented with more unusual patterns which can present additional difficulties. In particular, as we will outline in more detail in Sect. 2.3, *generative* patterns have the potential to overwhelm data tracking. However, even relatively straightforward many-task processing of large data sets can become problematic when individual tasks are sufficiently short-lived, as described in Sect. 2.4.

The Horizons Concept

In this work, we present *Horizons*, a concept which manifests as a special type of node in task or command graphs for distributed parallel runtime systems. The basic working principle of Horizons is that, when specific conditions are met during the live generation of a task graph, a Horizon node is introduced. Crucially, at this point, that particular Horizon does not yet have any impact on subsequent parallelism or the granularity of dependency tracking: it only becomes *active* at a later point, when it subsumes details about prior computations.

Core design goals and features of horizons include:

- Maintaining asynchronous command generation and execution.
- Allowing for a configurable trade-off in the level of detail regarding data state available for command generation.
- Allowing for a configurable cap on the degree of task-level parallelism concurrently managed by the system.
- Never directly introducing a synchronization point.
- Requiring no additional inter-node communication.

Our implementation of Horizons in the Celerity runtime system achieves all of these goals. Section 2 provides a concise overview of the Celerity system, and describes the types of access patterns and task graph structures which Horizons are particularly effective at managing. Section 3 explains how Horizons are generated, managed and applied, illustrating their impact on command generation. In Sect. 4 we present an in-depth empirical evaluation of the implementation of Horizons in Celerity, including both microbenchmarks and real-world applications. Finally, Sect. 5 concludes the paper.

Background

The Celerity Runtime System

Celerity is a modern, open C++ framework for distributed GPU computing [12]. Built on the SYCL industry standard [11] published by the Khronos Group, it aims to bring SYCL to clusters of GPUs with a minimal set of API extensions. A full overview of the SYCL and Celerity APIs is beyond the scope of this paper,[1] so in this section we will focus on how Celerity extends the data parallelism of SYCL kernels to distributed multi-GPU execution, and the data state tracking requirements this induces for the runtime system.

A typical SYCL program is centered around *buffers* of data and *kernels* which manipulate them. The latter are wrapped in so-called *command groups* and submitted to a *queue*, which is then processed asynchronously with respect to the host process. Crucially, buffers are more than simple pointers returned by a `malloc`-esque API: they are accessed through so-called *accessors*, which are declared within a command group before a kernel is launched. Upon creating buffer accessors, the user additionally has to declare *how* a buffer will be accessed, i.e., for reading, writing or both. This allows the SYCL runtime to construct a *task graph* based on the dataflow of buffers through kernels.

SYCL—in the same fashion as CUDA and OpenCL—abstracts the concept of a (GPU) hardware thread: it allows users express their programs in terms of linear-looking kernel code, which is invoked on an N-dimensional range of work items. Celerity extends this concept to distributed computation. While Celerity kernels are written in the same way as in SYCL, they can be executed across multiple devices on different nodes, with all resulting data transfers handled completely transparently to the user.

The most fundamental extension to SYCL introduced by Celerity are *range mappers*, functions that provide additional information about how buffers are accessed from a kernel. By evaluating these range mappers on sub-domains of the execution range, the Celerity runtime system infers which parts of a buffer will be read, and which ones will be written—at arbitrary granularity.

```
1  distr_queue queue;
2  auto rg = range<2>(512, 512);
3  buffer<float, 2> buf_in(hst_in.data(), rg);
4  buffer<float, 2> buf_out(rg);
5
6  queue.submit([=](handler& cgh) {
7      accessor in{buf_in, cgh,
8                  access::one_to_one{}, read_only};
9      accessor out{buf_out, cgh,
10                  access::one_to_one{}, write_only};
11     cgh.parallel_for(rg, [=](item<2> itm) {
12         out[itm] = in[itm] * 2.f;
13     });
14 });
```

Listing 1: A basic matrix operation in Celerity

Tasks

Listing 1 shows an example of a simple matrix operation implemented in Celerity. To transparently enable asynchronous execution, all compute operations in a Celerity program are invoked by means of a queue object. In the first line of Listing 1, this queue of type `celerity::distr_queue` is created. Subsequently, two two-dimensional buffer objects are created, with the former initialized from some host data `hst_in`.

The central call to `distr_queue::submit` on line 6 submits a command group, which creates a new *task* that will later be scheduled onto one or more GPUs across the given cluster. The index space of this task (the 2D range `rg` in this example) will be split into multiple *chunks* that can be executed by different workers. The provided callback (the kernel code) is subsequently invoked with an index object (`itm`) of corresponding dimensionality, which is used to uniquely identify each kernel thread.

Range Mappers

This program closely resembles a canonical SYCL program, with one important difference: Each constructor for `celerity::accessor` is provided with a *range mapper*, in this case a two-dimensional instance of the `one_to_one` mapper. This particular range mapper indicates

[1] Readers may refer to [10–12, 21], as well as the Celerity documentation at https://celerity.github.io/docs/getting-started.

that every work item of the 512×512 global iteration space accesses exactly one element from `buf_in` and `buf_out` each—precisely at the work item index.

In general, range mappers can be arbitrary user-defined functions, with a small set of constraints, primarily related to overlapping output accessors not being valid. This allows for a high degree of flexibility, while the included *one-to-one*, *slice*, *neighborhood* and *fixed* range mappers reduce verbosity and make user programs more readable in common cases.

Execution Principle

The actual execution of Celerity program involves three major steps, each of which proceeds asynchronously with the others in a pipelined fashion: (i) task graph generation, (ii) command graph generation, and (iii) execution.

The *task graph* encapsulates the behaviour of the program at a high level. Essentially, every submission on the queue is represented by a task, and dependencies are computed based on each task's accessor specification. In the lower-level *command graph*, task executions are split up for each GPU, and the required commands for transfers are also generated. Therefore, the number of nodes in the command graph is generally larger than the task graph by a factor of at least $O(N)$. These commands are finally executed on a set of parallel execution lanes.

Summary

While Celerity can be considered a task-based runtime system, its default mode of operation differs significantly from the more common approach taken, particularly in distributed memory settings. Instead of leaving the choice of how to split work or data fully or partially to the user, the Celerity approach is to consider each data-parallel computation as a single *splittable* task. The runtime system is provided with sufficient information, primarily by means of accessors and their associated range mappers, to split these tasks in various ways and distribute them across the cluster.

Data State Tracking

From a theoretical point of view (in practice, custom acceleration data structures are employed[2]), the runtime system has to track the state of each individual data element, in order to be able to build a data dependence graph and construct the necessary transfer commands. These data structures—one for each buffer managed by the runtime system—track the last operation which wrote to any particular data element. As such, they need to be updated for each write access requested by a program, and are queried whenever read access to a buffer is required by a kernel. The performance of these operations is thus crucial to the overall efficiency of the runtime system.

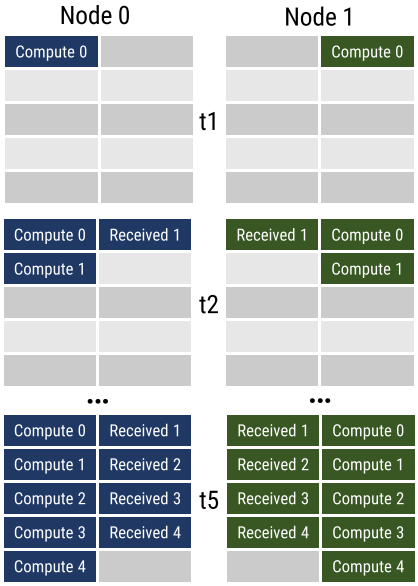

Fig. 1 State tracking with a generative access pattern

For data access patterns common in many physical simulations and linear algebra, the number of individual regions which need to be tracked generally scales with the number of GPUs in the system, as all elements are replaced in each successive time step or iteration of the algorithm. In these cases, *distributed command graph generation* [12], which only locally tracks the perspective on the total system state which is required for the operations on one node, is highly effective and can scale up to thousands of GPUs. However, it can not mitigate tracking data structure growth with some more complex access patterns, or when the user application generates a very large number of tasks operating on individual portions of the same buffer.

Generative Data Access Patterns

In some domains, data access patterns iteratively generate new data over the execution of a program, and might refer to all the generated data in some subsequent computations. We call these access patterns *generative*, and they present a unique challenge for data state tracking.

Figure 1 illustrates the state of the tracking data structure of a 2D buffer with a generative data access pattern running on two nodes, after one, two and 5 time steps. In this example pattern, every time step one row of the buffer is generated in parallel, and every subsequent time step requires all previously computed data. For this example, we assume a static 50:50 split in computation between the two

[2] https://github.com/celerity/celerity-runtime/pull/184.

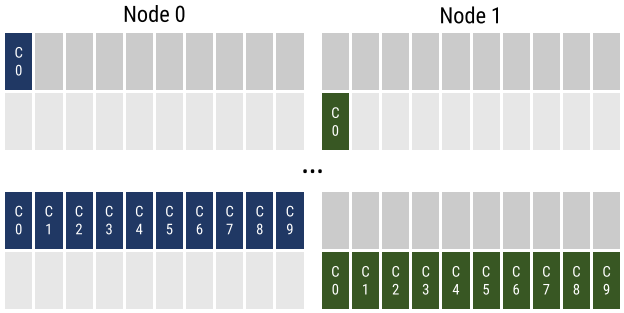

Fig. 2 State tracking for a many-independent-task processing pattern

participating nodes. As such, after timestep `t1`, each node will push its computed data to the other in order to perform the computation at `t2`, and so forth.

With N GPUs, this means that the tracking data structure will contain $O(N \cdot t)$ separate last writer regions at time step t. Even with a highly efficient data structure, the time to query the full previously computed area (e.g. all rows up to $t - 1$) will thus scale linearly (at best) with the number of time steps.

A simple solution to this particular problem might appear to be to only track whether some data is available locally or on some other node, rather than precise information on which command will have generated it. While this would result in a functionally correct execution, it also implies a complete sequentialization of the command graph up to the most recent data transfer. This would prevent e.g. automatic communication and computation overlapping, the asynchronous sending or receiving of many separate data chunks, or the parallel execution of several independent kernels accessing the same buffers. Horizons provide an elegant solution to this dilemma.

Explicit Many-Task Processing of Large Data Sets

Generative access patterns are not the only case in which some sort of consolidation of the data state tracking information is desired. While many data-parallel algorithms for which Celerity is used in practice lead to a relatively small number of user-defined task invocations—with parallelism provided by splitting the execution ranges of these tasks as required—some algorithms benefit from parallelism being expressed as individual, independent tasks at the user level. One of the simplest possible examples of this pattern that is also relevant to the data state tacking discussion is when a large number of individual parallel tasks independently process distinct sub-ranges of a given buffer.

Figure 2 illustrates the result of buffer tracking in such a scenario. In the upper pair of illustrations, only one task has been processed ("Compute 0" was chosen in this case, though, since the tasks are independent, this choice is arbitrary), and has computed the upper half of the first column of the buffer on node 0, and the lower half on node 1. The state of the internal tracking structure after all tasks have been processed is shown in the lower part of the figure.

For M user-defined, independent tasks, the tracking data structure will contain $M + 1$ entries in this case. Note that unlike the generative pattern, this complexity does not grow with the number of nodes or GPUs N, since there is no communication involved in this sample. In a real-world use case subsequent computation steps would likely access larger parts of the buffer, requiring data transfers, but for our purposes of analyzing the impact of Horizons this simple scenario is already sufficient.

Horizons

Figure 3 illustrates a simplified view of the command graph generated for the first five iterations of a computation with a basic generative data pattern (see Sect. 2.3) scheduled on two nodes/GPUs. It includes compute commands, as well as data push and receive commands. As each row of the involved data buffer is generated by subsequent time steps, the number of dependencies in the command graph scales with the iteration count, as indicated in the figure at location ①.

Horizons solve this issue by selectively coalescing data structures and dependencies, asynchronously and with a configurable level of detail being maintained. From a high-level point of view, "Horizons" describe synchronization points during the execution of a program, in both the task and command graph.

However, it is crucial to note that *no single horizon implies full and immediate synchronization*. Instead, at any point during the scheduling and command generation for a program (after the startup phase), two relevant Horizons exist: the older of the two is the most recent Horizon which was *applied*, which means that all tracking data related to commands scheduled before it was subsumed and coalesced; the newer of the two is the most recent Horizon to be *generated*—it will eventually be applied, but as of now it imposes no synchronization. As such, the window between the applied Horizon and the current execution front maintains all opportunities for parallel and asynchronous execution and fine-grained scheduling which would be available without Horizons.

For clarity, we split our detailed description of the horizons concept into three parts: (i) the decision making procedure, (ii) horizon generation, and (iii) horizon application.

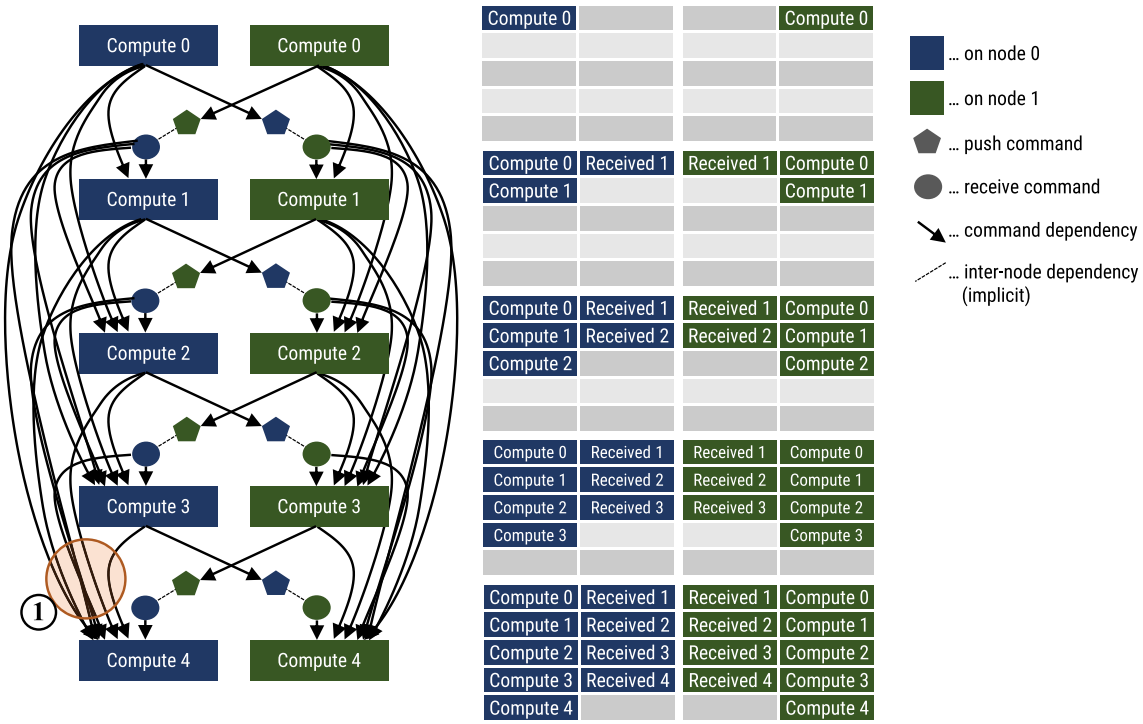

Fig. 3 Command graph and buffer tracking for a generative data pattern

Horizon Decision Making

The decision on whether and where to generate a new Horizon is made during task graph generation. There are two fundamental scenarios in which Horizon generation is useful:

- When working on a sequence of dependent tasks, as represented by the generative data access pattern example (see Sect. 2.3). We call these Horizons *depth-triggered*, as they simplify deep task graphs, that is, ones where a significant portion of all tasks is on the critical path. We discuss the circumstances under which these types of Horizons are generated in Sect. 3.1.1.
- When a very large number of independent parallel tasks are scheduled by the user code, as represented by the many-task data tracking example (see Sect. 2.4). We call these types of Horizons *breadth-triggered*, since they simplify tracking in cases where the task graph is very wide, i.e. when there are many independent asynchronous tasks. We discuss when precisely these Horizons are generated in Sect. 3.1.2.

Note that these two cases are distinct only in the conditions which trigger the generation of a Horizon. The actual generation and application of the Horizons, constituting the vast majority of their implementation, are shared between both cases.

Depth-Triggered Horizons

Whenever new nodes are inserted into the task graph, they are associated with the current critical path length C from the start of the program, computed in $O(P)$ from their P predecessors. Additionally, we also track the most recent Horizon position H, where e.g. $H = 5$ means that the most recent Horizon was generated at critical path length 5.

A dynamically configurable value $S > 0$, the *Horizon Step Size*, then defines how frequently new depth-triggered Horizons are generated. A new Horizon task is inserted into the task graph every time the critical path length grows by S, that is, whenever

$$C > H \quad \wedge \quad (C - H) \mod S \equiv 0 \quad .$$

Breadth-Triggered Horizons

The goal of breadth-triggered Horizons is to allow for consolidating tracking data in cases where a very large number

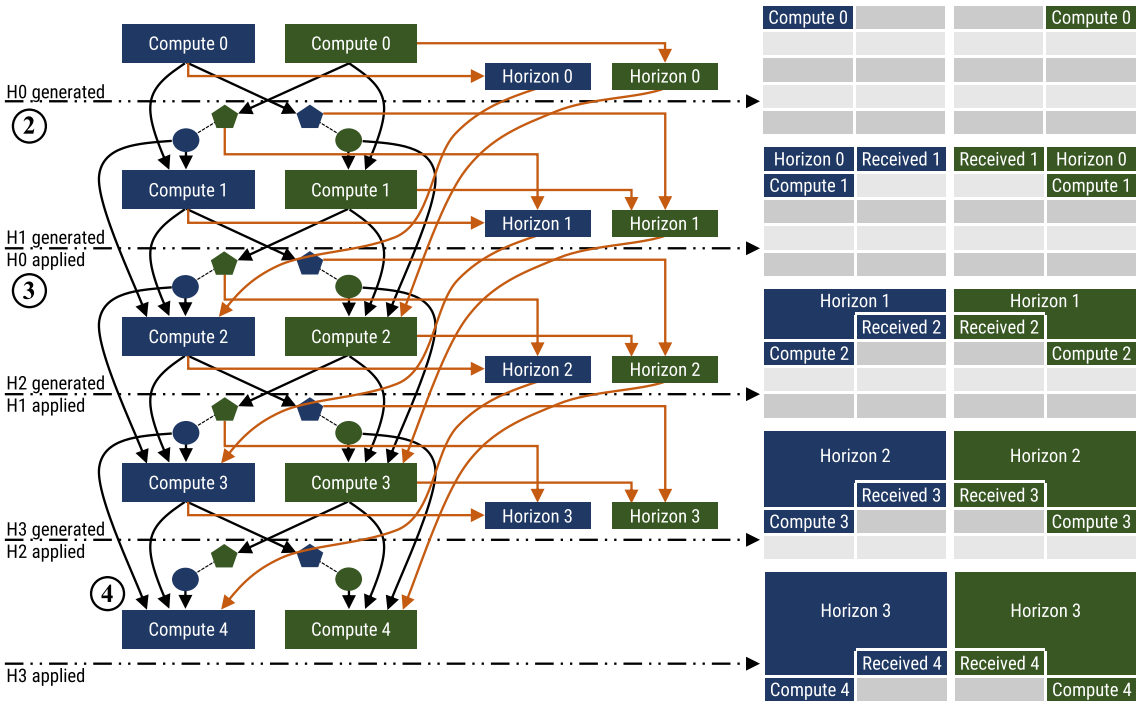

Fig. 4 Command graph and buffer tracking for a generative data pattern with Horizons, using the minimum step size $S = 1$

of independent asynchronous tasks are generated by the user program, as exemplified by the simple pattern described in Sect. 2.4.

To directly address this use case, a trigger condition based on the size of the current *execution front* of the task graph is employed. The execution front contains all commands for which there currently are no successors, and is easily tracked throughout the task generation process with only minimal, constant overhead per task. This execution front is also already required for other operations on the task graph, so in practice there is no additional cost for implementing breadth-triggered Horizons other than the check itself.

When this execution front is available, the specifics of breadth-triggered Horizons are extremely simple: given a *Maximum Task Execution Front Size* of M_E, a new Horizon task is inserted into the task graph iff, after the generation of any new task, the current task front size exceeds M_E. Note that therefore the smallest meaningful value for M_E is 2.

Horizon Generation

When command generation encounters a new Horizon task, a corresponding per-node horizon command is generated. This command has a true dependency on each of the nodes in the entire current per-node execution front of the command graph. As a consequence, after each Horizon generation, the execution front contains only the horizon command.

To illustrate this principle, Fig. 4 shows the generation of Horizon 0 at ② and Horizon 1 at ③. Note that the commands associated with the former only depend on the initial compute commands of each respective node, while all later horizons depend on both the most recent compute and receive commands on their respective node.

Whenever a Horizon is generated for e.g. critical path length C, if a previous Horizon generated for critical path length $C - S$ exists, it is *applied*.

Horizon Application

Applying a Horizon is arguably the most crucial step of the process, as it is what allows for the consolidation of tracking data structures, and therefore a reduction in dependencies. Crucially, Horizons are always applied with a delay of one step, which maintains fine-grained tracking and therefore asynchronicity for the most recent group of commands.

When a given Horizon is applied, all references to previous writers in the tracking data structures which refer to commands preceding the Horizon are updated to instead refer to the Horizon being applied. In the example shown in Fig. 4, at ③ Horizon 0 is applied, thus replacing Compute 0 in the tracking data structures. Note that all commands

are consecutively numbered internally, so there is no graph traversal or other complex operation required to determine whether a given Horizon should replace any given entry in the tracking data structure.

After Horizon application, during any subsequent command generation steps, dependencies which would have been generated referring to commands prior to the Horizon boundary directly will instead refer to the appropriate Horizon. A comparison between ④ in Fig. 4 and ① in Fig. 3 illustrates how Horizons thus maintain a constant command dependency structure with generative data access patterns.

Summary and Features of the Horizon Approach

The Horizon approach as presented has the following properties:

1. It is independent of the specifics of the data access pattern, and its induced dependency graph.
2. It maintains a constant maximum on the per-node dependencies which need to be tracked.
3. A window of high-fidelity dependency information is maintained, and the size of this window can be adjusted by setting the step size S.
4. An easily adjustable degree of task parallelism is maintained, which can be modified by setting M_E.
5. Horizon triggering and command generation are both efficient, as the required information (current critical path length and execution front) can be tracked with a small fixed overhead during the generation of each graph node.
6. Horizon application is highly efficient, as due to the numbering scheme of commands a simple integer check suffices (no graph traversal is required).
7. No additional communication is required for any step in the process, all nodes can continue to proceed completely asynchronously.

We show the practical value of this scheme and demonstrate the impact of varying the parameters S and M_E in Sect. 4.

Evaluation

In this section, we present empirical results which illustrate the effectiveness and efficiency of the Horizon approach as it is currently implemented in the Celerity runtime system. We first show microbenchmarks of simple generative data patterns to precisely track the impact of Horizon step sizes (S) on command generation times.

Secondly, we benchmark a representative explicit many-task processing pattern, illustrating the relationship between the maximum task execution front size (M_E) and the overall graph generation time of the Celerity system.

Thirdly, we demonstrate that Horizons have negligible overhead at both small and large scales, and can even be beneficial for programs without generative access patterns or many user-defined tasks, using *dry-run* benchmarks. In dry-run mode, the Celerity runtime system performs all the scheduling and command generation work of a real program, but skips the execution of its kernels. This allows us to quickly execute benchmarks on a large—simulated—number of nodes and observe the impact of various optimizations and data structure choices on task and command graph generation performance, without occupying a large-scale HPC cluster.

Finally, we show the impact of Horizons on a full run of a real-world application in room response simulation, which exhibits a generative access pattern.

Experiment Setup

The hardware and software stack for the microbenchmarks and dry-run benchmarks comprises a single node featuring an AMD Threadripper TR-2920X CPU, running Ubuntu Linux 22.04. As the dry-run benchmarks need no additional hardware and are relatively quick to complete, 30 runs of each configuration were performed and the median result is reported. The variation observed across all results of these runs was less than 3%, therefore we omit it from the charts for readability. The real-world application benchmarks were performed on the Marconi-100 supercomputer[3] at CINECA in Bologna, Italy, with a lower count of 5 runs each due to hardware availability limitations. Marconi-100 is an IBM Power9 system with 4 NVIDIA V100 GPUs per node.

Celerity is available on Github, with the specific revision `f190da3`[4] being used for this publication. The Celerity distribution also includes the source of the microbenchmarks discussed in the following sections. The benchmarks were performed with the hipSYCL (now AdaptiveCPP) SYCL implementation.[5]

Generative Access Microbenchmarks

Figure 5 shows the per-iteration time spent on command generation for a cluster of 512 GPUs, in a microbenchmark of a 2D generative access pattern, with different Horizon configurations. Note that this plot is logarithmic in the Y axis, to better capture the differences between the settings.

[3] https://www.top500.org/system/179845/

[4] https://github.com/celerity/celerity-runtime/commit/f190da3db723ac78b85590467ccb36c51150cb00

[5] https://github.com/AdaptiveCpp/AdaptiveCpp

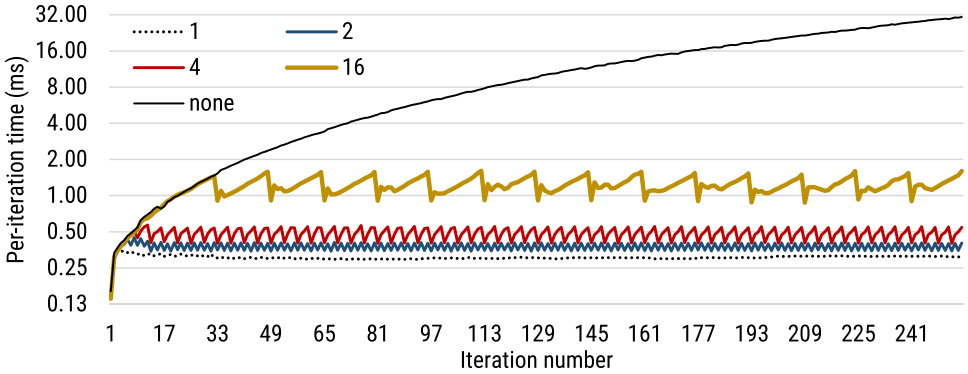

Fig. 5 Per-iteration time for 2D generative access micro-benchmark; each line shows a different horizon step setting S (or no Horizons), as indicated in the legend

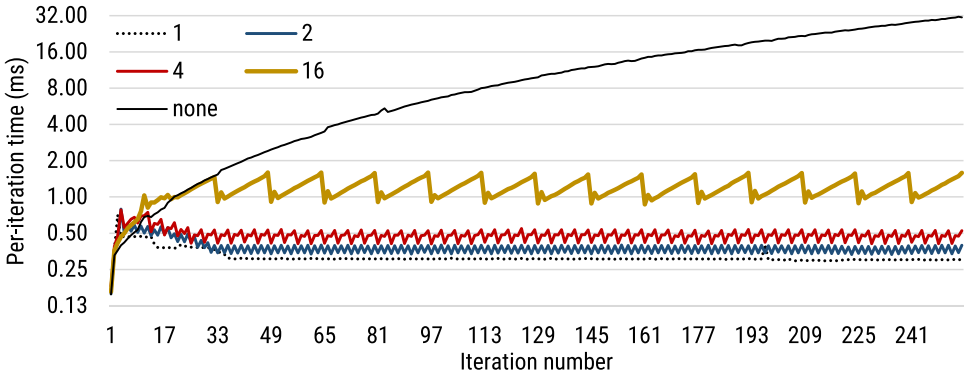

Fig. 6 Per-iteration time for 3D generative access micro-benchmark; each line shows a different horizon step setting S (or no Horizons), as indicated in the legend

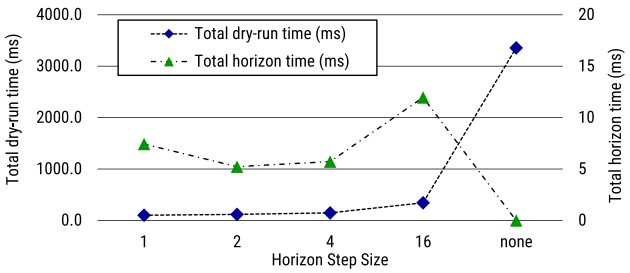

Fig. 7 Total times for 2D generative access microbenchmark

Without horizons (the solid black line), the command generation overhead grows with each iteration of the benchmark, as expected due to the growth of dependencies outlined in Sect. 3. With a Horizon step size of 16, a drop in overhead is seen for the first time in iteration 33, as the Horizon generated after iteration 16 was applied in iteration 32. The same pattern is visible for the smaller step sizes 4 and 2, but at a smaller scale. With step size 1, the per-iteration time is almost entirely flat.

Figure 6 provides the same view on a benchmark of a 3D generative access pattern. We note that the behavior once the first horizon has been applied is almost identical to the 2D case, despite the difference in buffer and therefore tracking dimensionality. In the initial few iterations, a slightly higher per-iteration time can be observed. This is due to the slightly more complex and nested data structures involved in tracking data dependencies for 3D access patterns, which incur a somewhat larger initial overhead until they are fully built and cached.

Figure 7 illustrates the total execution time (blue diamond, left axis) and total time spent on horizon generation and application (green triangle, right axis) of the same microbenchmark. Besides the remarkable decrease in the overall benchmark runtime due to Horizons, which matches the per-iteration results, the behaviour of the Horizon overhead is interesting: when generating a Horizon every time step, the overhead is slightly higher, then it drops, but increases again at $S = 16$. This result can be explained by the fact that, although Horizons are generated far less frequently, the accumulated complexity in the data tracking structure and command graph after 16 iterations makes Horizon generation significantly more expensive. However, even in this case, the Horizon generation overhead only amounts to a total of 12 ms over 256 iterations.

We also measured this data for 1D and 3D generative access patterns, and the shape across different Horizon step sizes is almost identical to the 2D case. The total times for the 1D pattern are generally lower—roughly 1/3rd of the 2D time—while the 3D times are slightly higher. This corresponds to the general expected tracking overhead, independently of Horizons, and proves that they

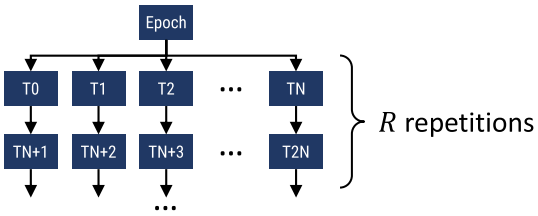

Fig. 8 Many-independent-task microbenchmark structure

work equally well for generative access patterns regardless of the dimensionality of the buffer being tracked.

Many-Task Parallelism Microbenchmarks

While the previous microbenchmarks demonstrate the effect of horizons for generative data access patterns, they do not cover the independent many-task parallelism case as outlined in Sect. 2.4. In order to analyze this pattern, we created a representative many-task application which generates N parallel, independent tasks each working on its own chunk of a buffer, with R repetitions. Figure 8 depicts the high-level task dependency graph modeled by this application.

We performed a benchmark of this application with $N = 15000$ and $R = 5$, and the resulting per-task generation times are depicted in Fig. 9. The first result which may initially be surprising is that the per-task time with no horizons (the dotted line) is independent of the number of nodes the application is scheduled on. This is due to the task-count-induced tracking overhead with $N = 15000$ completely dominating the task generation time. Since there is no communication required by this benchmark structure, the number of involved nodes does not have a significant impact.

Given this understanding of the application, it might then be counter-intuitive that, once more frequent breadth-triggered horizons are enabled by setting M_E to smaller values, the number of nodes starts to actually factor into the overhead. Two main factors explain this behaviour:

- The introduction of Horizons at reasonably low settings for M_E reduces the overall tracking overhead, making other aspects which are more dependent on the total node count more relevant again.
- With a larger number of nodes, there is a larger minimum number of distinct data fragments which need to be tracked, regardless of the chosen threshold M_E. This means that the potential for tracking overhead reduction due to Horizon application is lower with a larger number of nodes.

Finally, the results indicate that choosing a very high threshold M_E can actually be detrimental to performance, compared to not using Horizons at all. However, a setting of e.g. $M_E = 256$ extracts a very large benefit from Horizons, with an overhead reduction of factor 8.2, 7.9, and 2.5 respectively for 1, 16 and 256 Nodes. Such a setting does, due to the asynchronous application of Horizons (see Sect. 3.3), still allow for 512 high-level tasks to be running asynchronously in the system, each of which can be split to further increase available parallelism.

Overhead for Other Parallel Patterns

For Horizons to provide a suitable solution for coalescing dependencies in a *general* runtime system, they need to have no significant negative performance impact in applications with non-generative access patterns and a smaller number of highly splittable tasks. Figure 10 summarizes results for two such applications: *WaveSim*, a 2D stencil computation, and *Nbody*, an all-pairs N-body physics simulation.

In the *WaveSim* application, the overall impact of Horizons is negligible: the total dry-run time varies by less than 3 ms with and without Horizon use and with different step sizes, and less than 0.5 ms outside of the extreme Horizon step size setting of $S = 1$. For the *Nbody* benchmark, there is a more notable impact—although it is still minor compared to applications with generative patterns.

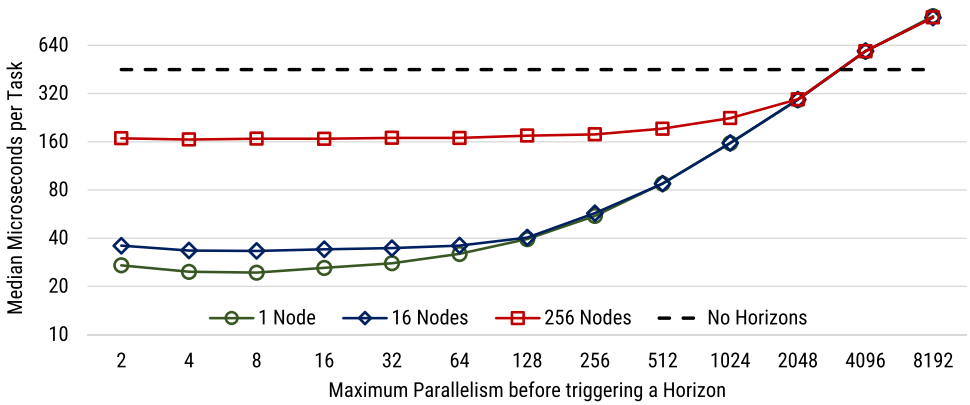

Fig. 9 Per-task time for independent many-task benchmark; each line shows a different number of cluster nodes, as indicated in the legend; the X axis depicts the maximum execution front size M_E

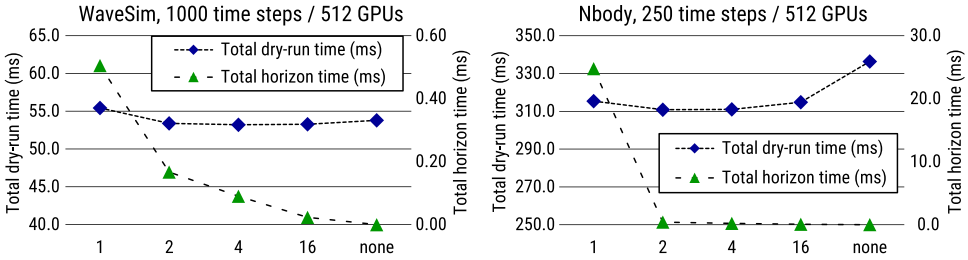

Fig. 10 Horizon impact and overhead for two non-generative applications. X-axis shows Horizon use/step size

Two particular results stand out: the horizon overhead at step size 1, and the fact that the introduction of Horizons has a positive overall performance impact on the order of 7%. The former is explained by the particular structure of this application, which has two different types of main compute kernels, one of which features only a one-on-one read dependency that can be satisfied locally, while the other requires all-to-all communication. With a Horizon step size of 1, Horizons are inserted after the latter kernel, requiring a much larger number of dependencies. The overall positive impact of Horizons can be explained by their application being utilized to clean up various internal data structures, which can be slightly beneficial even in non-generative cases.

Real-World Application

To confirm the data obtained using microbenchmarking and dry-run experimentation, Fig. 11 shows the result of a strong scaling experiment with the Celerity version of RSim [22], a room response simulation application, over 1000 time steps. RSim computes the spread of a light impulse through a 3D space modeled as a set of triangles. In each time step, the incident light for each triangle depends on the radiosity of all other triangles visible from it, at a point in time that depends on the spatial—and therefore also temporal—distance between the two triangles. As such, the main computational kernel of RSim exhibits a generative access pattern in which subsequent time steps depend on the per-element radiosity computed in prior time steps.

We compare the current default setting of the Celerity runtime system, Horizon step size 2, with no Horizons. In the latter case, with 4 and more GPUs, command generation overhead starts to dominate the overall simulation run time. With Horizons, near-linear strong scaling is maintained up to 16 GPUs, and strong scaling continues to 32 GPUs. The remaining drop from linear scaling, particularly at 32 GPUs, is not caused by data location tracking overhead in the runtime system. Instead, it can be attributed to the fact that this is a strong scaling experiment with fixed-size per-timestep communication requirements.

Conclusion

In this paper, we have presented *Command Horizons*, an approach to limiting the data tracking and command generation overhead in data-flow-driven distributed runtime systems with automatic communication, particularly in the presence of generative data access patterns or a very large number of user-defined asynchronous tasks, while maintaining asynchronicity.

Based on their current implementation in the Celerity runtime system, we have demonstrated that Horizons can be generated and applied very efficiently and with low overhead in a variety of applications, and that they are effective at capping command generation overhead and the complexity of internal tracking data structures at a stable level.

Horizons also have additional applications, e.g. in providing a consistent distributed state for decision making without requiring communication, which we hope to explore in the future.

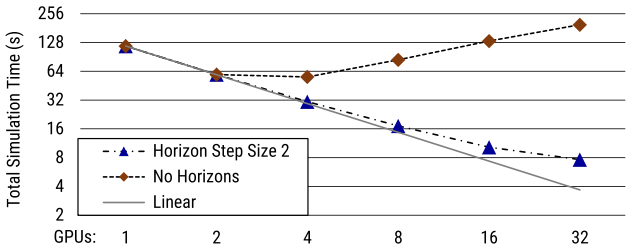

Fig. 11 Horizon impact on RSim application

Acknowledgements This project has received funding from the European High Performance Computing Joint Undertaking, grant agreement No 956137, as well as the Austrian Research Promotion Agency (FFG) via the UMUGUC project (FFG #4814683, 903595).

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

Data availability The relevant implementation source code is available in the Celerity github repository, which is already linked in the paper. (See "experiment setup").

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Message Passing Interface Forum. MPI: a message-passing interface standard, version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf Accessed 05 Feb 2019.
2. Augonnet C, Clet-Ortega J, Thibault S, Namyst R. Data-aware task scheduling on multi-accelerator based platforms. In: 2010 IEEE 16th international conference on parallel and distributed systems; 2010.
3. Thibault S. On runtime systems for task-based programming on heterogeneous platforms. Thesis, Université de Bordeaux; 2018. https://hal.inria.fr/tel-01959127 Accessed 24 Sep 2020.
4. Kumar S. Scheduling of dense linear algebra kernels on heterogeneous resources. PhD Thesis, Université de Bordeaux; 2017. https://tel.archives-ouvertes.fr/tel-01538516. Accessed 19 Nov 2020.
5. Bauer M, Treichler S, Slaugther E, Aiken A. Legion: expressing locality and independence with logical regions. In: 2012 international conference for high performance computing, networking, storage and analysis (SC). IEEE, New York; 2012.
6. Heller T, Diehl P, Byerly Z, Biddiscombe J, Kaiser H. Hpx—an open source C++ standard library for parallelism and concurrency. In: Proceedings of OpenSuCo, vol. 5; 2017.
7. Copik M, Kaiser H. Using sycl as an implementation framework for hpx. compute. In: Proceedings of the 5th international workshop on OpenCL, pp. 1–7; 2017.
8. Bosilca G, Bouteiller A, Danalis A, Faverge M, Herault T, Dongarra JJ. PaRSEC: exploiting heterogeneity to enhance scalability. Comput Sci Eng. 2013;15(6):36–45. https://doi.org/10.1109/MCSE.2013.98.
9. Bosilca G, Bouteiller A, Danalis A, Herault T, Lemarinier P, Dongarra J. DAGuE: a generic distributed DAG engine for high performance computing. In: 2011 IEEE international symposium on parallel and distributed processing workshops and PhD forum, pp. 1151–1158; 2011. https://doi.org/10.1109/IPDPS.2011.281. ISSN: 1530-2075.
10. Thoman P, Salzmann P, Cosenza B, Fahringer T. Celerity: high-level C++ for accelerator clusters. In: Euro-Par 2019: parallel processing vol. 11725, pp. 291–303. Springer, Basel; 2019. https://doi.org/10.1007/978-3-030-29400-7_21 . http://link.springer.com/10.1007/978-3-030-29400-7_21. Accessed 24 Sep 2020.
11. The Khronos Group: SYCL specification, version 2020 revision 5. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html. Accessed 05 Dec 2022.
12. Salzmann P, Knorr F, Thoman P, Gschwandtner P, Cosenza B, Fahringer T. An asynchronous dataflow-driven execution model for distributed accelerator computing. In: 2023 IEEE/ACM 23rd international symposium on cluster, cloud and internet computing (CCGrid), pp. 82–93; 2023. IEEE, New York.
13. Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J. OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett. 2011;21(02):173–93.
14. Chamberlain BL, Callahan D, Zima HP. Parallel programmability and the chapel language. Int J High Perform Comput Appl. 2007;21(3):291–312. https://doi.org/10.1177/1094342007078442. SAGE Publications Ltd STM. Accessed 24 Feb 2021.
15. Ebcioglu K, Saraswat V, Sarkar V. X10: programming for hierarchical parallelism and non-uniform data access. In: Proceedings of the international workshop on language runtimes, OOPSLA, Citeseer, vol. 30; 2004.
16. Slaughter E, Lee W, Treichler S, Bauer M, Aiken A Regent: a high-productivity programming language for HPC with logical regions. In: Proceedings of the international conference for high performance computing, networking, storage and analysis, SC '15, pp. 1–12; 2015. https://doi.org/10.1145/2807591.2807629. ISSN: 2167-4337.
17. Falcou J, Sérot J, Chateau T, Lapresté JT. Quaff: efficient C++ design for parallel skeletons. Parallel Comput Algorithm Skelet. 2006;32(7):604–15.
18. Javed N, Loulergue F. Parallel programming and performance predictability with Orléans skeleton library. In: 2011 international conference on high performance computing and simulation, pp. 257–263; 2011.
19. Sato S, Iwasaki H. A skeletal parallel framework with fusion optimizer for GPGPU programming. In: Hu Z, editor. Programm Lang Syst. Berlin: Springer; 2009. p. 79–94.
20. Ernstsson A, Kessler C. Extending smart containers for data locality-aware skeleton programming. Concurr Comput Pract Exp. 2019;31(5):5003.
21. Knorr F, Thoman P, Fahringer T. Declarative data flow in a graph-based distributed memory runtime system. In: International symposium on high-level parallel programming and applications (HLPP 2022); 2022.
22. Thoman P, Wippler M, Hranitzky R, Gschwandtner P, Fahringer T. Multi-GPU room response simulation with hardware raytracing. Concurr Comput Pract Exp. 2022;34(4):6663.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.