Command Horizons: Coalescing Data Dependencies While Maintaining Asynchronicity

Peter Thoman(✉) and Philip Salzmann

Distributed and Parallel Systems Group, University of Innsbruck,
Technikerstraße 21a, Innsbruck, Austria
{peter.thoman,philip.salzmann}@uibk.ac.at

Abstract. In runtime systems for distributed memory parallel computing which automatically manage dependencies and data transfers, a fundamental trade-off exists between the fidelity of dependency tracking and the overhead incurred by its implementation.

Precise tracking of data state allows for effective scheduling, which can leverage opportunities for compute and transfer parallelism. However, it also induces more overhead, and with some data access patterns this overhead can grow with e.g. the number of iterations of an algorithm.

We present the concept of command horizons, which allow coalescing of previous fine-grained tracking information while maintaining an easily configurable scheduling window with full information precision. Furthermore, they enable 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 demonstrates that horizons are effective in keeping the scheduling complexity constant, while their own overhead is negligible – below $10\mu s$ 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

1 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.

P. Diehl et al. (Eds.): WAMTA 2023, LNCS 13861, pp. 13–26, 2023.
https://doi.org/10.1007/978-3-031-32316-4_2

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) [12] is combined with a data parallel programming model such as OpenMP, CUDA or OpenCL. As a consequence of this complexity, development of new software for HPC is typically left to the select few, and research is often 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 with high performance code for distributed memory GPU clusters are higher-level *runtime systems*. These typically introduce a broad API and custom terminology, as well as enabling ecosystems of tooling and derived software projects.

A notable example is StarPU [1], 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 [11,15]. Nevertheless, StarPU's C API is rather low level and requires the explicit handling of data distribution when executing in cluster environments.

Legion [2] 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 [9] 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 [6].

PaRSEC [3] uses a custom graph representation language called JDF to describe the dataflow of an application [4]. 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 [16] was designed to minimally extend the SYCL programming standard [14] 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.

A related category comprises those projects which extend the grammar of existing programming languages, for example the pragma-based OmpSs [7], or introduce entirely new languages altogether, such as Chapel [5], X10 [8] or Regent [13].

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.

While for some data access patterns – e.g. stencil-like computations – this is quite manageable with a relatively simple approach, more unusual patterns 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.

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. Core design goals and features of Horizons include:

- Maintaining asynchronous command generation and execution.
- Allowing for a configurable tradeoff in the level of detail regarding data state available for command generation.
- 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 type of access patterns 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.

2 Background

2.1 The Celerity Runtime System

Celerity is a modern, open C++ framework for distributed GPU computing [16]. Built on the SYCL industry standard [14] 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.

[1] Readers may refer to [10,16] and [14], as well as the Celerity documentation at https://celerity.github.io/docs/getting-started.

Listing 1 A basic matrix operation in Celerity.

```
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, access::one_to_one{}, read_only};
8   accessor out{buf_out, cgh, access::one_to_one{}, write_only};
9   cgh.parallel_for(rg, [=](item<2> itm) {
10    out[itm] = in[itm] * 2.f;
11  });
12 });
```

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.

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 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 user-defined functions, allowing for a high degree of flexibility, with the included *one-to-one*, *slice*, *neighborhood* and *fixed* range mappers serving only to reduce verbosity 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.

2.2 Data State Tracking

From a theoretical point of view (in practice, custom acceleration data structures are employed), 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 operation performed by a program, and are queried whenever a buffer is read, and the performance of these operations is crucial to the overall efficiency of the runtime system.

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*, 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.

2.3 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 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.

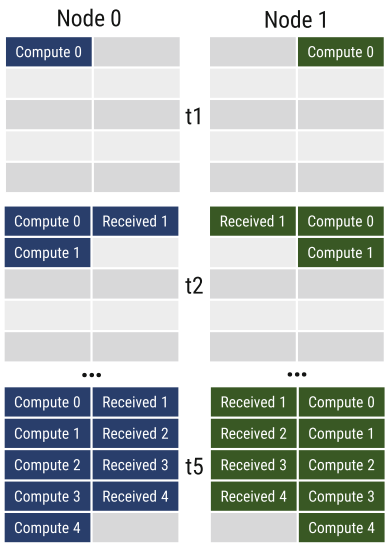

Fig. 1. State tracking with a generative access pattern.

With N GPUs, this means that the tracking data structure will contain $O(N * t)$ 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 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.

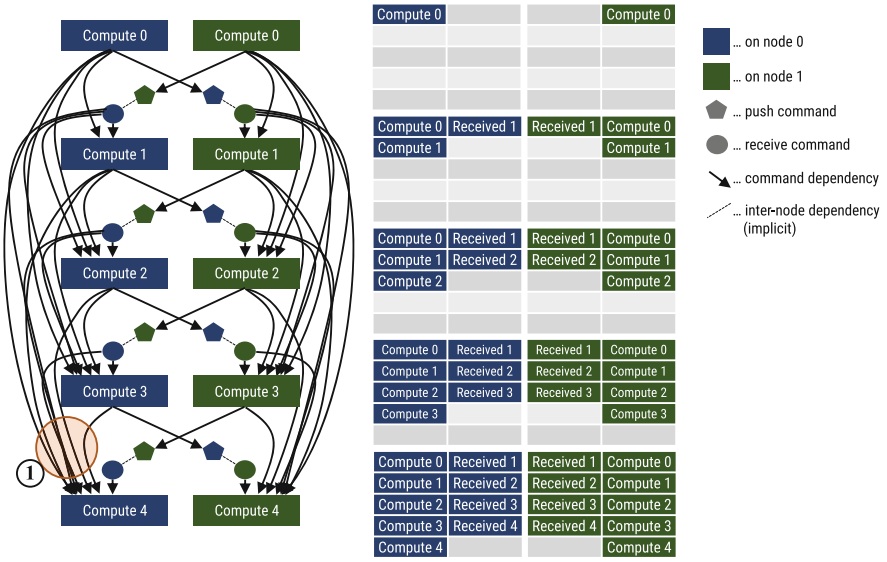

Fig. 2. Command graph and buffer tracking for a generative data pattern.

3 Horizons

Figure 2 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.

Horizon Decision Making. The decision on whether to generate a new Horizon is made during task graph generation. When nodes are inserted into the task graph, they track the current critical path length C from the start of the program. 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 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

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

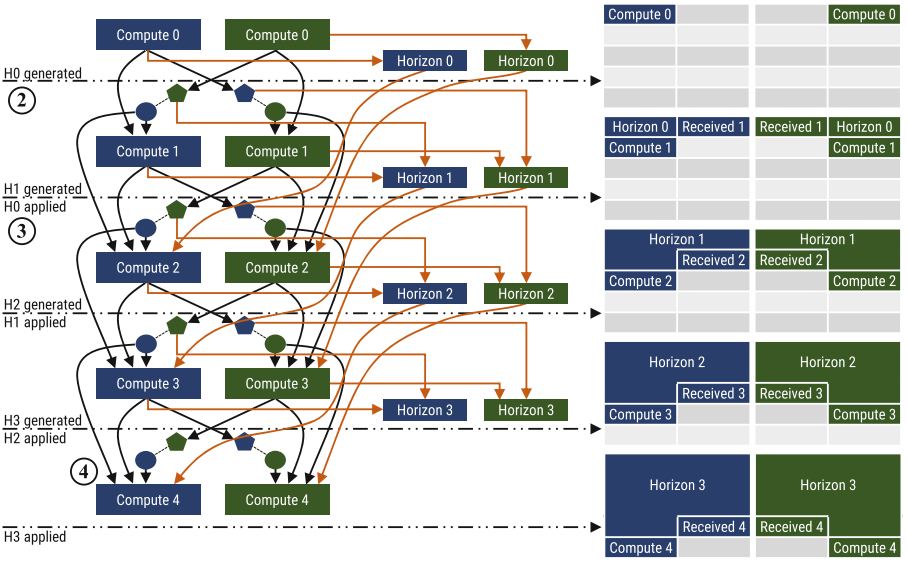

Fig. 3. Command graph and buffer tracking for a generative data pattern with Horizons, using the minimum step size $S = 1$.

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, which is easily tracked throughout the command

generation process and contains all commands for which there currently are no successors. As a consequence, after each Horizon generation, the execution front contains only the horizon command. Figure 3 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. Crucially, Horizons are always applied with a delay of one step, which maintains fine-grained tracking 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. 3, at ③ Horizon 0 is applied, thus replacing Compute 0 in the tracking data structures. As such, in any subsequent command generation steps, dependencies which would have been generated referring to any of these prior commands directly will instead refer to the appropriate Horizon. A comparison between ④ in Fig. 3 and ① in Fig. 2 illustrates how Horizons thus maintain a constant command dependency structure with generative data access patterns.

The Horizon approach as presented has the following advantages: *(i)* it is independent of the specifics of the data access pattern, *(ii)* it maintains a constant maximum on the per-node dependencies which need to be tracked, *(iii)* a window of high-fidelity dependency information is maintained, and the size of this window can be adjusted by setting the step size S, *(iv)* horizon generation is 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 command, *(v)* horizon application is highly efficient, as due to the numbering scheme of commands a simple integer check suffices (no graph traversal is required), and *(vi)* no additional communication is required.

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 present microbenchmarks of simple generative data patterns to precisely track the impact of Horizon step sizes on command generation times.

Secondly, we demonstrate that Horizons have negligible overhead at both small and large scales, and can even be beneficial for programs without generative access patterns, using *dry-run* benchmarks. In dry-run mode, the Celerity runtime system performs all the scheduling and command generation work of a

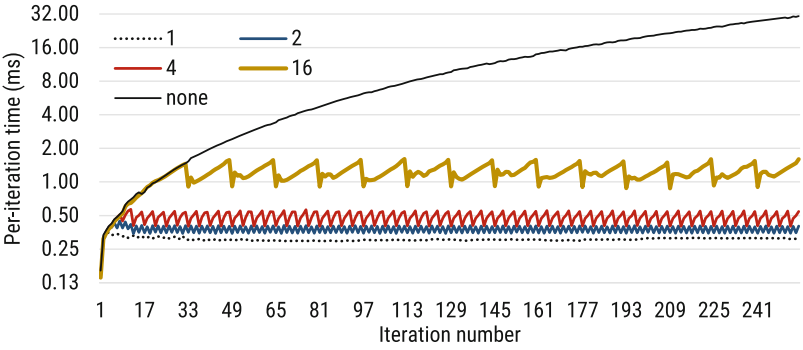

Fig. 4. Per-iteration time for 2D generative access microbenchmark; each line shows a different horizon step setting S (or no Horizons), as indicated in the legend.

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.

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 real-world application benchmarks were performed on the Marconi-100 supercomputer[2] at CINECA in Bologna, Italy, with 5 runs each.

Microbenchmarks. Figure 4 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.

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 Section 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 5 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,

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

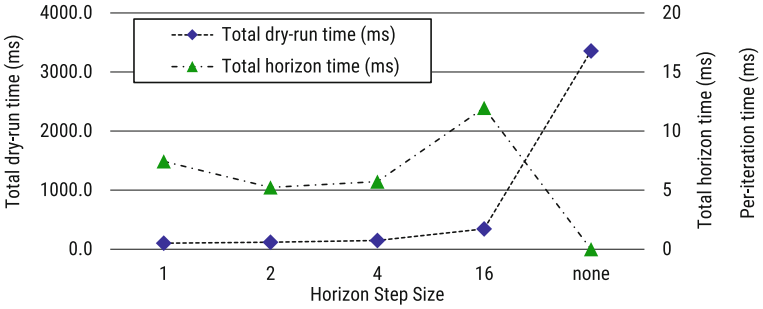

Fig. 5. Total times for 2D generative access microbenchmark.

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.

Overhead. 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. Figure 6 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. 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

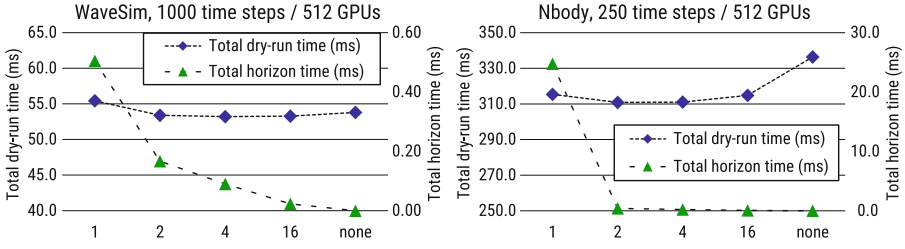

Fig. 6. Horizon impact and overhead for two non-generative applications. X-axis shows Horizon use/step size.

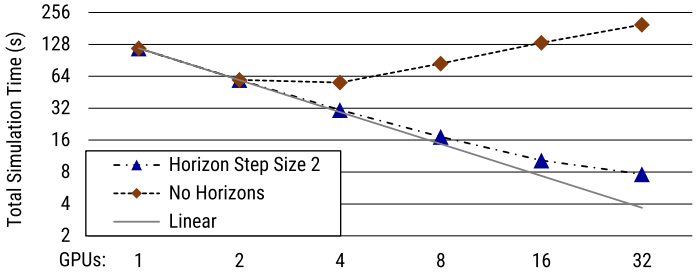

Fig. 7. Horizon impact on RSim application.

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. 7 shows the result of a strong scaling experiment with the Celerity version of RSim [17], 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 overhead in the runtime system. Instead, it can be attributed to the fact that this is a strong scaling experiment with per-timestep communication requirements.

5 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, 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 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.

Acknowledgements. This project has received funding from the European High Performance Computing Joint Undertaking, grant agreement No 956137.

References

1. 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)
2. 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 (2012)
3. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., Dongarra, J.J.: PaRSEC: exploiting heterogeneity to enhance scalability. Comput. Sci. Eng. **15**(6), 36–45 (2013)
4. 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). ISSN 1530-2075
5. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007)
6. 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)
7. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**(02), 173–193 (2011)
8. 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, vol. 30. Citeseer (2004)
9. 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)
10. 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)
11. Kumar, S.: Scheduling of dense linear algebra kernels on heterogeneous resources. Ph.D. thesis, Université de Bordeaux (2017)
12. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.1 (2015)

13. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: a high-productivity programming language for HPC with logical regions. In: SC 2015: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12 (2015). ISSN 2167-4337
14. The Khronos Group: SYCL Specification, Version 2020 Revision 5 (2022)
15. Thibault, S.: On runtime systems for task-based programming on heterogeneous platforms. Thesis, Université de Bordeaux (2018)
16. Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: high-level C++ for accelerator clusters. In: Yahyapour, R. (ed.) Euro-Par 2019. LNCS, vol. 11725, pp. 291–303. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29400-7_21
17. Thoman, P., Wippler, M., Hranitzky, R., Gschwandtner, P., Fahringer, T.: Multi-GPU room response simulation with hardware raytracing. Concurr. Comput. Pract. Exp. **34**(4), e6663 (2022)