Automatic Discovery of Collective Communication Patterns in Parallelized Task Graphs

Fabian Knorr[1*], Philip Salzmann[1], Peter Thoman[1*], Thomas Fahringer[1]

[1]University of Innsbruck, Austria.

*Corresponding author(s). E-mail(s): fabian.knorr@uibk.ac.at; peter.thoman@uibk.ac.at;
Contributing authors: philip.salzmann@uibk.ac.at; thomas.fahringer@uibk.ac.at;

Abstract

Collective communication APIs equip MPI vendors with the necessary context to optimize cluster-wide operations on the basis of theoretical complexity models and characteristics of the involved interconnects.
Modern HPC runtime systems with a programmability focus can perform dependency analysis to eliminate the need for manual communication entirely. Profiting from optimized collective routines in this context often requires global analysis of the implicit point-to-point communication pattern or tight constrains on the data access patterns allowed inside kernels.
The Celerity API provides a high degree of freedom for both runtime implementors and application developers by tieing transparent work assignment to data access patterns through user-defined range-mapper functions. Canonically, data dependencies are resolved through an intra-node coherence model and inter-node point-to-point communication.
This paper presents *Collective Pattern Discovery* (CPD), a fully distributed, coordination-free method for detecting collective communication patterns on parallelized task graphs. Through extensive scheduling and communication microbenchmarks as well as a strong scaling experiment on a compute-intensive application, we demonstrate that CPD can achieve substantial performance gains in the Celerity model.

Keywords: Task Graph, Scheduling, MPI, Collective Communication

1 Introduction

As we enter the Exascale era with ever increasing parallelism and heterogeneity in clusters, a growing number of HPC applications become bound primarily by memory and communication bottlenecks. Efficiently managing communication between memory hierarchies is now of the utmost importance for scaling any application beyond a small number of compute nodes.

With traditional HPC software stacks – i.e. MPI+X – these hardware developments necessitate an increasing level of expertise in parallelization and distributed software optimization on part of the application programmer. However, as the actual domain of the computations performed on HPC systems is generally some other physical science, such expertise is only available to large projects consortia, or by leveraging existing domain-specific software packages.

This state of the art hampers the development of new algorithms and science, as there is a clear trade-off: experiment with new algorithmic and scientific approaches while restricted to smaller-scale or less efficient computation; or accept the limits of existing software packages, but scale more easily to larger systems and problem sizes.

One approach towards bridging this gap between a focus on allowing relatively straightforward implementation of domain science on the one hand and the complexities of large heterogeneous distributed memory clusters on the other hand are *HPC runtime systems* which seek to automate aspects like data distribution. While systems like Celerity [14] can greatly reduce the burden on the application programmer, meeting the high degree of freedom necessary to target the vast cosmos of data access patterns found in scientific computing will require a communication model built around point-to-point primitives in the general case.

For communication patterns involving a large number of cluster nodes however, collective communication primitives as found in MPI can outperform point-to-point cascades in network latency and throughput while also reducing tracking overhead in the runtime. In this paper, we suggest that the conflict in requirements between API expressiveness, programmability and communication efficiency can best overcome by automated pattern detection and optimization on an existing point-to-point model.

To substantiate this claim, we present *Collective Pattern Discovery* for the Celerity model, a method which automates detection of data access patterns that map to collective communication steps and inserts eager collective communication steps where possible. Our approach is deterministic and fully distributed without coordination between participating nodes and exhibits low overhead. It neither requires training, observation of previous communication nor guidance from the application developer.

1.1 MPI Collectives

The MPI Standard [10] defines five categories of non-mutating collective operations that can replace equivalent, hand-rolled point-to-point communication cascades for improved latency and throughput.

These collectives are either symmetric or revolve around one *root* node; and transmitted data is either *personalized* (nodes receive disjoint buffer sub-ranges) or *non-personalized* (every node receives the full buffer range).

collective	operation	MPI function
broadcast	non-personalized one-to-all	`MPI_Bcast`, `MPI_Ibcast`
scatter	personalized one-to-all	`MPI_Scatter[v]`, `MPI_Iscatter[v]`
gather	all-to-one	`MPI_Gather[v]`, `MPI_Igather[v]`
all-gather	non-personalized all-to-all	`MPI_Allgather[v]`, `MPI_Iallgather[v]`
all-to-all	personalized all-to-all	`MPI_Alltoall[vw]`, `MPI_Ialltoall[vw]`

Table 1 Non-mutating collective operations provided by MPI

The significance of efficient collectives for MPI application performance becomes apparent in the extensive library of research on optimizing these operations in popular implementations [9, 13]. Accurate theoretical models allow latency- and throughput-optimized implementations to select optimal communication patterns depending on cluster topology [5] and problem size [11].

1.2 Celerity

Celerity is a high-level C++ runtime system for accelerator clusters, focusing on programmability in the complex environment of distributed-memory accelerator computing [14]. It provides developers with a dataflow-based parallelism model reminiscent of single-GPU programming while transparently distributing computation across compute nodes. In order to ease adoption and leverage existing standards as far as possible, its programming interface is closely related to the established SYCL API, with minimal extensions required for operation on distributed memory [6].

Celerity is built around fully distributed and asynchronous task and command graph generation, which has previously been shown to scale up to 128 GPUs for compute-intensive algorithms [12]. However, prior to this work, Celerity's implicit communication model was exclusively implemented through asynchronous MPI point-to-point operations.

1.3 Case Study: Direct N-Body Simulation

To familiarize the reader with the Celerity model and demonstrate the performance impact of collective communication later in this paper, we showcase the implementation of a direct gravitational N-body simulation as defined by

$$v_{i,t+1} := v_{i,t} + \sum_{j \neq i} \frac{G m_j (p_j - p_i)}{\|p_j - p_i\|^3} \Delta t, \qquad p_{i,t+1} := p_{i,t} + v_{i,t+1} \Delta t, \tag{1}$$

where p are 3-dimensional body positions, v their velocities, m their masses, G the gravitational constant and t are time steps of length Δt.

The abbreviated Celerity program in listing 1 represents this system in two virtualized buffers P and V. In a loop, it submits two kernels per time step: `time_step` computes $v_{i,t+1}$ from $v_{i,t}$ by integrating over the entirety of P for each work item i; then `update_p` updates $p_{i,t+1}$ in-place from $p_{i,t}$ and $v_{i,t+1}$.

```
 1  using namespace celerity;
 2
 3  buffer<double3, 1> P(N);
 4  buffer<double3, 1> V(N);
 5  const double M = 1.0 /* kg */;
 6
 7  distr_queue q;
 8  for (double t = 0.0f; t < T; t += dt) {
 9      q.submit([&](handler &cgh) {
10          accessor p(P, cgh, access::all(), read_only);
11          accessor v(V, cgh, access::one_to_one(), read_write);
12          cgh.parallel_for<class time_step>(range<1>(N), [=](item<1> i) {
13              double F = 0.0;
14              for (size_t j = 0; j < N; ++j) { F += gravity(p[i], p[j]); }
15              v[j] += M * F * dt;
16          });
17      });
18      q.submit([&](handler &cgh) {
19          accessor v(V, cgh, access::one_to_one(), read_only);
20          accessor p(P, cgh, access::one_to_one(), read_write);
21          cgh.parallel_for<class update_p>(range<1>(N),
22              [=](item<1> i) { p[i] += v[i] * dt; });
23      });
24  }
```

Listing 1 Simplified implementation of direct N-body simulation in Celerity.

Each kernel is submitted as part of an asynchronous *command group*, which ties the kernel function to an *execution geometry* (lines 12 and 21) and any number of *buffer accessors* (lines 10–11 and 19–20).

The execution geometry describes parallelization through a dimensionality (here 1), an execution range (here N), a work item offset (implicitly 0 here) and a work-group size (implicit and implementation-defined here).

Through lambda captures, accessors inject device-buffer pointers into the kernel while providing the scheduler with metadata in the form of an *access mode* (here `read_only`, `read_write`) and a *range mapper* (here `all` and `one_to_one`).

1.4 Range Mappers

Range mappers are an essential concept of the Celerity model, mapping sub-ranges of the execution range to sub-ranges of the buffer in an accessor. This enables the discovery of data requirements after arbitrary work assignment.

Given an execution range E, a **range mapper** $r : \mathcal{P}(E) \to \mathcal{P}(E)$ is any pure function that forms a homomorphism over the union of execution sub-ranges:

$$r(E_1 \cup E_2) \;=\; r(E_1) \cup r(E_2) \qquad \forall E_1, E_2 \subset E \tag{2}$$

Any range mapper r that is used in a writing access is further required to be **non-overlapping** to allow tracking of the unique producer for any buffer item:

$$E_1 \cap E_2 = \emptyset \;\Rightarrow\; r(E_1) \cap r(E_2) \;=\; \emptyset \qquad \forall E_1, E_2 \subset E \tag{3}$$

Celerity ships a selection of built-in range mapper functions. Relevant to the following discussion are `one_to_one` (the identity function, requires equal kernel and buffer dimensions), `all` (constant, accessing the entire buffer range) and `transposed` (an isomorphic shuffling of dimensions). Out of these, `one_to_one` and `transposed` exhibit the non-overlapping property, while `all` does not.

1.5 Graph-Based Scheduling

Celerity's parallel schedule is derived from the flow of command group submissions in two steps: The high level *task graph*, constructed synchronously on all participating nodes, describes execution on a cluster-wide level. From this task graph, each rank generates an individual *command graph* that models the kernel launches and communication steps performed within the node.

Work is assigned to accelerators by splitting the global execution range into near-equally-sized sub-ranges while observing any constraints imposed by the execution geometry. As one Celerity process usually drives all accelerators of a cluster node, scheduling will produce multiple execution sub-ranges locally. The graph generation process itself does not involve communication.

State-of-the-art Celerity resolves data-flow dependencies between nodes to point-to-point transfers. In this approach, each node tracks the buffer sub-ranges produced by kernels within its address space through a combination of R-trees [3], from which inbound communication sub-ranges (*await-push commands*) and outbound communication targets (*push commands*) are derived. Lowered to MPI point-to-point primitives, these commands satisfy any data access pattern that can be described by the range-mapper model. We refer the reader to [12] for more details about how Celerity implements its graph-based scheduling and dependency tracking.

Figure 1 shows an excerpt of the task and command graphs resulting from Listing 1. Here, as Celerity decides to assign the same execution sub-ranges to the same nodes across kernels, only the `all`-read requirement of `time_step` necessitate communication. The corresponding command graph contains $M - 1$ push commands and one await-push command on every node out of M.

1.6 Multi-Device Execution and Memory Coherence

Each Celerity process generates and streams its command graph to its *executor* thread, which drives all accelerators addressable by the node. The executor dynamically establishes memory coherence between host and device memories by tracking buffer writes and replications in separate R-trees, issuing memory transfers before passing kernels to the SYCL backend.

While this lazy-update approach effectively balances irregular workloads, missing context about the higher-level operation each sequence of commands is part of can lead to sup-optimal execution patterns at times. This holds especially true for the all-gather pattern found in our N-body simulation, for which the executor will issue a coherence update for every incoming transfer ($M-1$ for M nodes) instead of coalescing them into a single transfer.

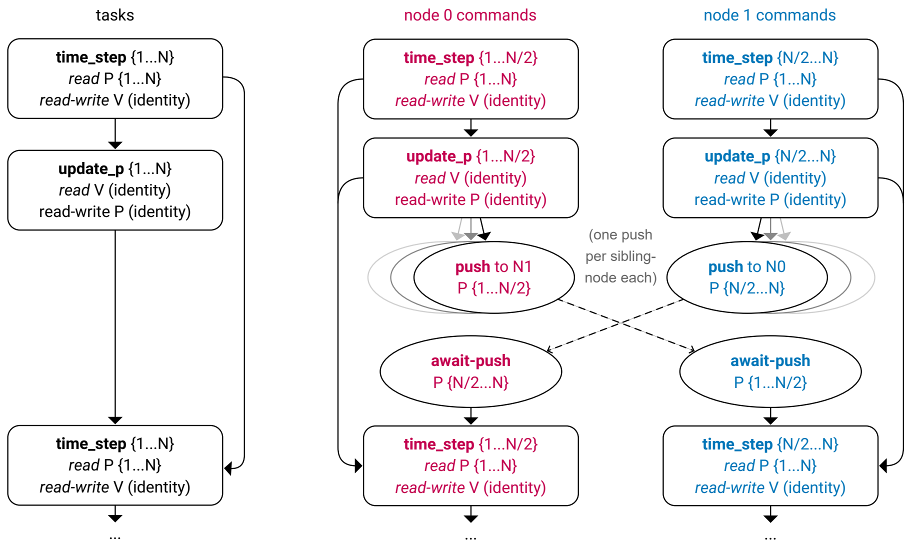

Fig. 1 Task graph (left) and command graphs (right) of a point-to-point communication schedule for direct N-body simulation from listing 1 on $M = 2$ nodes. We show tasks up to the second `time_step` kernel submission and hint at the additional push commands (grey) that would be required for a command graph on $M > 2$ nodes.

2 Related Work

Uncovering and exploiting opportunities for collective communication in user programs has been examined from different angles in recent literature.

These approaches can be broadly categorized into *bottom-up* schemes discovering collective patterns through centralized analysis of existing point-to-point programs, and *top-down* methods which derive these patterns from high-level cluster-wide representations and can frequently be coordination-free.

Knüpfer et al. [7] perform post-hoc, bottom-up analysis of application traces with MPI point-to-point communication, hinting potential sites for collective communication to the application developer help manual refactoring.

Hoefler et al. [4] use compiler transformations to replace point-to-point operations with library function calls that build a communication DAG at runtime. In a centralized bottom-up analysis pass, this approach reliably detects all regular (i.e. non-`MPI_*[vw]`) collective patterns. By re-using optimized schedules across program iterations, the authors are able to amortize the overhead of their optimization.

libWater [2] is an OpenCL-based runtime that dynamically offloads work from a designated *root node* to devices attached to other MPI processes. In a bottom-up scheme, it detects gather, scatter and broadcast patterns among the point-to-point commands generated as part of data redistribution pass and inserts MPI collective operations accordingly.

Denis et al. [1] extend the PaRSEC runtime to opportunistically discover broadcast patterns bottom-up during task graph build time. To avoid the synchronization penalty from orchestrating a call to `MPI_Bcast` from otherwise independent schedulers, the sending node initiates a binomial-tree broadcast through point-to-point messages which are forwarded by intermediate nodes.

In a top-down approach, the cluster backend of SkePU [8] leverages MPI collectives to exchange data between operations where applicable. The rigid skeleton model significantly eases the modelling of global data movement and computational patterns when compared to Celerity, which must allow near-arbitrary non-overlapping writes based on range mappers.

Collective Pattern Discovery as presented in the remainder of this document falls into the *top-down* category, analyzing data requirements of a parallelized task-graph through a distributed and coordination-free algorithm.

3 Collective Pattern Discovery

Collective Pattern Discovery (CPD) is a novel, deterministic, synchronization- and coordination-free method for detecting instances of all five collective data exchange patterns found in section 1.1. In two phases, CPD transforms both the replicated task graph and the per-node individual command graph to identify dataflow edges that can profit from eager collective communication.

By guaranteeing that all nodes generate collective commands in identical order regardless of individual work assignment, it satisfies the MPI requirement that all ranks in a communicator participate in every collective operation.

3.1 Forward Task Generation

The first step in Collective Pattern Discovery (CPD) locates *potential* edges in task graph, where an eager collective operation may preempt later point-to-point buffer updates that would be inserted lazily on command generation.

Although the task graph is oblivious to communication and fully independent of the underlying cluster configuration (including the number of participating nodes), it must still keep track of collectives to guarantee that all nodes participate in the same operations. This also avoids inadvertently exchanging buffer ranges multiple times, as the task graph will reveal whether a dataflow dependency terminates at the original data producer or whether there are intermediate tasks for which the data has potentially been exchanged before.

CPD thus inserts a **forward task** whenever a read-requirement of task c (the *consumer*) would introduce the first task-level dependency on the original writer task p (the *producer*) for the accessed region (algorithm 1).

To maximize the number of forward tasks that result in non-trivial collective communication after work assignment, CPD ignores any task edges it deems to be *communication-free* by assuming that tasks which share an execution geometry will receive identical work assignment in the scheduler.

```
$R_{t,B}$ := read-set of task $t$ on buffer $B$
$W_{t,B}$ := write-set of task $t$ on buffer $B$
$W^*_{t,B}$ := subset of $W_{t,B}$ not overwritten by any subsequent task
$A_t(F) := \{r \mid r \text{ is a range mapper in } t \wedge r(E_t) \cap F \neq \emptyset\}$

procedure IsCommunicationFree($p$, $c$, $F$)
    {Assume tasks of identical geometry will have identical work assignment}
    if $F = \emptyset$ then return True
    else if either $p$ or $c$ is a forward task then return False
    else if $p$ and $c$ have different execution geometry then return False
    else if $A_p(F, \text{write}) = A_c(F, \text{read})$ then return True
    else return False

procedure GenerateForwardTasks($t$)
    for each buffer $B$ with $R_{t,B} \neq \emptyset$ do
        for each previous task $p \neq t$ with $W^*_{p,B} \cap R_{t,B} \neq \emptyset$ do
            $F \leftarrow W^*_{t,B} \cap R_{t,B}$
            for each task $c \notin \{t, p\}$ dependent on $p$ do
                if not IsCommunicationFree($p$, $c$, $W^*_{p,B} \cap R_{c,B}$) then
                    $F \leftarrow F \setminus R_{c,B}$
            if not IsCommunicationFree($p$, $t$, $F$) then
                insert forward-task $f$ with dependencies $p \rightarrow f \rightarrow t$
                $R_{f,B} \leftarrow W_{f,B} \leftarrow F$
```

Algorithm 1 Forward-task generation for a command-group task t

3.2 Eager Collective Command Generation

In the Celerity model, work assignment and thus the number of nodes participating in a task is a function of the execution geometry and the number of nodes and accelerators in the system. This ensures that command graph generation, while distributed, agrees on a single global schedule. Our implementation guarantees this through fully-static scheduling. Dynamic scheduling methods remain compatible with CPD, provided that their schedules are deterministic and reproducible around forward tasks.

After work assignment, the second step of CPD materializes forwards between producer and consumer tasks as **collective commands** if they match one of the patterns

collective	producer nodes	consumer nodes	producer range mappers	consumer range mappers
gather	M	1	non-overlapping	any
all-gather	M	M	non-overlapping	constant
broadcast	1	M	non-overlapping	constant
scatter	1	M	non-overlapping	non-overlapping
all-to-all	M	M	— non-trivial transposition —	

Table 2 Discovery patterns for collective operations on $M > 1$ nodes

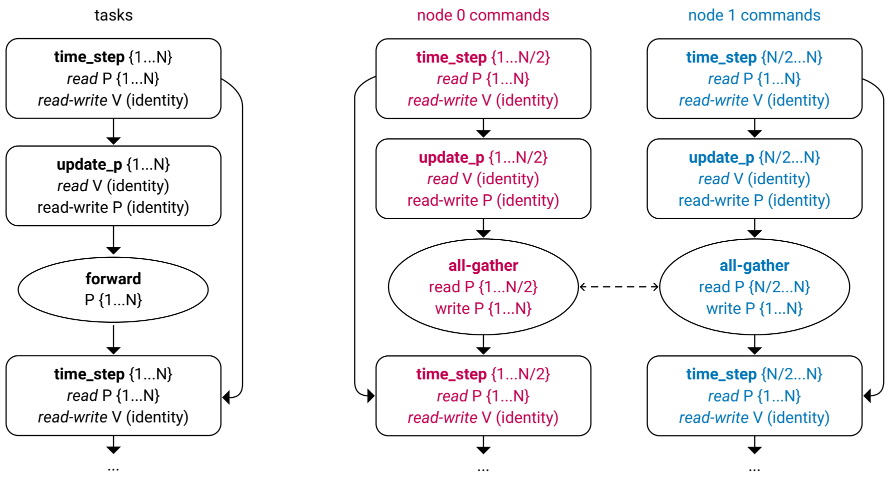

Fig. 2 Task graph (left) and command graphs (right) of a direct N-body simulation with Collective Pattern Discovery. The forward task on P materializes as a all-gather operation, replacing the push-await cascade seen in figure 1.

found in table 2. Any non-matching forward task is dropped, and communication will proceed through the generic point-to-point algorithm.

The pattern matching approach is independent of the exact buffer regions each node accesses, rather, the collective operation is determined in constant time from the number producer and consumer commands and range-mapper metadata. The non-overlapping property of producer (writer) range mappers is assumed to hold by definition (see section 1.4). Our implementation detects *constant* and *non-overlapping* consumer range mappers as well as *transpositions* through meta-programming on the range-mapper functions.

The common *gather*, *all-gather*, *scatter* and *broadcast* patterns are identified by analyzing read- and write range mappers in separation.

The *all-to-all* communication pattern is identified through a consumer access that forms a *non-trivial transposition* of the corresponding producer, i.e. one that is not communication-free after work assignment:

1. Producer task p has exactly one write range mapper w; consumer task c exactly one read range mapper r participating in the forwarded region F
2. It holds that $w(E_p) = r(E_c) = F$
3. For any dimension d, all mappings of nodes i to produced buffer ranges $w_d(E_{p,i})$ and $r_d(E_{c,i})$ are either constant or the identity function
4. There exists d such that $w_d(E_{p,i})$ is constant while $r_d(E_{c,i})$ is the identity
5. There exists d such that $w_d(E_{p,i})$ is the identity while $r_d(E_{c,i})$ is constant.

Figure 2 visualizes the effects of Collective Pattern Discovery on command-graph generation for the N-body simulation in listing 1.

Collective Pattern Discovery first analyzes the data flow between the initial `time_step` and `update_p` tasks. Since producer and consumer both access buffer V through the same identity range mapper and the tasks have identical execution geometry, the edge is considered to be communication-free and no forward task is generated.

The read of $P\{1 \ldots N\}$ by the second `time_step` kernel however applies a different range mapper than the producer `update_p`. As the buffer has not been read by any task since, CPD inserts a forward task on $P\{1 \ldots N\}$.

After work assignment, the producer–consumer relationship around P connects an M-node non-overlapping producer to a M-node constant consumer, matching the all-gather pattern of table 2. Celerity thus inserts an *all-gather* command on each node, which becomes the new writer of $P\{1 \ldots N\}$.

Since all data requirements of the second `time_step` are now fulfilled, no additional push-await pairs are generated during dependency analysis.

3.3 Collective Command Execution

Celerity lowers all collective commands to their non-blocking MPI counterparts (e.g. `MPI_Iallgatherv`). As required by the standard, these operations are initiated in-order, but can overlap for the remainder of their execution time.

Since each process potentially drives multiple accelerators, the runtime compiles larger device-to-device collectives from the host-to-host MPI operations by issuing local memory transfers before and after the MPI invocation.

Knowledge about the cluster-wide collective operation provides optimization potential beyond the lazy coherence update mechanism (section 1.6) employed for point-to-point transfers: Celerity will issue a parallel *device broadcast* to update all accelerator memories after completing an MPI collective operation with receiver-broadcast semantics (*broadcast* and *all-gather* patterns).

4 Evaluation

To assess the performance characteristics of Collective Pattern Discovery in isolation, we implement a set of synthetic benchmarking applications that require communication between device memories (table 3).

benchmark	step	kernel	reads	writes
all-gather		N	$B \leftarrow \{1 \ldots N\}$	$B' \leftarrow$ identity
gather-scatter	1.	1	$B \leftarrow \{1 \ldots N\}$	$B \leftarrow \{1 \ldots N\}$
	2.	N	$B \leftarrow$ identity	$B' \leftarrow$ identity
gather-bcast	1.	1	$B \leftarrow \{1 \ldots N\}$	$B \leftarrow \{1 \ldots N\}$
	2.	N	$B \leftarrow \{1 \ldots N\}$	$B' \leftarrow$ identity
all-to-all		$N \times N$	$B \leftarrow$ transpose$(0,1)$	$B' \leftarrow$ identity
stencil (control)		$N \times N$	$B \leftarrow$ neighborhood$(1,1)$	$B' \leftarrow$ identity

Table 3 Access patterns of the synthetic benchmarks examined in this section. Executing the steps of each program in a loop generates detectable collective communication patterns (except *stencil*). After each iteration, buffers B and B' are swapped.

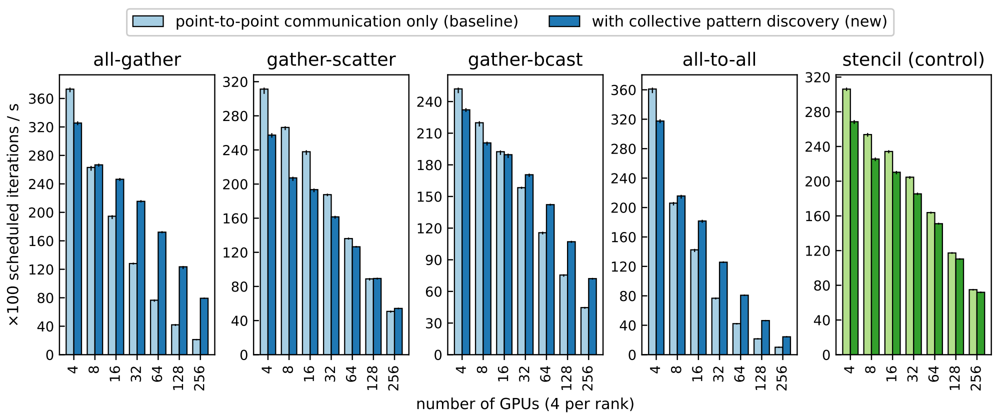

Fig. 3 Scheduler throughput for each program listed in table 3 (higher is better). Reported is median of 100 benchmarks together with minima and maxima.

Where applicable, one-to-all communication is paired with an all-to-one operation to maintain meaningful dataflow throughout the programs. As control we study the overhead of CPD on a stencil-like program with a neighborhood exchange pattern that does not benefit from collective communication.

All benchmarks in this section were run on the Marconi-100 supercomputer in Bologna, Italy, rank 26 of the TOP500 list as of June 2023[1]. It is a cluster of 980 IBM Power AC922 nodes with four Nvidia Volta V100 GPUs each, intra-node NVLink 2.0, and dual Infiniband EDR system interconnect.

Celerity was built using Clang 12.0.1 and OpenSYCL 0.9.2[2] with `-O3` optimization, linking against CUDA 11.7 and IBM Spectrum MPI 10.4.0. All binaries were executed with mimalloc 2.0.9[3] replacing the system allocator.

4.1 Scheduling Microbenchmarks

Celerity generates task- and command graphs concurrently with kernel execution and data transfers. Scheduling latency can thus usually be hidden after startup, but applications with very short-running device kernels may become throughput-limited.

By isolating the scheduling process, we can analyze scheduler throughput as a function of node count. Each node must compute the work assignment of every other node in the system to detect potential non-collective data requirements. The number of communication commands tracked however remains constant with CPD while increasing linearly with point-to-point communication.

Figure 3 demonstrates that all patterns except *gather-scatter* greatly profit from CPD's reduction in tracking complexity, with *all-gather* achieving a more than 3× throughput increase for 256 nodes. For small node counts, the constant-time overhead of forward-task generation yields a visible drop in scheduler performance, both for

[1] https://www.top500.org/lists/top500/list/2023/06/
[2] https://github.com/OpenSYCL/OpenSYCL/releases/tag/v0.9.2
[3] https://github.com/microsoft/mimalloc/releases/tag/v2.0.9

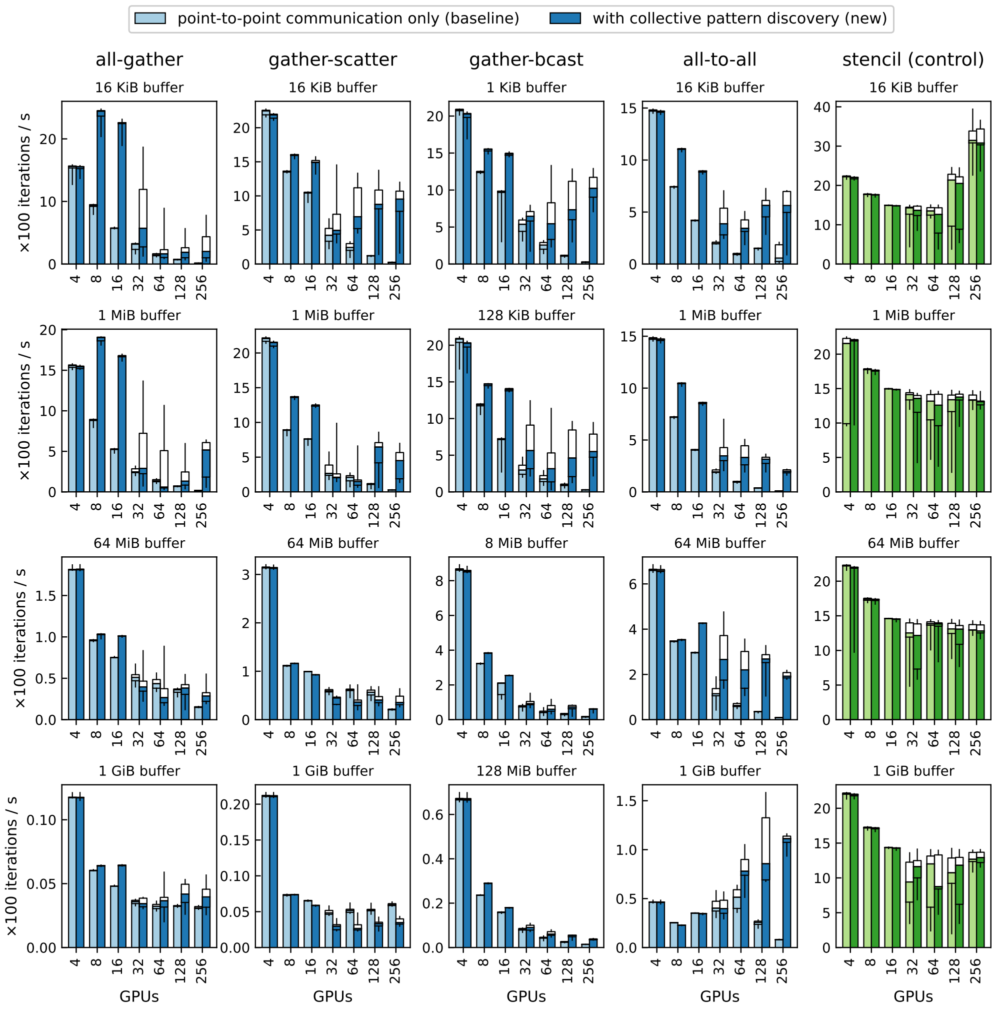

Fig. 4 Throughput of communication-only system benchmarks from table 3 with kernel execution disabled (higher is better). Shown is a mixed bar-box plot containing the median, center quartiles and minima / maxima over 20 runs on varying node configurations. Each measurement is the mean over 20 iterations.

collective and non-collective patterns. As we will show in section 4.2, this reduction in throughput is negligible for large-scale runs.

4.2 Communication-Only System Benchmarks

As Celerity is structured around accelerator computation, we benchmark device-to-device transfer performance specifically by executing the synthetic benchmarks from table 3 with and without CPD while disabling kernel execution.

Figure 4 visualizes the communication throughput achieved as benchmark iterations per second. All collective patterns profit massively from reduced overheads on

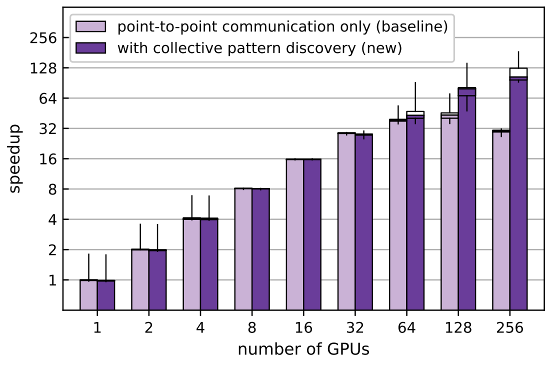

Fig. 5 Strong-scaling speedup of 20 time steps of the direct N-body simulation for $N = 1{,}048{,}576$ in double precision. We report the median, center quartiles and minima / maxima over 20 benchmark runs allocated to varying node configurations by the workload manager.

small buffer sizes, and all except *gather-scatter* can consistently take advantage of reduced bandwidth requirements on larger-sized buffers.

For large node counts, we can observe a high variance in the performance of MPI collective communication, which is caused by process scheduling differences on part of the SLURM workload manager.

The non-collective stencil program shows no difference in communication times between enabling or disabling CPD, demonstrating that the increase in scheduler latency seen in figure 3 can be fully hidden.

4.3 Strong Scaling Experiment: Direct N-Body Simulation

To evaluate the efficacy of CPD on a full application, we implement and optimize the direct N-body simulation from section 1.3 as a Celerity application. Compared to the simplified listing 1, we use an array-of-struct (AOS) to struct-of-array (SOA) transformation on P and V, increase parallelism in `time_step` by writing one item in V per 32 threads and reduce the required global-memory bandwidth in the same kernel by shared-memory tiling the read of V.

We choose a strong-scaling experiment specifically to showcase the effects of transitioning from a compute-bound to a communication-bound problem as the node count increases. Figure 5 shows the speedup attained from a varying number of GPUs participating in the simulation of $N = 1{,}048{,}576$ bodies.

Up to 64 GPUs (16 nodes), both point-to-point and collective communication scale equally. Increasing beyond 128 GPUs yields no additional speedup for the point-to-point configuration, but does so significantly when Collective Pattern Discovery is enabled.

Profiling reveals that scalability in this case is limited primarily by latency of small host-to-device copies for every incoming message, which CPD can effectively reduce through the use of a *device broadcast* (section 3.3).

5 Conclusion

This work introduces Collective Pattern Discovery (CPD), a novel, deterministic, distributed and coordination-free method for reliably identifying opportunities for collective communication in the parallelized task graphs of the Celerity model.

In a two-stage approach, CPD identifies task graph edges suitable for eager communication in the form of *forward tasks* and matches the concrete data exchange pattern after work assignment to generate per-node *collective commands*. This transforms a large class of distributed-memory interactions into collective operations while reliably avoiding duplicated communication.

Through synthetic scheduling and communication benchmarks, we demonstrated how CPD reduces tracking overhead of large runs in the runtime system by replacing a linear number of point-to-point communication pairs with singular collective operations. On large transfers, this transformation allow us to profit from decades of research on MPI collective optimization.

In a strong-scaling experiment, we were able to prove sizable gains in scalability over the point-to-point model, effectively scaling a direct N-body simulation implemented in Celerity to 256 GPUs for the first time.

5.1 Limitations and Future Work

While demonstrably highly efficient in common settings, the graph transformations performed by Collective Pattern Discovery (CPD) cannot claim algorithmic optimality in the general case. For example, the eager generation of forward tasks masks the original producer task of the forwarded buffer sub-region: if the forward is not materialized, or later tasks would benefit from a superset of the generated collective (e.g. a logical *all-gather* access following a simple *gather*), an opportunity for collective communication will be missed. Future work could be able to improve CPD in these situations through a lookahead scheme analyzing longer sequences of tasks at once.

Applicability to other Frameworks

As evident from the technical descriptions in this paper, Collective Pattern Discovery is specialized for the execution model of Celerity. It assumes parallelized task graphs that are user-annotated with *range mappers* to express fine-grained data dependencies.

Other systems that wish to implement CPD will need their own method to statically discover eligible read- and write operations in the distributed program, equivalent to table 2. This task is easiest for an API that is explicit about data access patterns, as has already been demonstrated by the successful incorporation of MPI collective operations in skeleton libraries [8].

Acknowledgements

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

References

[1] Denis, A., Jeannot, E., Swartvagher, P., Thibault, S.: Using dynamic broadcasts to improve task-based runtime performances. In: Euro-Par 2020, Warsaw, Poland, August 24–28, 2020, Proceedings 26. pp. 443–457. Springer (2020)

[2] Grasso, I., Pellegrini, S., Cosenza, B., Fahringer, T.: libWater: Heterogeneous distributed computing made easy. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. p. 161–172. ICS '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2464996.2465008

[3] Guttman, A.: R-trees: a dynamic index structure for spatial searching. ACM SIGMOD Record **14**(2), 47–57 (Jun 1984). https://doi.org/10.1145/971697.602266

[4] Hoefler, T., Schneider, T.: Runtime detection and optimization of collective communication patterns. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. p. 263–272. PACT '12, ACM/doi, New York, NY, USA (2012). https://doi.org/10.1145/2370816.2370856

[5] Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's collective communication operations for clustered wide area systems. In: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 131–140. PPoPP '99, ACM, New York, NY, USA (1999). https://doi.org/10.1145/301104.301116

[6] Knorr, F., Thoman, P., Fahringer, T.: Declarative data flow in a graph-based distributed memory runtime system. International Journal of Parallel Programming pp. 1–22 (2022)

[7] Knüpfer, A., Kranzlmüller, D., Nagel, W.E.: Detection of collective MPI operation patterns. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11. pp. 259–267. Springer (2004)

[8] Majeed, M., Dastgeer, U., Kessler, C.: Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). p. 468. Citeseer (2013)

[9] Mamidala, A.R., Kumar, R., De, D., Panda, D.K.: MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In: 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID). pp. 130–137. IEEE (2008)

[10] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.0, https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[11] Pjesivac-Grbovic, J., Angskun, T., Bosilca, G., Fagg, G.E., Gabriel, E., Dongarra, J.J.: Performance analysis of MPI collective operations. In: 19th IEEE International Parallel and Distributed Processing Symposium. pp. 8–pp. IEEE (2005)

[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 23rd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). p. (to appear). IEEE (2023)

[13] Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in MPICH. The International Journal of High Performance Computing Applications **19**(1), 49–66 (2005)

[14] Thoman, P., Salzmann, P., Cosenza, B., Fahringer, T.: Celerity: High-level C++ for accelerator clusters. In: Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25. pp. 291–303. Springer (2019)