NCCL Profiler Plugin API – A Feasibility Study

Contents

1 Abstract

Artificial intelligence (AI) has established itself as a primary use case in high-performance computing (HPC) environments due to its compute-intensive and resource-intensive workloads. Analyzing and optimizing application performance is therefore essential to maximize efficiency and reduce costs. Many AI workloads involve communication between GPUs, often distributed across numerous GPUs in multi-node systems. The NVIDIA Collective Communication Library (NCCL) serves as the core library for implementing optimized communication primitives on NVIDIA GPUs. To provide detailed performance insights, NCCL offers a flexible profiler plugin API. This allows developers to directly integrate custom profiling tools into the library to extract detailed performance data on communication operations. This feasibility study explores the capabilities and integration mechanisms of the API.

First, this study provides background information on NCCL, followed by an explanation of the Profiler API accompanied with code examples and visualizations. Next, considerations for developers of the Profiler API and its potential integration with Score-P is discussed. Finally, the study concludes with a summary of the findings.

2 Introduction to NCCL

NCCL was first introduced by NVIDIA in 2015 at the Supercomputing Conference[1] with code being made available on GitHub[2]. The release of NCCL 2.0 in 2017 brought support for NVLink, however this was initially only available as pre-built binaries. With the release of NCCL 2.3 in 2018, it returned to being fully open source. The NCCL Profiler Plugin API was even later introduced with NCCL 2.23 in early 2025.

Before taking a closer look at the Profiler Plugin API, it is helpful to have some rudimentary understanding on certain designs in NCCL.

[1] `https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf`
[2] `https://github.com/NVIDIA/nccl`

2.1 Comparison to MPI

Although NCCL is inspired by the Message Passing Interface (MPI) in terms of API design and usage patterns, there are notable differences due to their respective focuses:

- **MPI**: Communication is CPU-based. A rank corresponds to a single CPU process within a communicator.
- **NCCL**: Communication is GPU-based, with CPU threads handling orchestration. A rank corresponds to a GPU device within a communicator; the mapping from ranks to devices is surjective. A single CPU thread can manage multiple ranks (i.e., multiple devices) in a communicator using the functions `ncclGroupStart` and `ncclGroupEnd`. A CPU thread can also manage multiple ranks from different communicators (i.e same device alloted by multiple ranks from different communicators) through communicator creation with `ncclCommSplit` or `ncclCommShrink`. This means the mapping from ranks to threads is also surjective.

2.2 Relevant NCCL internals

It helps to understand what NCCL does internally when an application calls the NCCL User API.

A typical NCCL application follows this basic structure:

- create nccl communicators
- allocate memory for computation and communication
- do computation and communication
- clean up nccl communicators

During NCCL communicator creation, NCCL internally spawns a thread called `ProxyService`. This thread lazily starts another thread called `ProxyProgress`[3], which handles network requests for GPU communication during collective and P2P operations. See Fig. 1.

`if`-guards ensure that these threads are created once per `ncclSharedResources`[4]. By default every NCCL communicator has its own shared resource. When the application calls `ncclCommSplit` or `ncclCommShrink`, where the original communicator was initialized with a `ncclConfig_t` with fields `splitShare` or `shrinkShare` set to `1`, the newly created communicator uses the same shared resource (and the proxy threads) as the parent communicator.

Later, whenever the application calls the NCCL User API, NCCL internally decides what network operations to perform and calls `ncclProxyPost` to post them to a proxyOpsPool (See Fig. 2).

The ProxyProgress thread reads from this pool when calling `ncclProxyGetPostedOps` and progresses the ops. See Fig. 3.

[3] https://github.com/NVIDIA/nccl/tree/master/src/proxy.cc
[4] https://github.com/NVIDIA/nccl/tree/master/src/include/comm.h

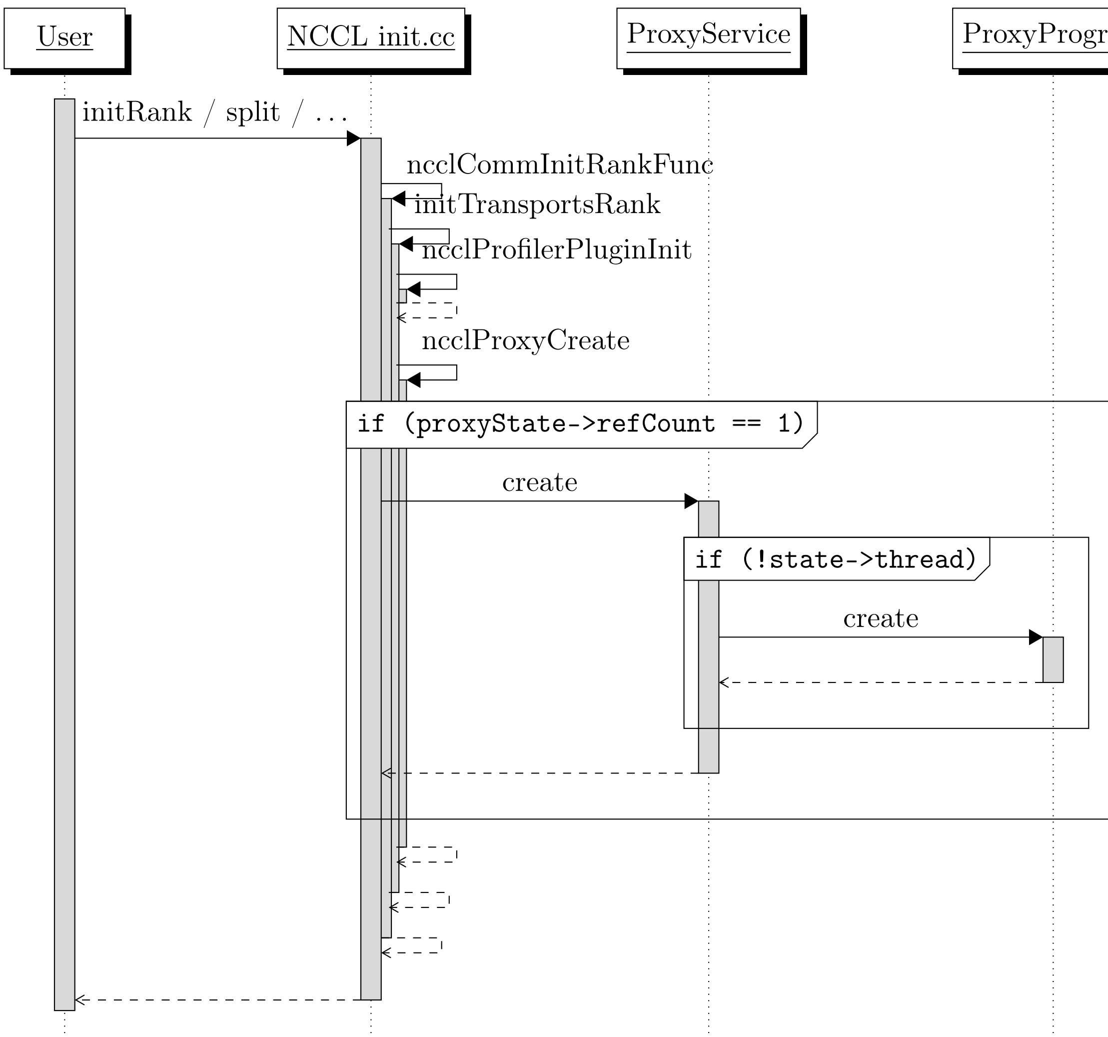

Figure 1: Thread creation: User API → NCCL internal init → create ProxyService → create ProxyProgress.

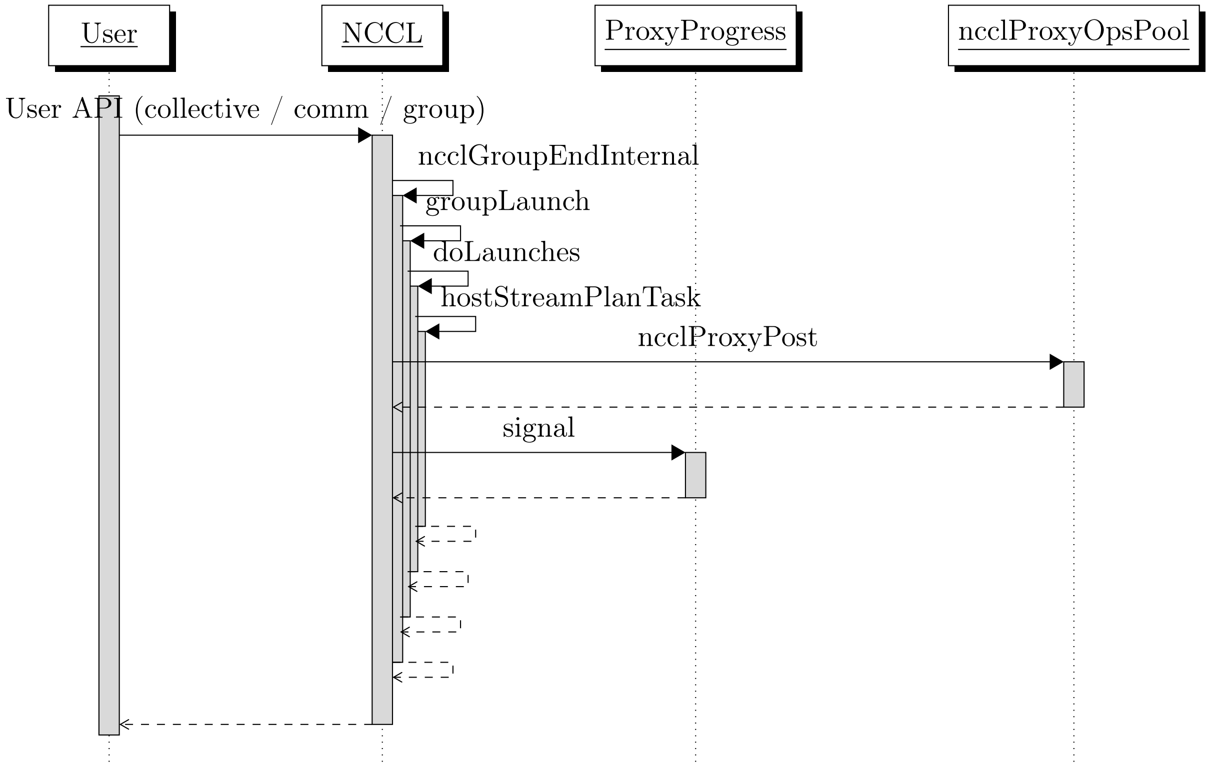

Figure 2: Flow from User API to `ncclProxyPost`

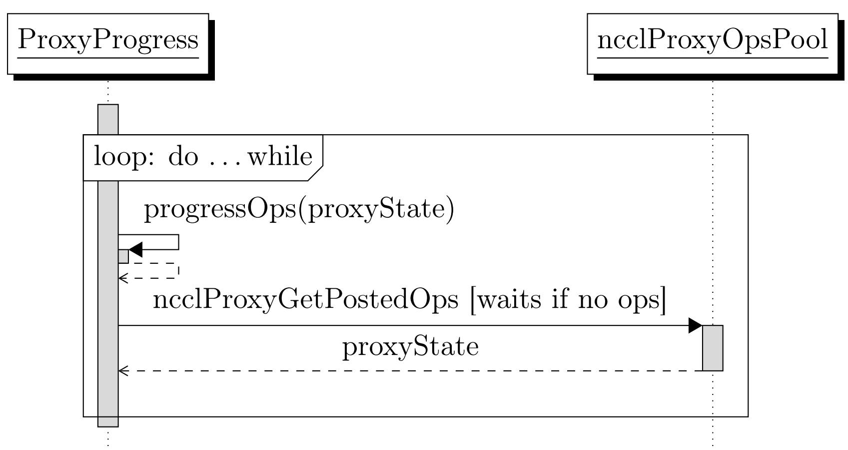

Figure 3: /src/proxy.cc `ncclProxyProgress` progressing loop: progress ops, then get posted ops (or wait).

Familiarity with this network activity pattern will aid in understanding the Profiler Plugin API's behavior discussed in the following section.

3 Profiler Plugin

Whenever a communicator is created, NCCL looks for the existence of a profiler plugin and loads it if it has not already been loaded on the process. NCCL then initializes the plugin with the created communicator. Whenever the application makes calls to the Collectives or P2p API (e.g. `ncclAllReduce`) with that communicator, NCCL calls the profiler API in different regions of the internal code. When the communicator is destroyed, the profiler plugin is unloaded if this was the only communicator on the process.

3.1 Profiler plugin loading mechanism

Each time a NCCL communicator is created, `ncclProfilerPluginLoad`[5] is called, where NCCL looks for a shared library that represents the profiler plugin by checking an environment variable. It then calls `dlopen`[6] and `dlsym` to load the library immediately with local symbol visibility:

```
profilerName = ncclGetEnv("NCCL_PROFILER_PLUGIN");
// ...
handle* = dlopen(name, RTLD_NOW | RTLD_LOCAL);
// ...
ncclProfiler_v5 = (ncclProfiler_v5_t*)dlsym(handle, "ncclProfiler_v5");
```

If the library has already been loaded on the process, this procedure is skipped. A `profilerPluginRefCount` keeps track of the number of calls to this procedure to ensure correct unloading during finalization. See Fig. 4. The NCCL documentation[7] also describes some further loading logic:

- If `NCCL_PROFILER_PLUGIN` is set: attempt to load the library with the specified name; if that fails, attempt libnccl-profiler-<NCCL_PROFILER_PLUGIN>.so.
- If `NCCL_PROFILER_PLUGIN` is not set: attempt `libnccl-profiler.so`.
- If no plugin was found: profiling is disabled.
- If `NCCL_PROFILER_PLUGIN` is set to `STATIC_PLUGIN`, the plugin symbols are searched in the program binary.

The plugin loading mechanism expects the struct variable name to follow the naming convention `ncclProfiler_v{versionNum}`, which also indicates the API version.

The profiler API has changed multiple times with newer NCCL releases. NCCL features a fallback mechanism to load older struct versions. However one instance is known, where a profiler plugin

[5] https://github.com/NVIDIA/nccl/tree/master/src/plugin/profiler.cc
[6] https://github.com/NVIDIA/nccl/tree/master/src/plugin/plugin_open.cc
[7] https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-profiler-plugin

being developed against the NCCL release 2.25.1 with Profiler API version 2, was unable to run with the latest NCCL release[8]. Around this time, the NCCL repository has undergone a refactor related to the profiler plugin.

Figure 4: User API → NCCL communicator init → load profiler plugin and call `profiler->init`.

[8] `https://github.com/variemai/ncclsee`

3.2 Profiler API

The plugin must implement a profiler API specified by NCCL by exposing a struct[9]. This struct should contain pointers to all functions required by the API. A plugin may expose multiple versioned structs for backwards compatibility with older NCCL versions.

```
ncclProfiler_v5_t ncclProfiler_v5 = {
  const char* name;
  ncclResult_t (*init)(...); // called when a communicator is created
  ncclResult_t (*startEvent)(...); // at start of operations/activities
  ncclResult_t (*stopEvent)(...); // at end of these operations/activities
  ncclResult_t (*recordEventState)(...); // to record state of certain operations
  ncclResult_t (*finalize)(...); // called when a communicator is destroyed
};
```

As of NCCL v2.29.2, version 6 is the latest, which was released on on Dec 24, 2025. This release happened well after the begin of the study, so the focus will be on version 5. Version 6 introduced additional profiler API callbacks for Copy-Engine based collective operations, otherwise version 6 and version 5 remain the same.

Five functions must be implemented for the API. Internally NCCL wraps calls to the profiler API in custom functions which are all declared in a single file[10].

NCCL invokes the profiler API at different levels to capture start/stop of NCCL groups, collectives, P2P, proxy, kernel and network activity. As the API function names suggest, this will allow the profiler to track these operations and activities as events.

The API functions and where NCCL invokes them are explained in the following sections.

3.2.1 init

`init` initializes the profiler plugin with a communicator. `init` is called immediately after `ncclProfilerPluginLoad`, which happens every time a communicator is created (see Fig. 4). This may happen multiple times for the same profiler instance, if further communicators are created on that process. NCCL passes follwing arguments:

```
ncclResult_t init(
  void** context, // out param - opaque profiler context
  uint64_t commId, // communicator id
  int* eActivationMask, // out param - bitmask for which events are tracked
  const char* commName, // user assigned communicator name
  int nNodes, // number of nodes in communicator
  int nranks, // number of ranks in communicator
  int rank, // rank identifier in communicator
  ncclDebugLogger_t logfn // logger function
);
```

[9] https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler/profiler_v5.h
[10] https://github.com/NVIDIA/nccl/tree/master/src/include/profiler.h

If the profiler plugin `init` function does not return `ncclSuccess`, NCCL disables the plugin.

`void** context` is an opaque handle that the plugin developer may point to any custom context object; this pointer is passed again in `startEvent` and `finalize`. This context object is separate per communicator.

The plugin developer should set `int* eActivationMask` to a bitmask[11], indicating which event types the profiler wants to track:

```
enum {
  ncclProfileGroup = (1 << 0), // group event type
  ncclProfileColl = (1 << 1), // host collective call event type
  ncclProfileP2p = (1 << 2), // host point-to-point call event type
  ncclProfileProxyOp = (1 << 3), // proxy operation event type
  ncclProfileProxyStep = (1 << 4), // proxy step event type
  ncclProfileProxyCtrl = (1 << 5), // proxy control event type
  ncclProfileKernelCh = (1 << 6), // kernel channel event type
  ncclProfileNetPlugin = (1 << 7), // network plugin-defined, events
  ncclProfileGroupApi = (1 << 8), // Group API events
  ncclProfileCollApi = (1 << 9), // Collective API events
  ncclProfileP2pApi = (1 << 10), // Point-to-Point API events
  ncclProfileKernelLaunch = (1 << 11), // Kernel launch events
};
```

The default value is to 0, which means no events are tracked by the profiler. Setting it to 4095 will track all events.

`ncclDebugLogger_t logfn` is a function pointer to NCCL's internal debug logger (`ncclDebugLog`). NCCL passes this so the plugin can emit log lines through the same channel and filtering as NCCL: the plugin may store the callback and call it with `(level, flags, file, line, fmt, ...)` when it wants to log. Messages then appear in NCCL's debug output (e.g. stderr or `NCCL_DEBUG_FILE`) and respect the user's `NCCL_DEBUG` level and subsystem mask. Using `logfn` keeps profiler output consistent with NCCL's own logs.

3.2.2 startEvent

`startEvent` is called when NCCL begins certain operations:

```
ncclResult_t startEvent(
  void* context, // opaque profiler context object
  void** eHandle, // out param - event handle
  ncclProfilerEventDescr_v5_t* eDescr // pointer to event descriptor
);
```

As of release v2.29.2 NCCL does not use the return value. `void** eHandle` may point to a custom event object; this pointer is passed again in `stopEvent` and `recordEventState`. `eDescr`[12] describes the started event.

[11] https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/nccl_profiler.h

[12] https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler/profiler_v5.h

The field `void* parentObj` in the event descriptor is the `eHandle` of a parent event (or null). The use of this field can be explained as following:

All User API calls to Collective or P2P operations will start a Group API event. When networking is required, ProxyCtrl Events may be emitted. Depending on the `eActivationMask` bitmask returned in the `init` function, further (child) events will be emitted in deeper regions of the nccl code base. It can be thought of as an event hierarchy[13] with several depth levels:

```
Group API event
    |
    +- Collective API event
    |   |
    |   +- Collective event
    |       |
    |       +- ProxyOp event
    |       |   |
    |       |   +- ProxyStep event
    |       |       |
    |       |       +- NetPlugin event
    |       |
    |       +- KernelCh event
    |
    +- Point-to-point API event
    |   |
    |   +- Point-to-point event
    |       |
    |       +- ProxyOp event
    |       |   |
    |       |   +- ProxyStep event
    |       |       |
    |       |       +- NetPlugin event
    |       |
    |       +- KernelCh event
    |
    +- Kernel Launch event

ProxyCtrl event
```

The `parentObj` inside `eDescr` will be a reference to the `eHandle` of the respective parent event for the current event according to this hierarchy. Thus, if the `eActivationMask` set during `init` enables tracking for event types lower in the hierarchy, NCCL always also tracks their parent event types.

3.2.3 stopEvent

```
ncclResult_t stopEvent(void* eHandle); // handle to event object
```

`stopEvent` tells the plugin that the event has stopped. `stopEvent` for collectives simply indicates to the profiler that the collective has been enqueued and not that the collective has been completed.

As of NCCL v2.29.2 NCCL does not use the return value.

[13] `https://github.com/NVIDIA/nccl/tree/master/ext-profiler/README.md`

`stopEvent` is called in the same functions that call `startEvent`, except for the GroupApi event. Fig. 5 shows when NCCL emits `startEvent` and `stopEvent` after a user API call. The ProxyProgress thread also emits `startEvent` and `stopEvent` while progressing ops (see Fig. 6).

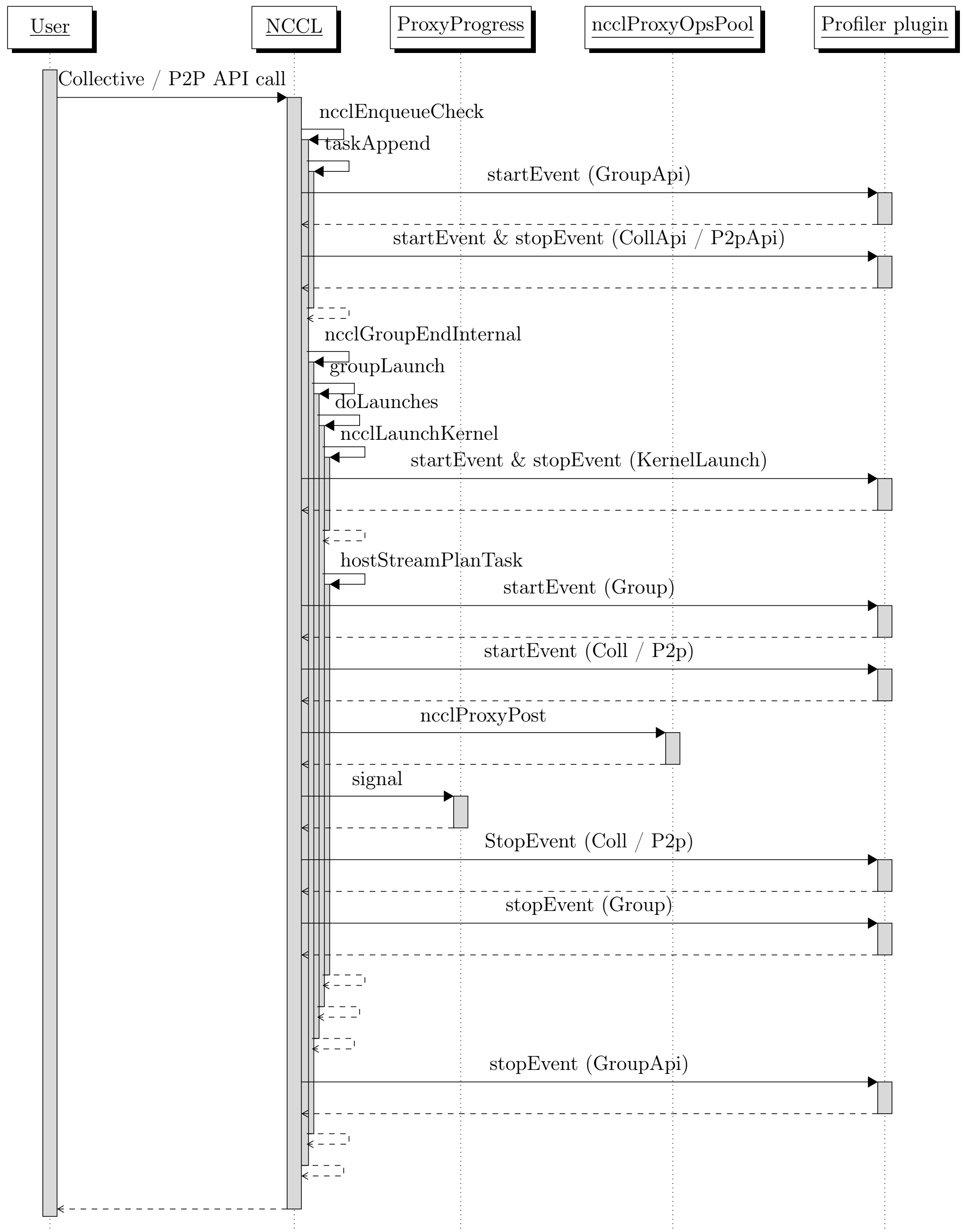

Figure 5: Flow from NCCL API calls to profiler events. In case of `ncclGroupStart / ncclGroupEnd`. multiple events of everything (except GroupApi) are called. Internally, some Collectives (e.g. ncclAlltoAll) are implemented as multiple p2p ops, triggering many P2pApi and P2p events.

Figure 6: `ncclProxyProgress`: progressOps emits ProxyStep/KernelCh/NetPlugin events. getPostedOps emits ProxyOp events. Several events ProxyCtrl are also emitted

`op->progress()` progresses transport specific ops. This is implemented as a function pointer type[14]. Confusingly the variable is called ‘op‘, although its type is `ncclProxyArgs` and *not* `ncclProxyOp`.

```
typedef ncclResult_t (*proxyProgressFunc_t)(struct ncclProxyState*, struct ncclProxyArgs
    *);

struct ncclProxyArgs {
  proxyProgressFunc_t progress;
  struct ncclProxyArgs* next;
  /* other fields */
}
```

This allows calls to different the implementations of the `progress` function for different transport

[14] `https://github.com/NVIDIA/nccl/tree/master/src/include/proxy.h`

methods[15][16][17][18]. Each implementations calls the profiler API to inform about a different event type (ProxyStep, KernelCh or Network plugin specific).

3.2.4 recordEventState

```
ncclResult_t recordEventState(
  void* eHandle,
  ncclProfilerEventState_v5_t eState,
  ncclProfilerEventStateArgs_v5_t* eStateArgs
);
```

Some event types can be updated by NCCL through `recordEventState` (state and attributes)[19]. `recordEventState` is called in the same functions that call `startEvent` and are happening after `startEvent`.

3.2.5 finalize

```
ncclResult_t finalize(void* context);
```

After a user API call to free resources associated with a communicator, `finalize` is called. Afterwards, a reference counter tracks how many communicators are still being tracked by the profiler plugin. If it reaches 0, the plugin will be closed via `dlclose(handle)`. Fig. 7 depicts the flow from user API call to `finalize`.

[15] https://github.com/NVIDIA/nccl/tree/master/src/transport/net.cc
[16] https://github.com/NVIDIA/nccl/tree/master/src/transport/coll_net.cc
[17] https://github.com/NVIDIA/nccl/tree/master/src/transport/p2p.cc
[18] https://github.com/NVIDIA/nccl/tree/master/src/transport/shm.cc
[19] https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler/profiler_v5.h

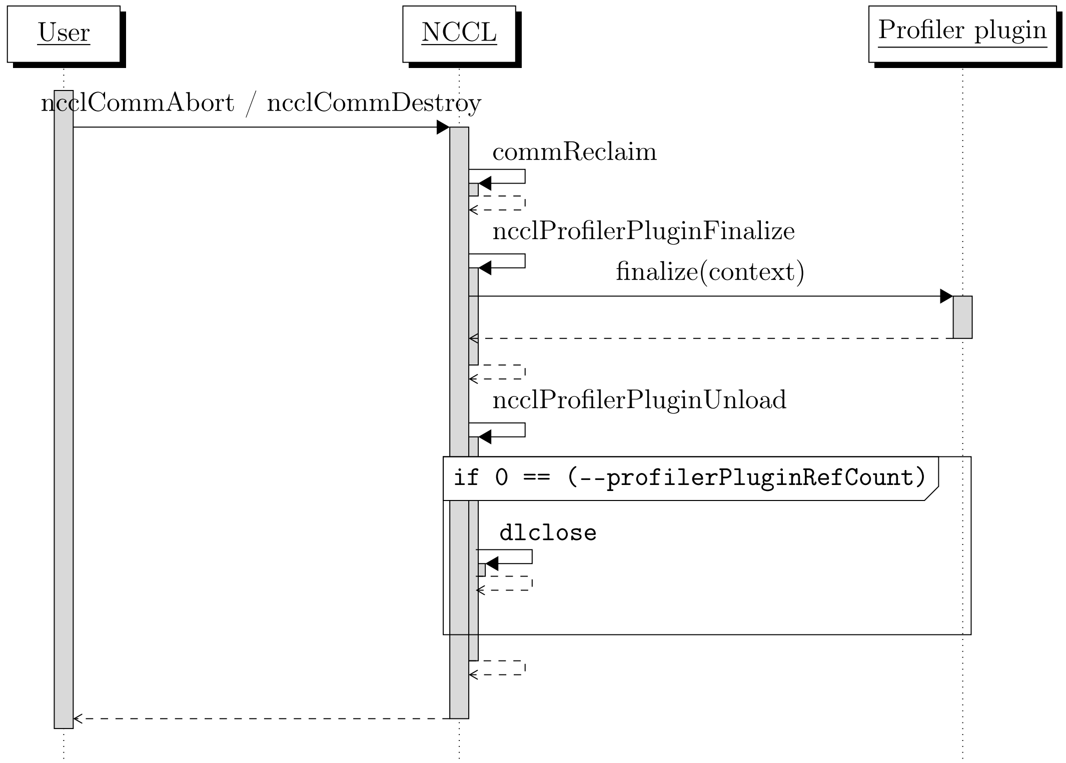

Figure 7: User API → `commReclaim` → `finalize` → plugin unload.

3.2.6 name

The profiler plugin struct also has a `name` field. The name field should point to a character string with the name of the profiler plugin. It will be used for all logging, especially when `NCCL_DEBUG=INFO` is set.

4 Code examples and visualizations

The following examples illustrate the profiling behavior for different user application settings:

- One Device per Thread
- Multiple Devices per Thread via `ncclGroupStart` and `ncclGroupEnd`
- One Device per Thread and aggregated operations via `ncclGroupStart` and `ncclGroupEnd`

A profiler plugin that logs all call information to a file has been developed and is used in all examples. An exemplary illustration is shown below:

```
struct MyContext { /* custom context struct */ };
struct MyEvent { /* custom event struct */ };
```

```c
MyEvent* allocEvent(args) { /* handles event allocation */ }
uint64_t getTime() { /* gets time */ }
void writeJsonl() { /* writes call details to process specific log file as structured
    jsonl */ }

ncclResult_t myInit( /* args - **context, *eActivationMask, ... */ ) {
  *context = malloc(sizeof(struct MyContext));
  *eActivationMask = 4095; /* enable ALL event types */

  writeJsonl(getTime(), "Init", args);
  return ncclSuccess;
}

ncclResult_t myStartEvent( /* args - **eHandle, ... */ ) {
  *eHandle = allocEvent(args);

  writeJsonl(getTime(), "StartEvent", args);
  return ncclSuccess;
}

ncclResult_t myStopEvent(void* eHandle) {
  writeJsonl(getTime(), "StopEvent", eHandle);

  free(eHandle)
  return ncclSuccess;
}

ncclResult_t myRecordEventState( /* args - ... */ ) {
  writeJsonl(getTime(), "RecordEventState", args);
  return ncclSuccess;
}

ncclResult_t myFinalize(void* context) {
  writeJsonl(getTime(), "Finalize", args);

  free(context);
  return ncclSuccess;
}

ncclProfiler_v5_t ncclProfiler_v5 = {
  "MyProfilerPlugin",
  myInit,
  myStartEvent,
  myStopEvent,
  myRecordEventState,
  myFinalize,
};
```

Alongside the logging profiler plugin, a visualization tool as been built, that ingests the profiler logs to inspect the exact behavior of internal calls from NCCL to the Profiler API. It displays the events as colored bars on a timeline and separates them on different lanes. Each lane also displays some information about the communicator, rank and thread corresponding to the event. Additionally, blue dotted lines indicate the relationship between events according to the `parentObj` field and red lines indicate which collective events belong to the same collective operation.

Further, a hover feature was added to inspect all details of an event, however this feature is not used in the following illustrative examples.

4.1 One Device per Thread

This example visualizes an AllReduce collective across multiple GPUs (see Fig. 8 and Fig. 9). Each NCCL thread manages a single GPU. This may be achieved by starting out with the same number of MPI tasks with each task running single threaded; or by having less MPI tasks, but the tasks create multiple thread workers. Custom initialization without MPI is also possible if desired.

```
// broadcast a commId

// ...

ncclCommInitRank(&rootComm, nRanks, commId, myRank);

// ...

ncclAllReduce(sendBuff, recvBuff, BUFFER_SIZE, ncclFloat, ncclSum, rootComm, streams);

// ...

ncclCommDestroy(rootComm);
```

The profiler API calls are visualized in Fig. 8 and Fig. 9. Below follows a full description of the calls to the profiler API induced by the example program:

First, the profiler API `init` is called for each rank. This occurs during NCCL's internal communicator creation, when the application calls `ncclCommInitRank`. After the application calls `ncclAllReduce`, many Profiler API calls to `stateEvent`, `stopEvent`, and `recordEventState` are triggered: Intially, startEvent for the `groupApi` (green bar) is called. Below it, the startEvent and soon the stopEvent for the AllReduce `collApi` event are called. The yellow bar shows when NCCL enqueues the GPU kernel launch (`KernelLaunch` event). The two bars below represent the `group` and `coll` events. NCCL also spawns a proxy progress thread per rank, which does additional profiler API calls. The first red `ProxyCtrl` event shows the proxy progress thread was asleep. Next, a new `ProxyCtrl` event shows time for the proxy thread to append proxy ops. Then, appended ops start progressing (`ProxyOps` events), which in `op->progress()` starts `ProxyStep` and `KernelCh` events that inform about low level network activity in updates via `recordEventState` like `ProxyStepRecvGPUWait` (see Fig. 9). Network activity eventually completes and the AllReduce collective finishes. The next `ProxyCtrl` event only shows the proxy thread sleeping again. Finally, profiler `finalize` is called, which happens when the application cleans up NCCL communicators and no further communicators are tracked in the profiler in each respective thread.

`ProxyStep` events are emitted in cross node communication environments. If this type of communication is not required, then `ProxyStep` events will not happen either.

Figure 8: One device per thread: A visualization of the calls generated to the Profiler API, starting from communicator creation, followed by a collective operation and communicator destruction. `ProxyStep` events have been omitted for visual clarity, see Fig. 9 for a depiction.

Figure 9: One device per thread: In Fig. 8 `ProxyStep` events have been omitted for visual clarity. However, in multinode settings, many additional profiler API calls for proxyStep events happen, informing about the low level network steps in their event details via `recordEventState` (indicated as red circles above each of the event bars). The blue dotted lines indicate the `parentObj` of each proxyStep event, which are the above proxyOp events.

4.1.1 Multiple Devices per Thread (ncclGroup)

In this example[20], one NCCL thread manages all GPUs on the same node. This is achieved by wrapping communication initialization in `ncclGroupStart` and `ncclGroupEnd` for each managed GPU. In this orchestration setting, **NVIDIA's documentation states that collective API calls should also be wrapped in ncclGroup.** Here, only one collective operation (per device) is inside the ncclGroup:

```
// broadcast a commId
```

[20] `https://github.com/NVIDIA/nccl/tree/master/examples/03_collectives/01_allreduce/`

```
// ...

ncclGroupStart();
for (int i=0; i<ngpus; i++) {
  cudaSetDevice(dev);
  ncclCommInitRank(comms+i, ngpus*nRanks, id, myRank*ngpus+i);
}
ncclGroupEnd();

// alternatively to above method, NCCL provides the convenience function
// ncclCommInitAll();

// ...

ncclGroupStart();
for (int i = 0; i < num_gpus; i++) {
  ncclAllReduce( /* ... */ );
}
ncclGroupEnd();

// ...

for (int i = 0; i < num_gpus; i++) {
  ncclCommDestroy(comms[i]);
}
```

In this example case, the profiler API behavior remains largely the same: The one difference is that NCCL internally calls the profiler API groupApi event only one time in total for aggregated operations within a thread. Otherwise all other events are processed as usual and are called their usual amount of times irrespective of `ncclGroup`. This is visualized in Fig. 10. This behaviour also holds true within a process. It also holds when grouping (single) collectives for different communicators.

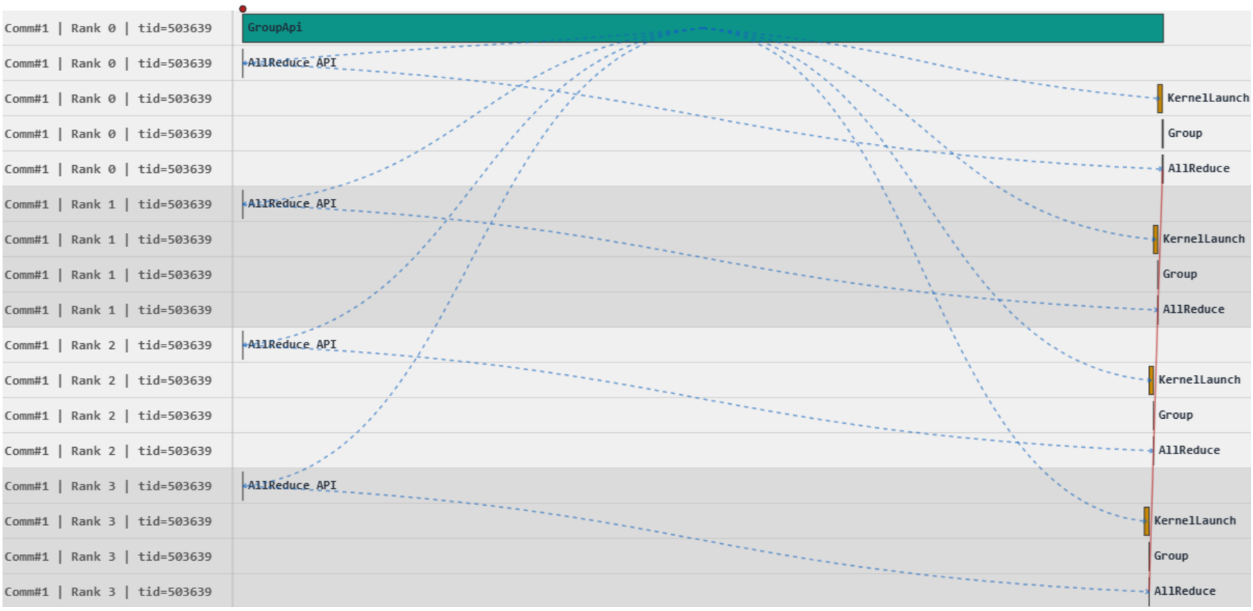

Figure 10: Multiple devices per thread: Events from the proxy thread as well as init and finalize calls are omitted. Collective API calls from multiple GPUs managed by a single thread only trigger a single `GroupApi` event.

4.1.2 Aggregated operations

In this example, the setting is such that only a single GPU is managed by a thread, but multiple collective operations are grouped (i.e. to optimize communication efficiency):

```
// broadcast a commId

// ...

ncclCommInitRank(&rootComm, nRanks, rootId, myRank);

// ...

ncclGroupStart();
ncclAllReduce( /* ... */ );
ncclBroadcast( /* ... */ );
ncclReduce( /* ... */ );
ncclAllGather( /* ... */ );
ncclReduceScatter( /* ... */ );
ncclGroupEnd();

// ...
```

The behavior changes can be described as follow:

- single GroupApi event per thread

- single KernelLaunch event per thread
- single Group event per thread

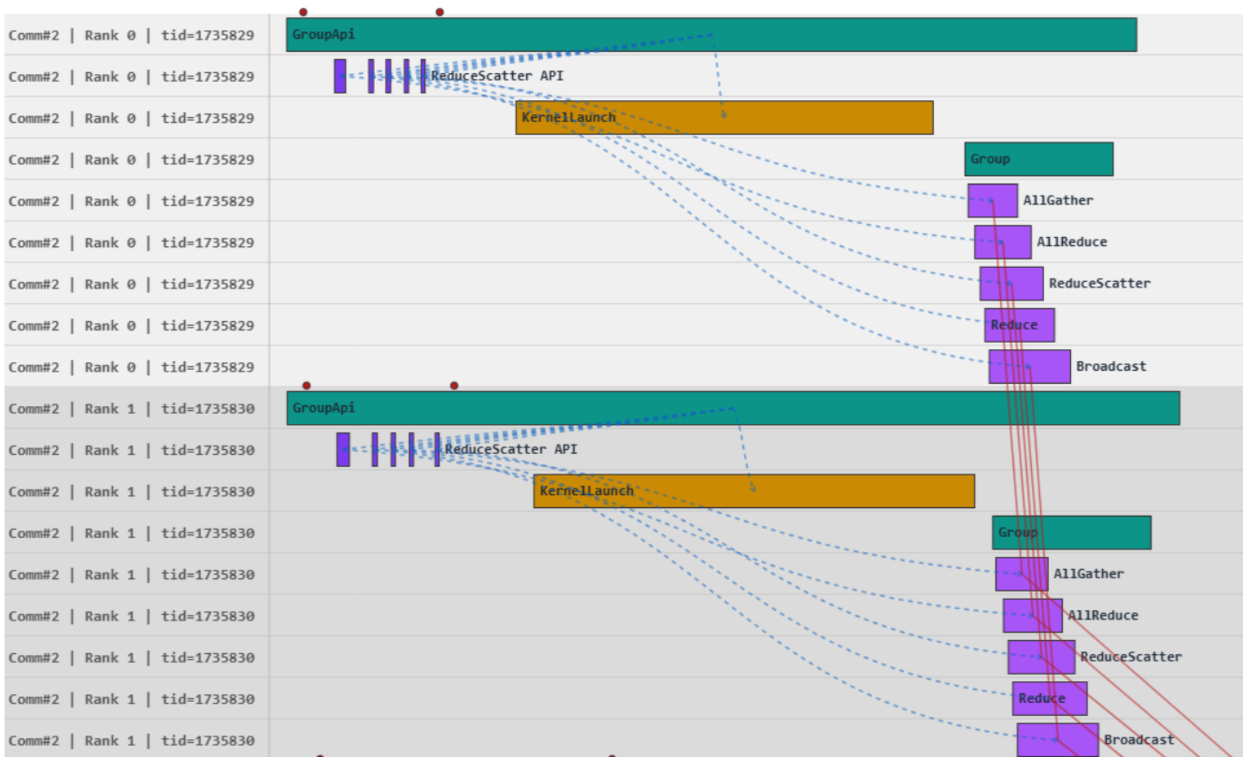

Figure 11: one GPU per thread with aggregated operations: multiple collective calls are grouped together and nccl does only a single kernel launch per thread.

5 Performance and scalability of the Profiler Plugin API

Experiments were run to assess the performance and scalability of profiler plugins. These experiments measure the overhead induced internally by NCCL to serve the profiler plugin, but do not intend to measure the performance of a profiler plugin itself as the plugin is fully customizable to the needs of the developer.

Thus, the profiler developed for the experiments only initializes a dummy context struct, returns NULL for event handles and tracks all events (`eActivationMask` set to 4095).

```
// an 'empty' NCCL Profiler Plugin

struct MyContext {
  char dummy;
};

ncclResult_t myInit(void** context, uint64_t commId, int* eActivationMask, const char*
    commName, int nNodes, int nranks, int rank, ncclDebugLogger_t logfn) {
```

```c
  *context = malloc(sizeof(struct MyContext));
  *eActivationMask = 4095; /* enable ALL event types */
  return ncclSuccess;
}

ncclResult_t myStartEvent(void* context, void** eHandle, ncclProfilerEventDescr_v5_t*
    eDescr) {
  *eHandle = NULL;
  return ncclSuccess;
}

ncclResult_t myStopEvent(void* eHandle) {
  return ncclSuccess;
}

ncclResult_t myRecordEventState(void* eHandle, ncclProfilerEventState_v5_t eState,
    ncclProfilerEventStateArgs_v5_t* eStateArgs) {
  return ncclSuccess;
}

ncclResult_t myFinalize(void* context) {
  free(context);
  return ncclSuccess;
}

ncclProfiler_v5_t ncclProfiler_v5 = {
  "EmptyProfiler",
  myInit,
  myStartEvent,
  myStopEvent,
  myRecordEventState,
  myFinalize,
};
```

For testing the performance overhead in collective and P2P operations, **nccl-tests** from NVIDIA was used[21].

The applications `sendrecv_perf` and `all_reduce_perf` were launched with follwing test parameters: message size 64 B, 1 000 000 iterations per size, 100 warmup iterations. Single-node jobs used one node and 4 GPUs; multi-node jobs used 2 nodes, 4 GPUs per node, 8 MPI ranks in total. For each experiment, the application was run once without the profiler and once with the empty profiler plugin.

The Table 1 shows the average latency per operation (time in μs) across iterations. The empty profiler adds roughly 8 to 9 μs overhead per operation in single-node runs (4 GPUs), but introduces negligible overhead in multi-node runs (8 GPUs across 2 nodes).

[21] `https://github.com/NVIDIA/nccl-tests`

Table 1: Profiler overhead: nccl-tests `sendrecv_perf` (P2P) and `all_reduce_perf` (collectives). Latency averaged over 1M iterations.

Test	Environment	Without profiler (μs)	With profiler (μs)
P2P (`sendrecv_perf`)	Single-node (4 GPUs)	14.3	23.88
	Multi-node (2×4 GPUs)	13.05	12.95
Collectives (`all_reduce_perf`)	Single-node (4 GPUs)	14.96	23.29
	Multi-node (2×4 GPUs)	17.99	18.34

Using the profiler plugin when scaled to many gpus across multiple nodes is effortless and did not require any changes in the profiler plugin for the used code examples and experiments.

6 Discussion

6.1 Considerations for developers of a Profiler Plugin

Profiler Visualization. The visualization tool used in the code examples is helpful for understanding the internal call behavior to the Profiler API by NCCL and will be made available along with this report. It may serve as a reference to compare against for other developers that build a profiler plugin or visualizer

Correlating Collective Events with `seqNumber`. When profiling is enabled, NCCL counts the number of calls for each type of collective function per communicator.

/src/include/comm.h

```
struct ncclComm {
  uint64_t seqNumber[NCCL_NUM_FUNCTIONS];
  /* other fields */
}
```

/src/plugin/profiler.cc

```
ncclResult_t ncclProfilerStartTaskEvents(struct ncclKernelPlan* plan) {
  /* other code */
  __atomic_fetch_add(&plan->comm->seqNumber[ct->func], 1, __ATOMIC_RELAXED);
  /* other code */
}
```

This value is present in the `eDescr` for collective events and can be used to identify which collectives operations belong together across processes (see Fig. 12).

Tracing low level activity back to NCCL API calls with `parentObj`. If a plugin developer wants utilize this field, they should ensure that potential address reuse does not create ambiguity to what the parentObj was originally pointing to. *Custom memory management is advised.* This field is useful when trying to understand which user API call triggered which events of lower level operations or activity such as network activity (see Fig. 12).

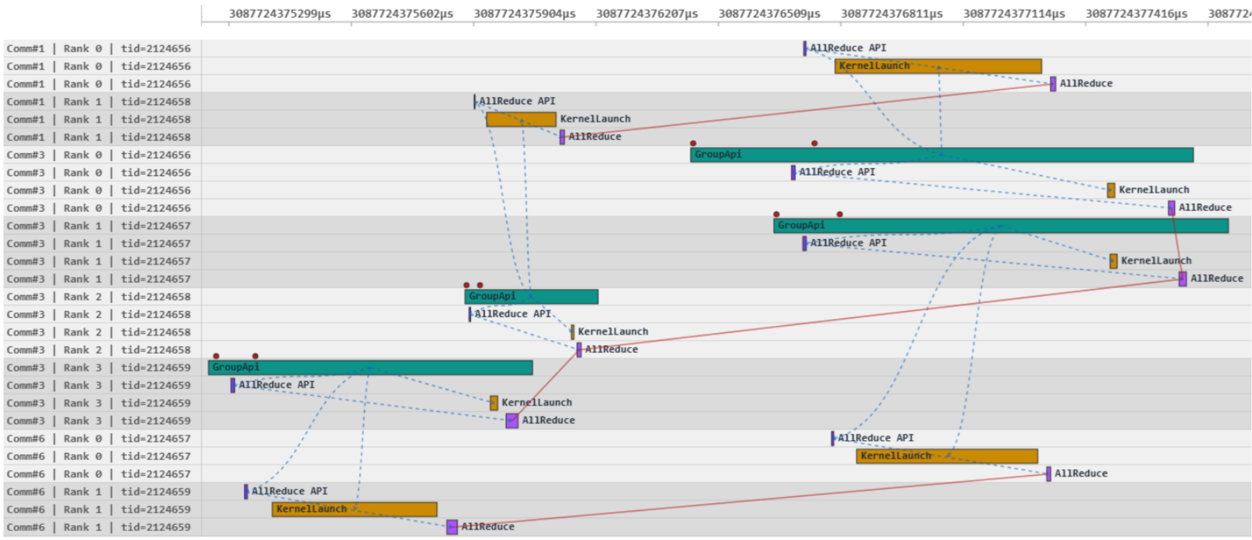

Figure 12: An example illustrating how `parentObj` and `seqNumber` can be used to better understand the timing of concurrent collective operations.

Process origin for profiler callbacks with PXN enabled. Unless Setting the environment variable `NCCL_PXN_DISABLE`=0 (default 1), due to PXN (PCIe x NVLink) some proxy ops may be progressed in a proxy thread from another process, different to the one that originally generated the operation. Then `parentObj` in `eDescr` is not safe to dereference; the `eDescr` for `ProxyOp` events includes the originator's PID, which the profiler can match against the local PID. The `eDescr` for `ProxyStep` does not provide this field. However a workaround is possible:

The passed `context` object in `startEvent` is also unsafe to dereference due to PXN. the profiler plugin developer may internally track initialized contexts and whether the passed `context` belongs to the local process. This is also indicative of PXN.

Tracking communicator parent–child relationships. With the current Profiler plugin API, it is not possible to detect whether a communicator originates from another one (e.g., via `ncclCommSplit` or `ncclCommShrink`). The plugin's `init` callback only receives a single communicator ID (`commId`, which corresponds to `comm->commHash`), as well as `commName`, `nNodes`, `nRanks`, and `rank`; there is no `parentCommId` or similar argument. In split/shrink, the `commHash` of the child node is calculated internally as a one-way digest of the `commHash` of the parent node and the split parameters (`splitCount`, `color`). Therefore, the relationship cannot be restored based on the ID alone.

6.2 Known limitations

Kernel event instrumentation uses counters exposed by the kernel to the host and the proxy progress thread. Thus the proxy progress thread infrastructure is shared between network and profiler. If the proxy is serving network requests, reading kernel profiling data can be delayed, causing loss of accuracy. Similarly, under heavy CPU load and delayed scheduling of the proxy progress thread, accuracy can be lost.

From profiler version 4, NCCL uses a per-channel ring buffer of 64 elements. Each counter is complemented by two timestamps (ptimers) supplied by the NCCL kernel (start and stop of the operation in the kernel). NCCL propagates these timestamps to the profiler plugin so it can convert them to the CPU time domain.

(Source: **/ext-profiler/README.md**)

6.3 Potential Integration with Score-P

The Score-P measurement infrastructure[22] is a highly scalable and easy-to-use tool suite for profiling and event tracing of HPC applications. It supports a number of analysis tools. Currently, it works with Scalasca, Vampir, and Tau and is open for other tools and produces OTF2 traces and CUBE4 profiles.

For Score-P, it is important that communicator identities are unique. NCCL achieves this for `ncclGetUniqueId`[23] without a central coordinator. Each call fills the bootstrap handle embedded in `ncclUniqueId` with

- a random 64-bit `magic` value from `/dev/urandom`, and
- the socket `address` of a new listening socket (IP, port), whose port is chosen by the operating system.

The pair (random magic, IP+port) is unique in practice: different MPI tasks or repeated calls in one process each get distinct random magic and a distinct OS-assigned port so collisions are avoided.

The NCCL profiler plugin is callback-driven *by NCCL*. NCCL loads it via `dlopen` and invokes `startEvent`, `stopEvent`, and `recordEventState` during collective and P2P operations.

In one potential integration strategy, a developer would implement the NCCL profiler API and could use Score-P's user instrumentation API (e.g., `SCOREP_USER_REGION_BY_NAME_BEGIN`/`END`) to inject NCCL Profiler Events as regions into Score-P. The region name could be derived from the event descriptor (e.g., `collApi.func` for `ncclAllReduce`). In this design, NCCL drives the profiler and the profiler forwards events into Score-P. NCCL collective operations then appear as regions in Score-P profiles and traces.

[22] `https://www.vi-hps.org/projects/score-p/overview`
[23] `https://github.com/NVIDIA/nccl/tree/master/src/init.cc`

Alternatively, NVTX and CUPTI could be leveraged with CUDA adapters from Score-P. Similarly, the NCCL profiler plugin would use the NVTX API[24] to annotate regions. Additionally, kernel tracing of with CUPTI can be integrated directly into the NCCL profiler plugin. The CUPTI API provides functions to correlate application code regions with CUPTI activity via `cuptiActivityPushExternalCorrelationId` and `cuptiActivityPopExternalCorrelationId`. This can be handily integrated with the profiler API, whenever `KernelLaunch` events are started and stopped, while continuously incrementing the correlation id in a thread-safe manner. Fig. 13 provides an example visualization of this method. CUPTI can be initialized and cleaned up within the profiler plugin's own `init` and `finalize` functions.

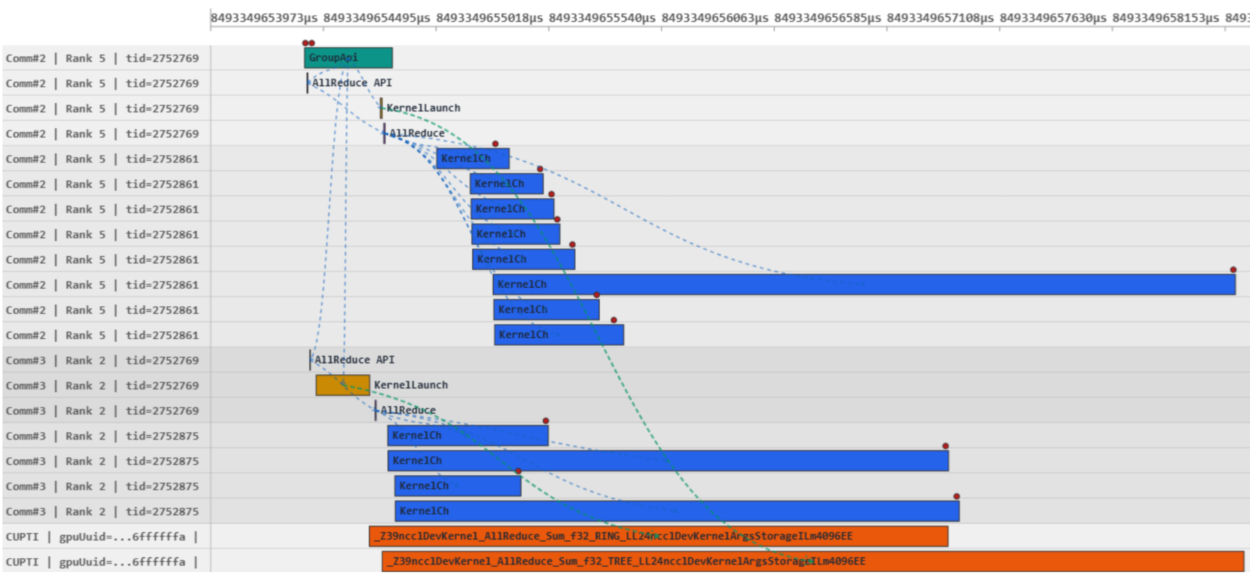

Figure 13: CUPTI activity is visualized as orange event bars. Through a correlation Id, it is possible to trace the activity back to `KernelLaunch` events

7 Conclusion

This feasibility study examined the NCCL Profiler Plugin API and its suitability for integration with Score-P. The report provided background on NCCL and its design, explained how the profiler plugin is detected and loaded, and described the API definition with its five core callbacks `init`, `startEvent`, `stopEvent`, `recordEventState`, `finalize`. Code examples and visualizations illustrate the event flow from API calls to NCCL's internal profiler callbacks. Performance experiments showed that an empty profiler adds roughly 8–9 μs overhead per operation in single-node runs but introduces negligible overhead in multi-node runs, and scaling to many GPUs across nodes required no changes to the profiler plugin. The discussion covered developer considerations, known limitations, and a potential integration strategy with Score-P.

The NCCL Profiler API allows for highly customized plugins tailored to the analysis needs, whether for simple timing, kernel tracing via CUPTI, or integration with external tools such as Score-P.

[24] https://nvidia.github.io/NVTX/doxygen/group___m_a_r_k_e_r_s___a_n_d___r_a_n_g_e_s.html

A notable advantage is its low overhead: NVIDIA advertises their `inspector`[25] implementation as efficient enough for "always-on" profiling in production. On the downside, profiler plugins may require maintenance and active development, since NCCL is actively developed. API versions evolve and new features are being introduced.

[25] `https://github.com/NVIDIA/nccl/tree/master/ext-profiler/inspector`