NCCL Profiler Plugin API – A Feasibility Study

Contents

1 Abstract **2**

2 About NCCL **2**

2.1 Comparison to MPI 2

2.2 Relevant NCCL internals 3

3 The Profiler API **5**

3.1 How NCCL detects the profiler plugin 5

3.2 The profiler API definition 8

3.2.1 init 9

3.2.2 startEvent 10

3.2.3 stopEvent 11

3.2.4 recordEventState 14

3.2.5 finalize 14

3.2.6 name 15

3.3 Code Examples 15

3.3.1 One Device per Thread 16

3.3.2 Multiple Devices per Thread (ncclGroup) 18

3.3.3 One Device per Thread & Grouped Collectives 20

3.4 Performance and scalability of the Profiler Plugin API 21

4 Discussion **23**

4.1 Considerations for developers of a Profiler Plugin 23

4.2 Known limitations 26

4.3 Potential Integration with Score-P 26

5 Conclusion **26**

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 is explained 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 About NCCL

NCCL (pronounced "Nickel") was first introduced by NVIDIA in 2015 at the Supercomputing Conference (SC15), with an accompanying presentation highlighting its optimized collectives for multi-GPU systems. Although code was made available on GitHub[1], the release of NCCL 2.0 in 2017, which brought support for NVLink, 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.

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:

[1]`https://github.com/NVIDIA/nccl`

- **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
createNcclComm();

// allocate memory for computation and communication
prepareDeviceForWork();

// do computation and communication
callNcclCollectives();
// ...

// finalize and clean up nccl communicators
cleanupNccl();
```

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

The guards `if (proxyState->refCount == 1)` and `if (!state->thread)` ensure that these threads are created once per shared resource (struct `ncclSharedResources`). The SharedResource has a ProxyState field. The fields in ProxyState are used to ensure only one instance of each thread exists:

/src/include/comm.h

```
struct ncclSharedResources {
  struct ncclComm* owner; /* communicator which creates this shared res. */
  struct ncclProxyState* proxyState;
  // other fields
}
```

/src/include/proxy.h

```
struct ncclProxyState {
  int refCount;
  pthread_t thread;
```

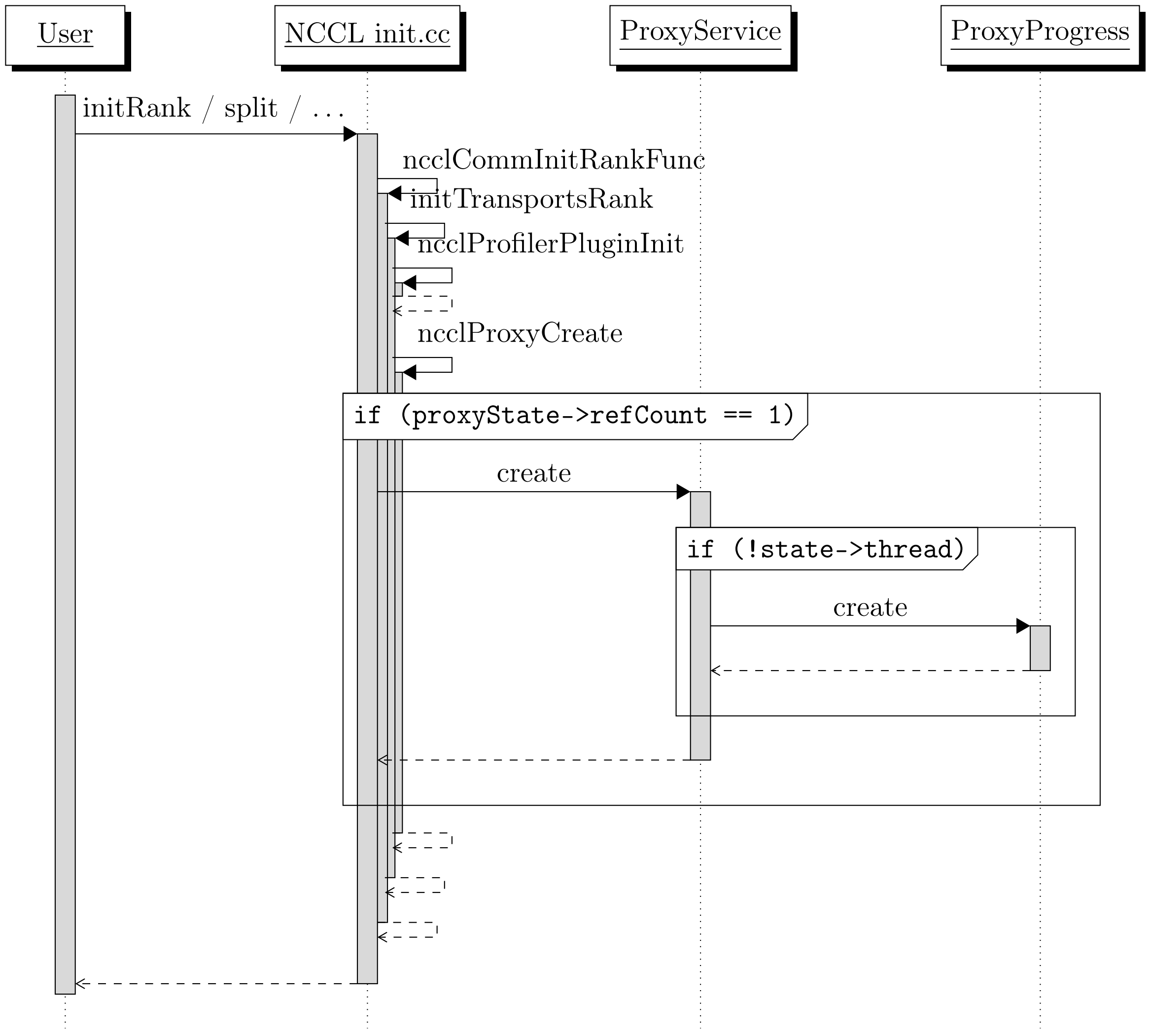


Figure 1: Thread creation: User API → NCCL internal init → create ProxyService → create ProxyProgress.

```
  ncclProxyProgressState progressState;
  // other fields
}

struct ncclProxyProgressState {
  struct ncclProxyOpsPool* opsPool;
  // other fields
}

struct ncclProxyOpsPool {
  struct ncclProxyOp ops[MAX_OPS_PER_PEER*NCCL_MAX_LOCAL_RANKS];
  // other fields
}

struct ncclProxyOps {
  // other fields
}
```

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 shares the shared resource (and the proxy threads) with the parent communicator.

```
/* proxyState is shared among parent comm and split comms
comm->proxyState->thread is pthread_join()'d by commFree() in init.cc
when the refCount reduces down to 0.  */
```

(Quoted from /**src**/**proxy.cc**)

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 Figure 2).

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

Understanding this behaviour for network-related activity is helpful in relation to the behavior of Profiler Plugin API in the next section.

3 The Profiler API

3.1 How NCCL detects the profiler plugin

When a NCCL communicator is created, NCCL looks for a shared library that represents the profiler plugin by checking an environment variable:
`profilerName = ncclGetEnv("NCCL_PROFILER_PLUGIN");`
It then calls
`handle* = dlopen(name, RTLD_NOW | RTLD_LOCAL);`

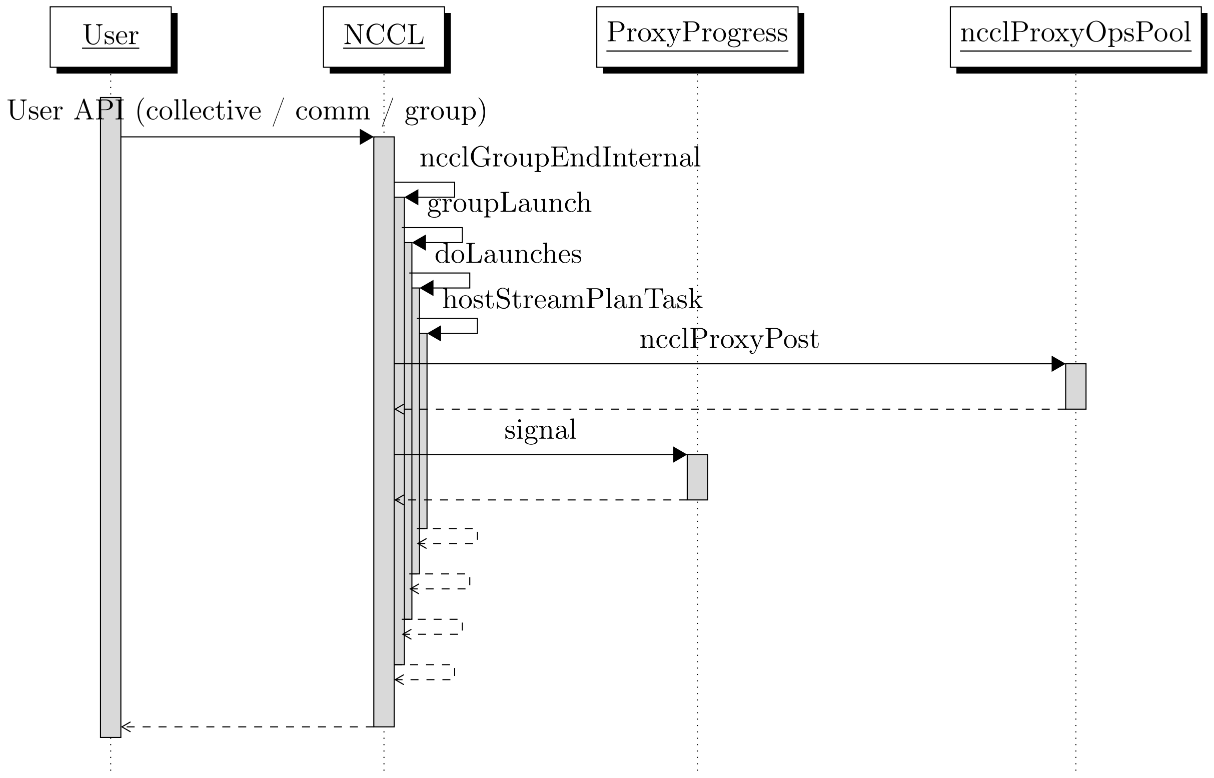


Figure 2: Flow from User API to `ncclProxyPost`

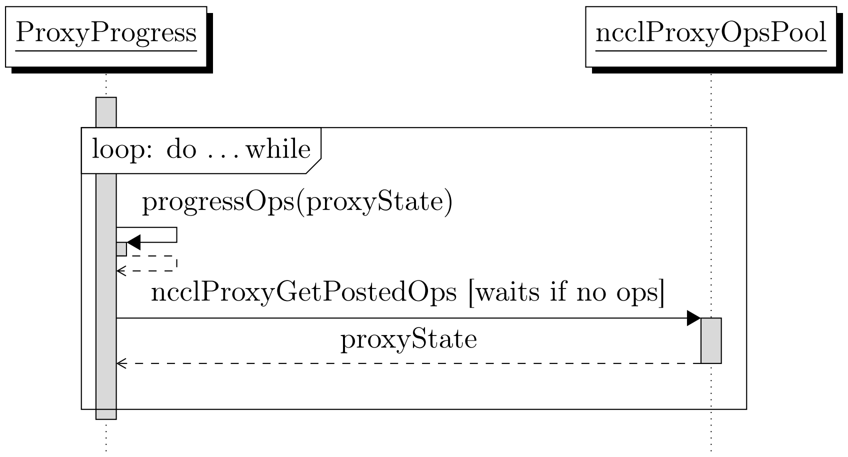


Figure 3: /**src**/**proxy.cc** `ncclProxyProgress` progressing loop: progress ops, then get posted ops (or wait).

and
`ncclProfiler_v5 = (ncclProfiler_v5_t*)dlsym(handle, "ncclProfiler_v5");`
to load the library immediately with local symbol visibility. See Figure 4.

- 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.

(Source:
`https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-profiler-plugin`)

The profiler plugin is loaded when creating a communicator (before proxy thread creation). 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 being developed against the NCCL release 2.25.1 with Profiler API version 2, was unable to run with the latest NCCL release[2]. NCCL developers may be required to actively maintain older API versions, to ensure they safely work when old behaviour is getting deprecated and do not unexpectedly get handed new features from new API versions, of which the Profiler Plugin developer is not aware of (when faithfully implementing old API versions).

The exact implementation is in **/src/plugin/plugin_open.cc** and **/src/plugin/profiler.cc**.

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

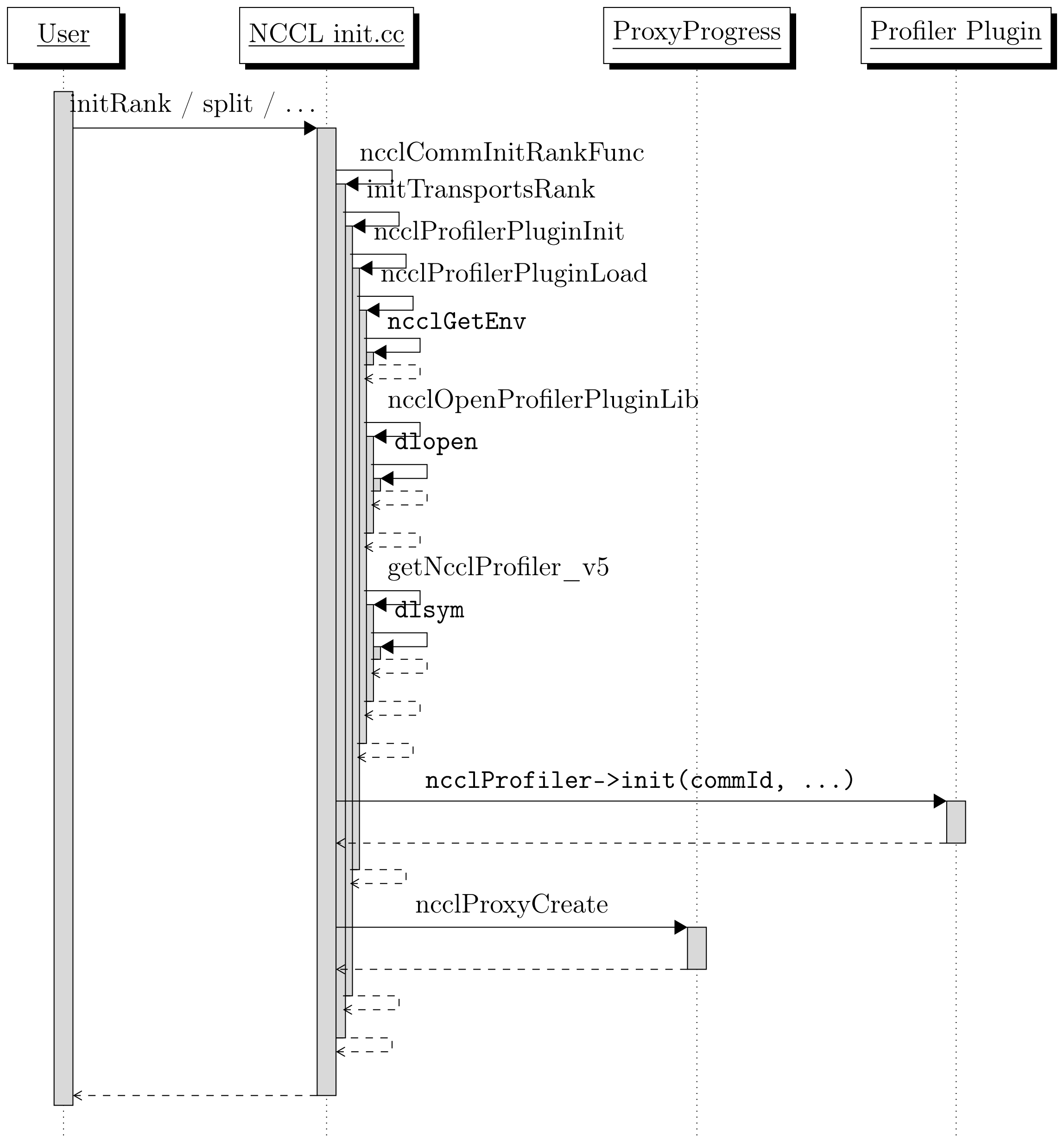


Figure 4: User API → NCCL init → load profiler plugin and call `profiler->init`.

3.2 The profiler API definition

The plugin must implement a profiler API specified by NCCL by exposing a struct. 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)(...); // NCCL calls this right after loading
  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)(...); // before unloading the plugin
};
```

The full profiler API is under **/src/include/plugin/profiler/profiler_v{versionNum}.cc**. As of NCCL v2.29.1, version 6 is the latest; five functions must be implemented. Internally NCCL wraps calls to the profiler API in custom functions (found in **/src/include/profiler.h**).

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. 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
);
```

Every time a communicator is created, `init` is called immediately upon successful plugin load in `ncclProfilerPluginLoad` (see Figure 4). If the profiler plugin `init` function does not return `ncclSuccess`, NCCL disables the plugin.

> As soon as NCCL finds the plugin and the correct ncclProfiler symbol, it calls its init function. This allows the plugin to initialize its internal context used during profiling of NCCL events.

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

`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 indicating which event types the profiler wants to track. The mapping is in **/src/include/plugin/nccl_profiler.h**; internally the mask defaults to 0 (no events). 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 v2.29.1 NCCL does not care about the return value. `void** eHandle` may point to a custom event object; this pointer is passed again in `stopEvent` and `recordEventState`. `eDescr` describes the started event. Its exact details can be found in **/src/include/plugin/profiler/**.

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 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
```

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

If the profiler enables tracking for event types lower in the hierarchy, NCCL also tracks their parent event types. The `parentObj` inside `eDescr` will be a reference to the `eHandle` of the respective parent event for the current event according to this hierarchy.

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.

NCCL ignores the return value.

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

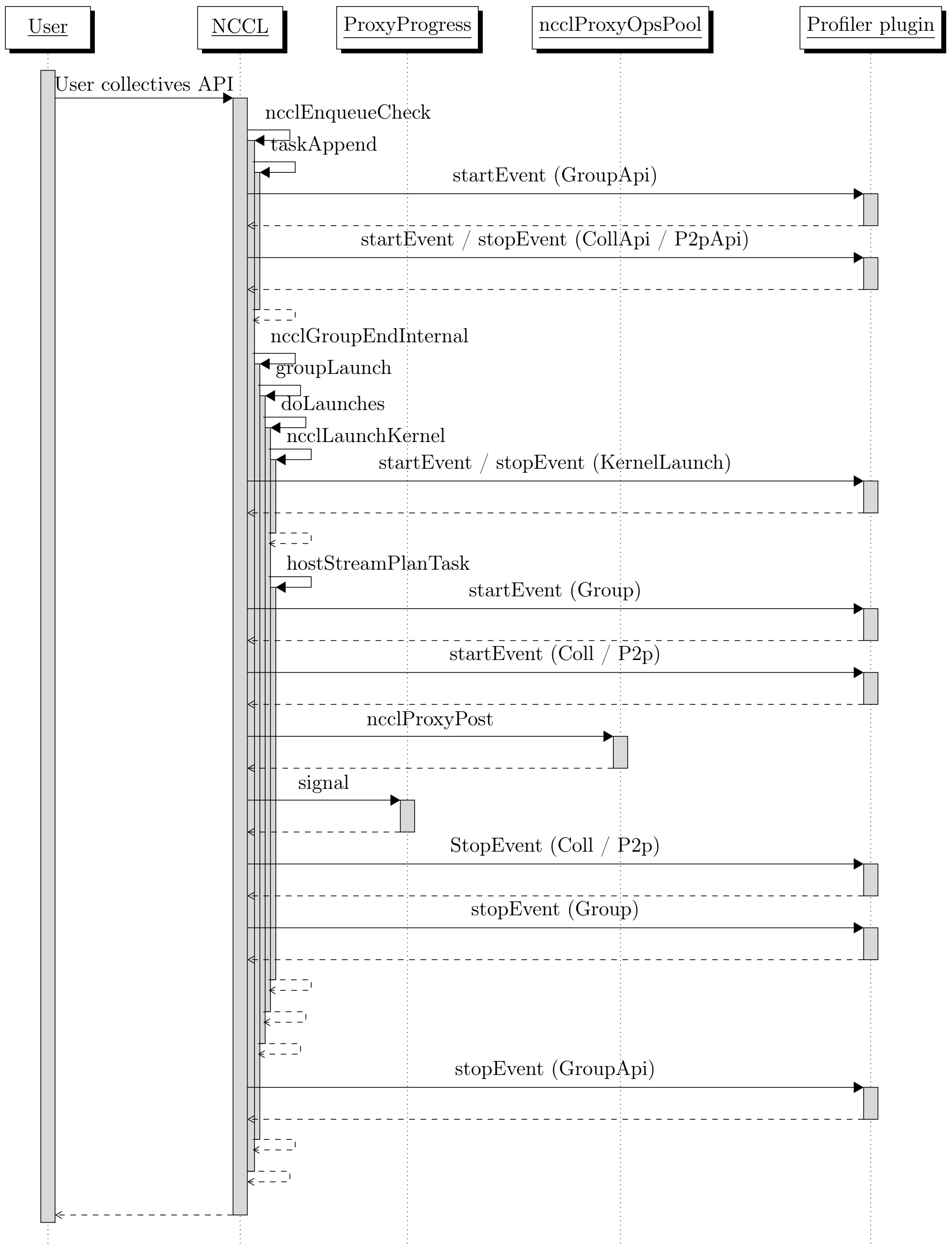


Figure 5: Flow from NCCL API calls to profiler events. In case of `ncclGroupStart / ncclGroupEnd`. multiple events of everything (except GroupApi) are called. internally, some Collectives (e.g. ncclAlltoAll) are implemented as many p2p ops, triggering many P2pApi and P2p events. Implementation: **/src/init.cc**,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

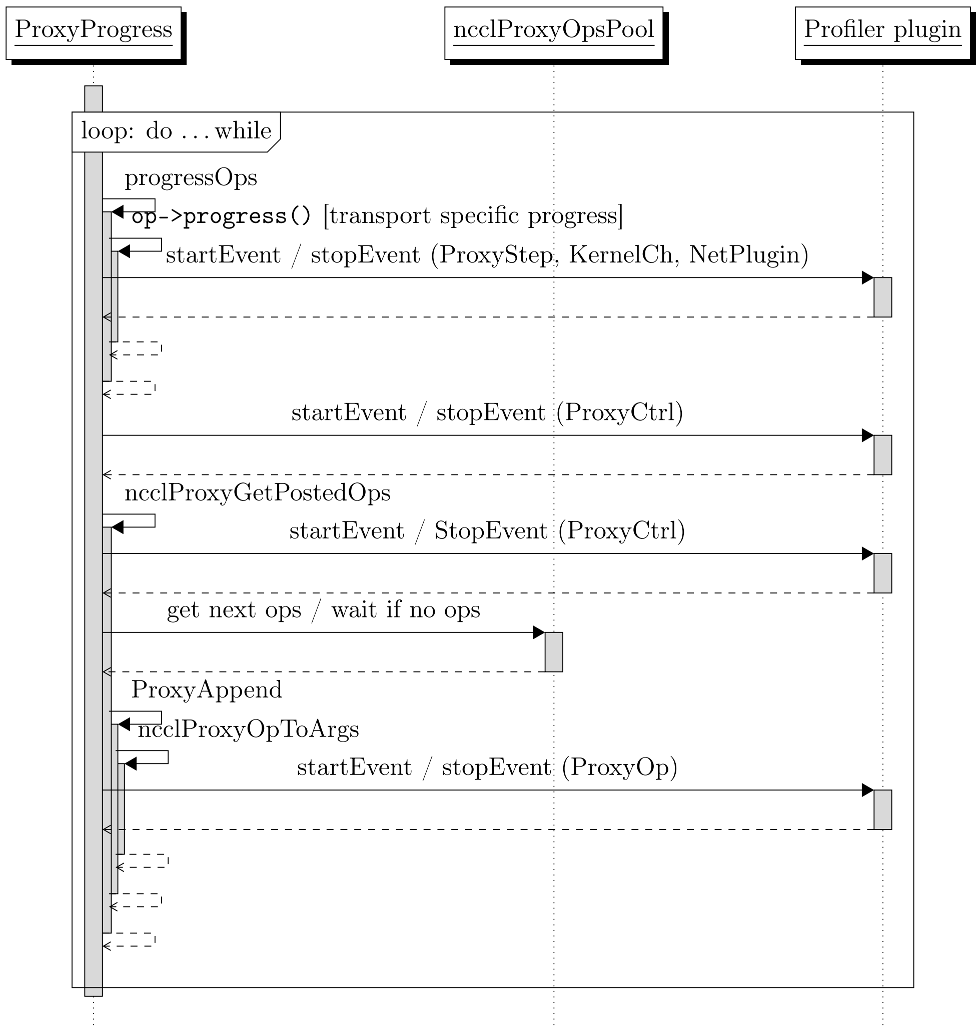

ProxyProgress
ncclProxyOpsPool
Profiler plugin
loop: do . . . while
progressOps
op->progress() [transport specific progress]
startEvent / stopEvent (ProxyStep, KernelCh, NetPlugin)
startEvent / stopEvent (ProxyCtrl)
ncclProxyGetPostedOps
startEvent / StopEvent (ProxyCtrl)
get next ops / wait if no ops
ProxyAppend
ncclProxyOpToArgs
startEvent / stopEvent (ProxyOp)

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 (defined in **/src/include/proxy.h**). 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 */
}
```

Which specific function this calls depends on the Op. This also decides which profiler events (ProxyStep, KernelCh or Network plugin specific) are started and stopped. The transport-specific progress functions are in **/src/transport/net.cc**, **coll_net.cc**, **p2p.cc**, **shm.cc**.

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). Supported states can be found under **/src/include/plugin/profiler/profiler_v{versionNum}.h**.

Called at the same sites as `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)`. Figure 7 depicts the flow from user API call to `finalize`. See implementation at **/src/init.cc**, **/src/plugin/profiler.cc**, **/src/plugin/plugin_open.cc**.

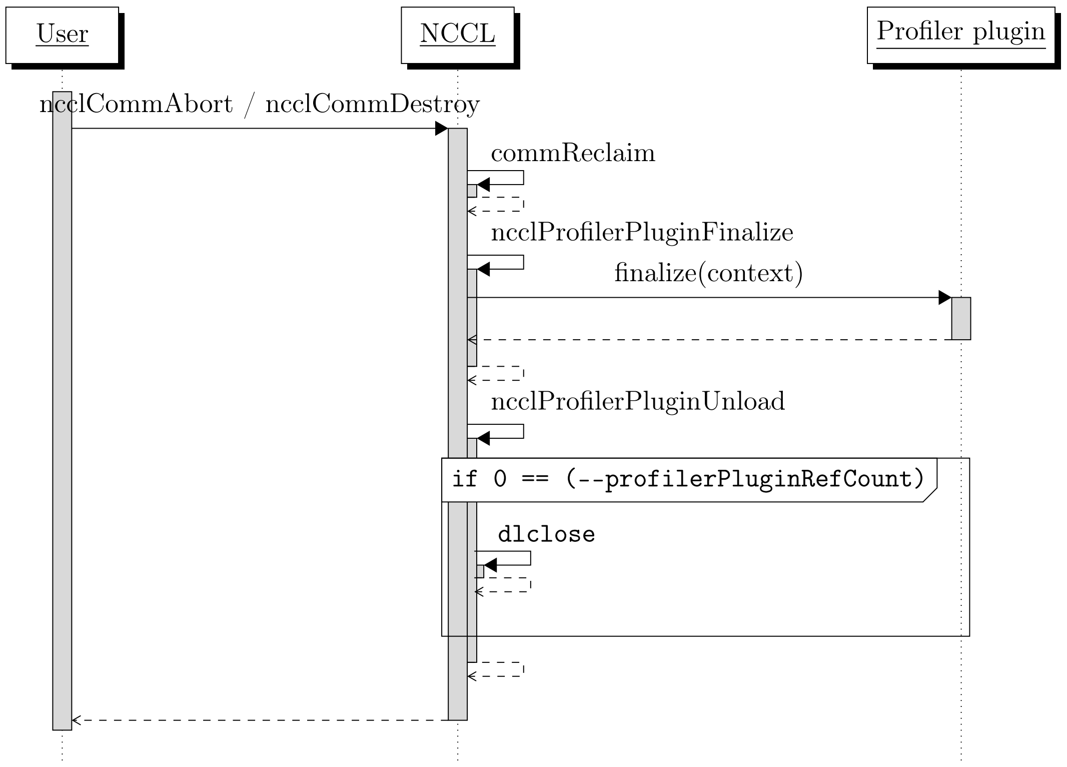

User
NCCL
Profiler plugin
ncclCommAbort / ncclCommDestroy
commReclaim
ncclProfilerPluginFinalize
finalize(context)
ncclProfilerPluginUnload
if 0 == (--profilerPluginRefCount)
dlclose

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.

3.3 Code Examples

The following examples illustrate the profiling behavior under different environment settings and user application.

- One Device per Thread
- Multiple Devices per Thread utilizing `ncclGroupStart` and `ncclGroupEnd`
- One Device per Thread and grouped collectives utilizting `ncclGroupStart` and `ncclGroupEnd`

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

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

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

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

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

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

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

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

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

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

  free(context);
  return ncclSuccess;
}

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

3.3.1 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);

// ...

ncclCommFinalize(rootComm);
ncclCommDestroy(rootComm);
```

The profiler API calls are visualized in Fig 8 and Fig 9

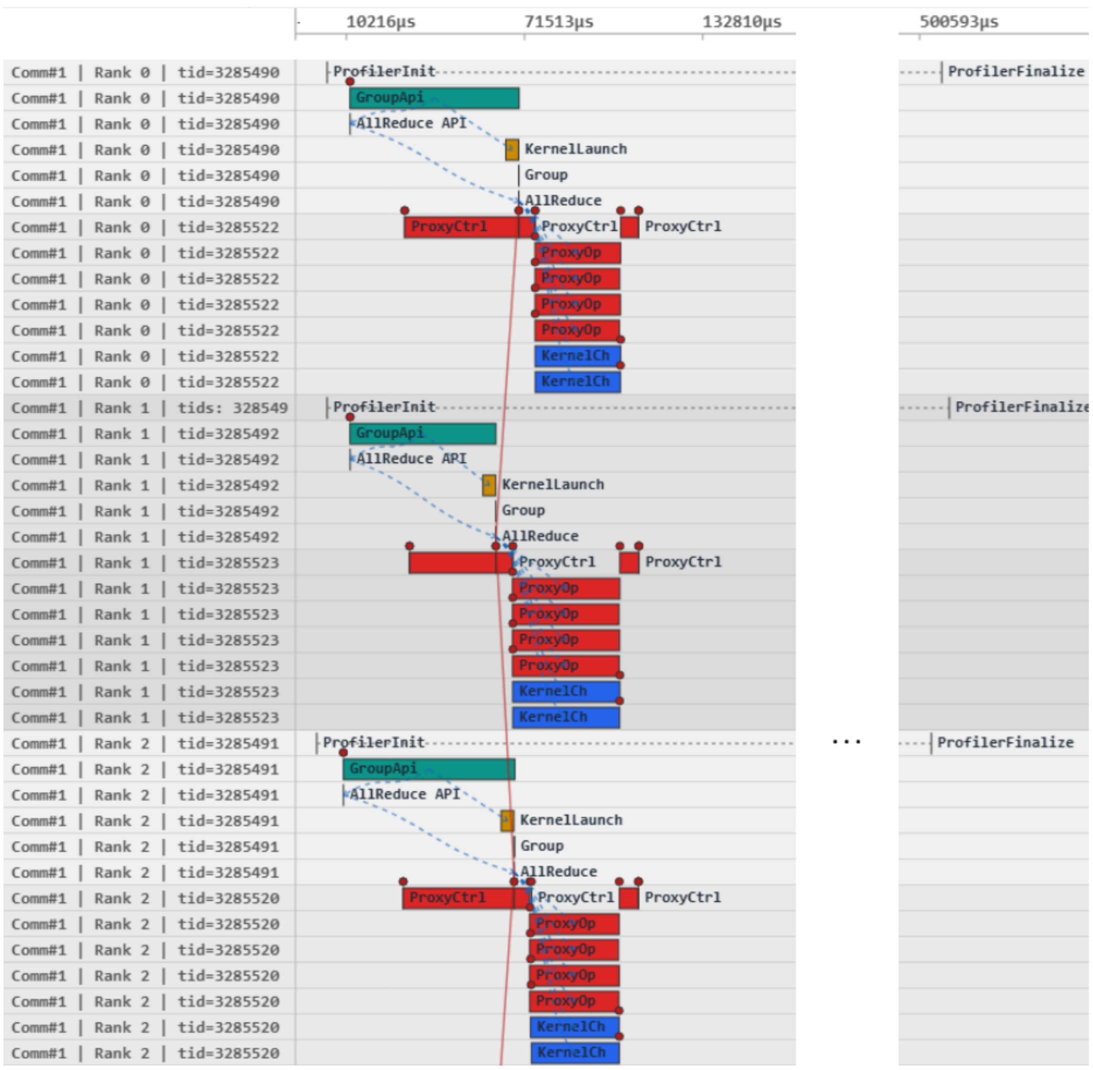

10216µs
71513µs
132810µs
500593µs
Comm#1 | Rank 0 | tid=3285490
Comm#1 | Rank 0 | tid=3285522
Comm#1 | Rank 1 | tids: 328549
Comm#1 | Rank 1 | tid=3285492
Comm#1 | Rank 1 | tid=3285523
Comm#1 | Rank 2 | tid=3285491
Comm#1 | Rank 2 | tid=3285520
ProfilerInit
GroupApi
AllReduce API
KernelLaunch
Group
AllReduce
ProxyCtrl
ProxyOp
KernelCh
ProfilerFinalize
...

Figure 8: One device per thread: First, the profiler API `init` is called for each rank. This occurs during NCCL's internal communicator creation, roughly when the application calls `ncclCommInitRank`. Then, the application calls `ncclAllReduce`. This triggers many `stateEvent`, `stopEvent`, and `recordEventState` calls to the profiler API. First, the `groupApi` event starts (green bar). 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 makes 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. Network activity evntually completes and the AllReduce collective finishes. The next `ProxyCtrl` event 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. Matching `coll` events across ranks can be identified for the same collective. This is shown by red lines connecting events with equal `seqNum` values (from `eDescr` in the profiler API `init` call) across ranks

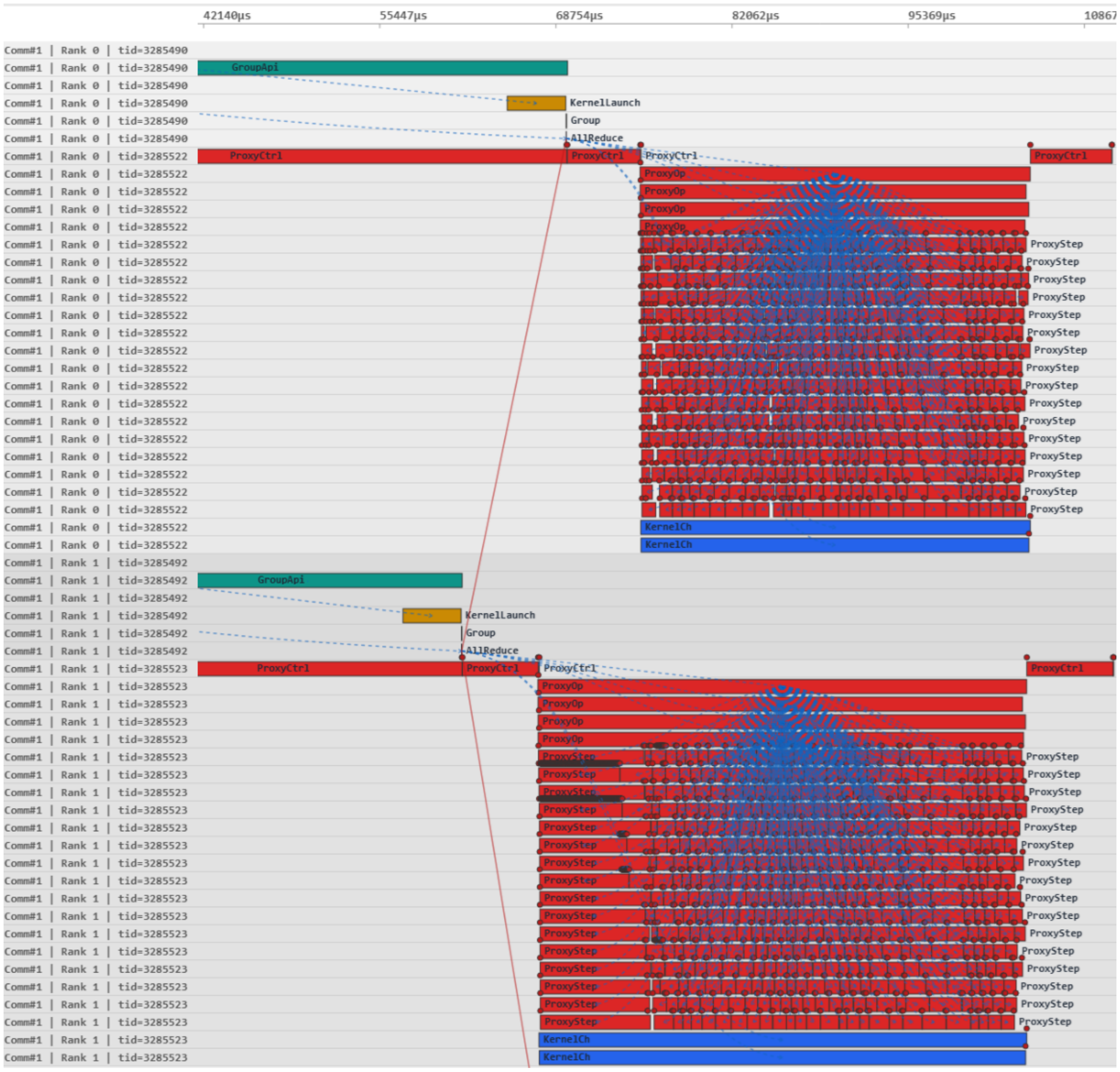

42140µs
55447µs
68754µs
82062µs
95369µs
10867
Comm#1 | Rank 0 | tid=3285490
Comm#1 | Rank 0 | tid=3285522
Comm#1 | Rank 1 | tid=3285492
Comm#1 | Rank 1 | tid=3285523
GroupApi
KernelLaunch
Group
AllReduce
ProxyCtrl
ProxyOp
ProxyStep
KernelCh

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 are done, informing about the low level network steps in their event details. The blue dotted lines indicate the `parentObj` of each proxyStep event, which are the above proxyOp events.

3.3.2 Multiple Devices per Thread (ncclGroup)

In this example, 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 all collective API calls should also be wrapped in ncclGroup.**

Here, only one collective operation (per device) is inside the ncclGroup:

```
// broadcast a commId

// ...

ncclGroupStart();
for (int i=0; i<num_gpus; i++) {
  int dev = devlist ? devlist[i] : i;
  cudaSetDevice(dev);
  ncclCommInitRankDev(comms+i, num_gpus,1, &uniqueId, i, dev, &config, __func__);
}
ncclGroupEndInternal();

// 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]);
}
```

The full example is located in /**examples**/**03_collectives**/**01_allreduce**/.

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 grouped collectives 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.

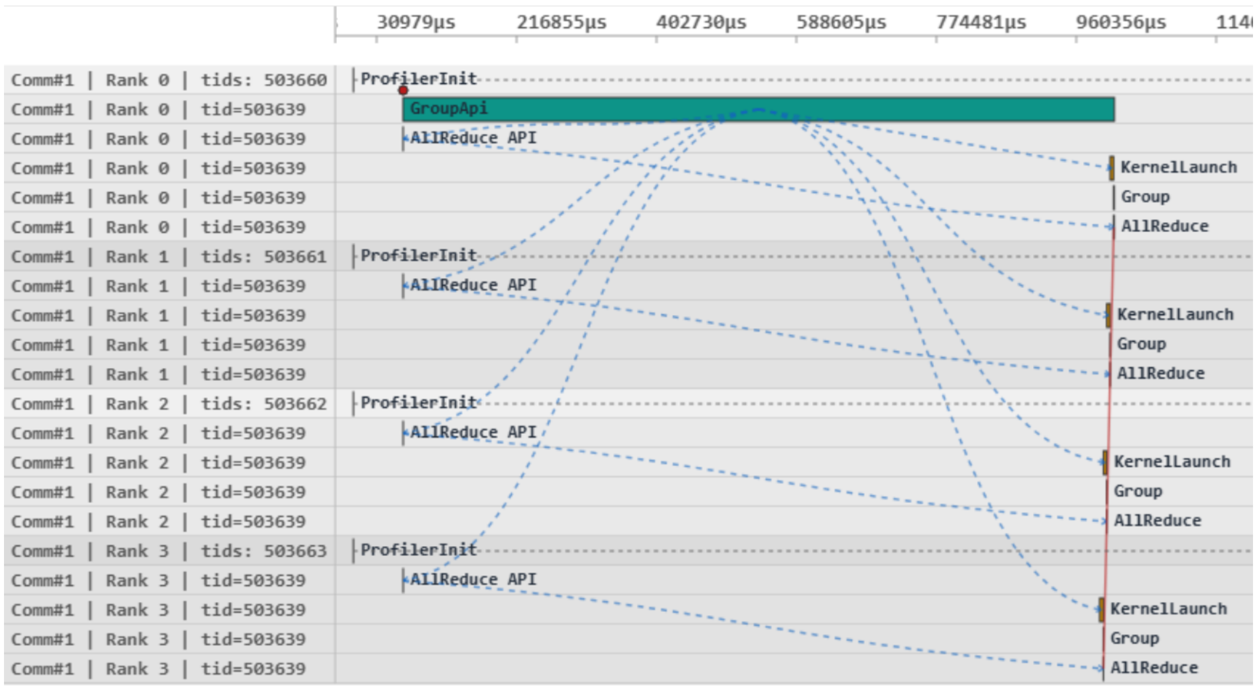

30979µs
216855µs
402730µs
588605µs
774481µs
960356µs
Comm#1 | Rank 0 | tids: 503660
Comm#1 | Rank 0 | tid=503639
Comm#1 | Rank 1 | tids: 503661
Comm#1 | Rank 1 | tid=503639
Comm#1 | Rank 2 | tids: 503662
Comm#1 | Rank 2 | tid=503639
Comm#1 | Rank 3 | tids: 503663
Comm#1 | Rank 3 | tid=503639
ProfilerInit
GroupApi
AllReduce API
KernelLaunch
Group
AllReduce

Figure 10: Multiple devices per thread: Events from the proxy thread are omitted. Grouped collectives within a thread only trigger a single `GroupApi` event

3.3.3 One Device per Thread & Grouped Collectives

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):

```
// comm init
ncclUniqueId rootId;
if (myRank == 0) { ncclGetUniqueId(&rootId); }
MPI_Bcast(&rootId, sizeof(ncclUniqueId), MPI_BYTE, 0, MPI_COMM_WORLD);
ncclComm_t rootComm;
ncclCommInitRank(&rootComm, nRanks, rootId, myRank);

// collective ops
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

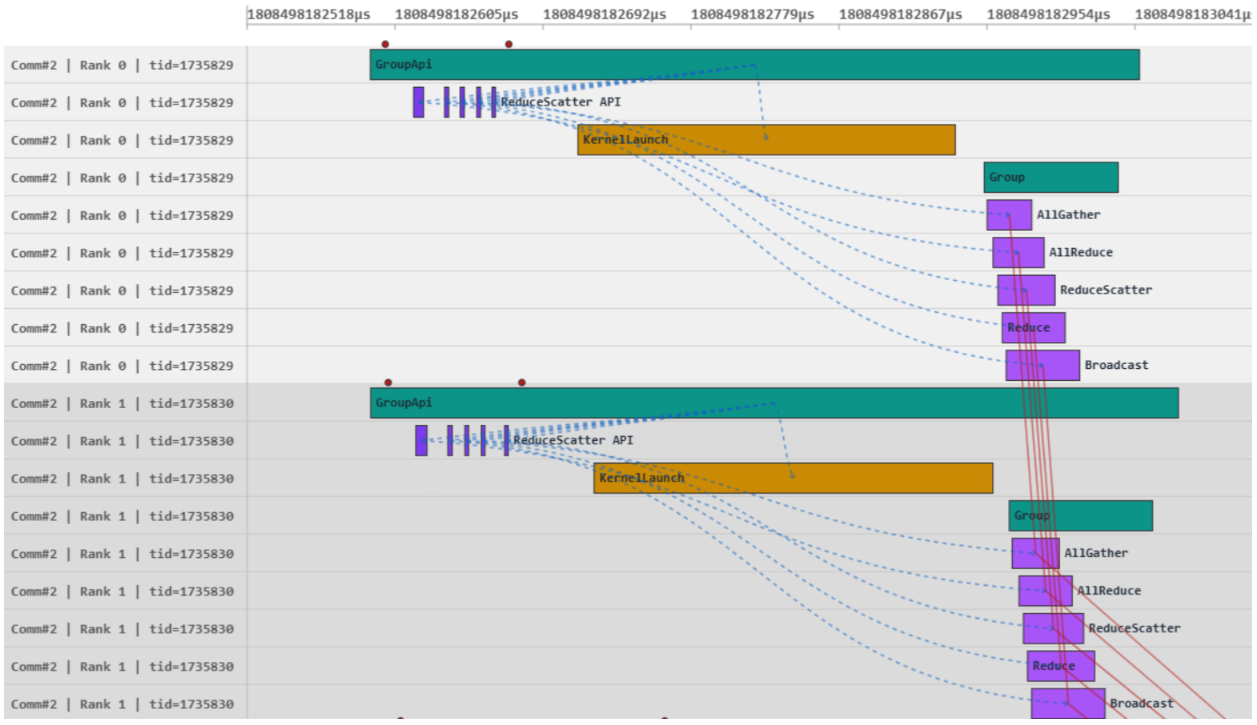


Figure 11: one GPU per thread with grouped collectives: multiple collective calls are grouped together and nccl does only a single kernel launch per thread.

3.4 Performance and scalability of the Profiler Plugin API

Experiments were run to assess the performance and scalability of profiler plugins.

The profiler used for the experiments initializes a dummy context struct, returns NULL for event handles and tracks all events 4095 (including those for network activity).

```
// 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) {
  *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[3].

The applications 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).

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

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

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.

4 Discussion

4.1 Considerations for developers of a Profiler Plugin

Logging. TODO is this even worth talking about? Code snippet: custom logging infrastructure, timestamping

example records may look like this

```
{"type":"ProfilerLifecycle","func":"ProfilerInit","gpuUuid":"785dffffffb4ffffffad-0
    fffffff87-ffff","commId":16819983883469885264,"rank":0,"start":{"ts
    ":9222912570430.104,"pid":990077,"tid":990096},"stop":{"ts":9222912570430.104,"pid
    ":990077,"tid":990096},"duration":0.0,"details":{"nranks":4,"commName":"","eventMask
    ":4095}}
{"recordType":"state","eventAddr":"0x7f6813f084d8","ts":9222912622786.445,"name":"
    EndGroupApiStart","id":23,"pid":990077,"tid":990077}
{"recordType":"event","type":"ncclProfileCollApi","gpuUuid":"785dffffffb4ffffffad-0
    fffffff87-ffff","commId":16819983883469885264,"rank":0,"start":{"ts
    ":9222912622801.756,"pid":990077,"tid":990077},"stop":{"ts":9222912622811.686,"pid
    ":990077,"tid":990077},"duration":9.930,"parentObj":"0x7f6813f084d8","eventAddr":"0
    x7f6813f08578","ctx":"0x7f6813eccfa0","details":{"func":"AllReduce","count
    ":33554432,"dtype":"ncclFloat32","root":0,"stream":"0x4bdd010","graphCaptured":false
    }}
```

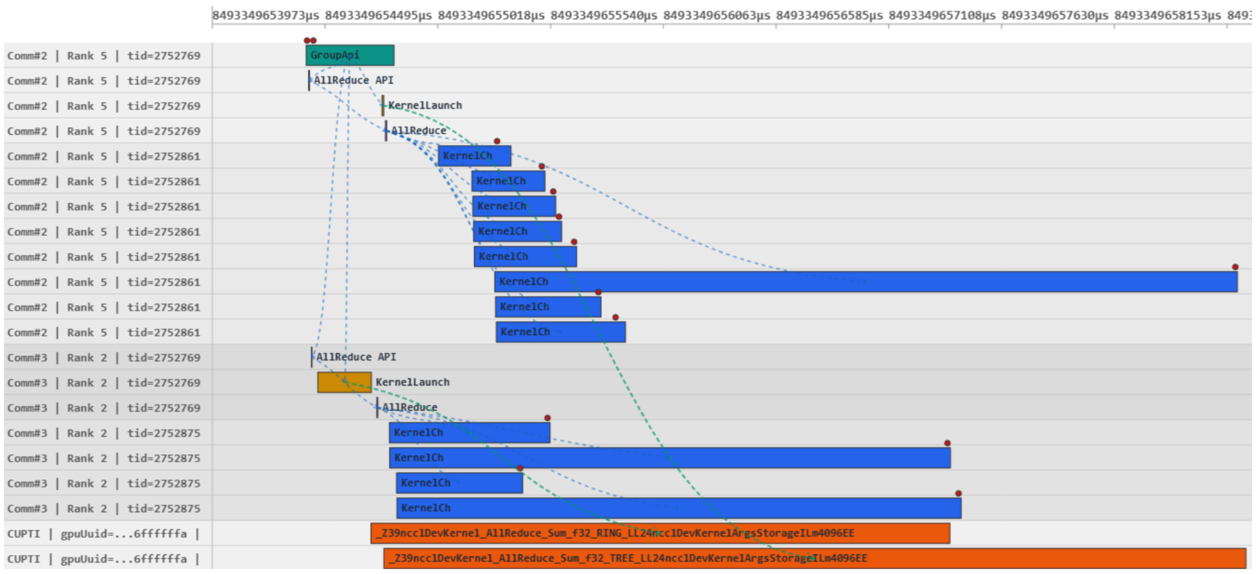
Profiler Visualization. TODO Building a visualization tool on top of a nccl profiler plugin is simple. For the purpose providing helpful visualizations in this report, such a visualizer has been developed The profiler plugin logs API calls and events as jsonl, which feed the visualization tool.

Tracking & running metrics. TODO is this even worth talking about?

> Due to the asynchronous nature of NCCL operations, events associated with collectives and point-to-point are not easy to delimit precisely. `stopEvent` for collectives simply indicates to the profiler that the collective has been enqueued. Without proxy and/or kernel activity the plugin cannot determine when a collective ends. With proxy/kernel events enabled, the plugin can estimate when it ends.

(slightly rephrased from **/ext-profiler/README.md**)

Kernel tracing with CUPTI. 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 12 provides an example visualization of this method. CUPTI can be initialized and cleaned up within the profiler plugin's own `init` and `finalize` functions.



8493349653973µs
8493349654495µs
8493349655018µs
8493349655540µs
8493349656063µs
8493349656585µs
8493349657108µs
8493349657630µs
8493349658153µs
Comm#2 | Rank 5 | tid=2752769
Comm#2 | Rank 5 | tid=2752861
Comm#3 | Rank 2 | tid=2752769
Comm#3 | Rank 2 | tid=2752875
CUPTI | gpuUuid=...6ffffffa |
GroupApi
AllReduce API
KernelLaunch
AllReduce
KernelCh
_Z39ncclDevKernel_AllReduce_Sum_f32_RING_LL24ncclDevKernelArgsStorageILm4096EE
_Z39ncclDevKernel_AllReduce_Sum_f32_TREE_LL24ncclDevKernelArgsStorageILm4096EE

Figure 12: CUPTI activity is visualized as orange event bars. Through a correlation Id, it is possible to trace the activity back to `KernelLaunch` events

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 13).

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 13).

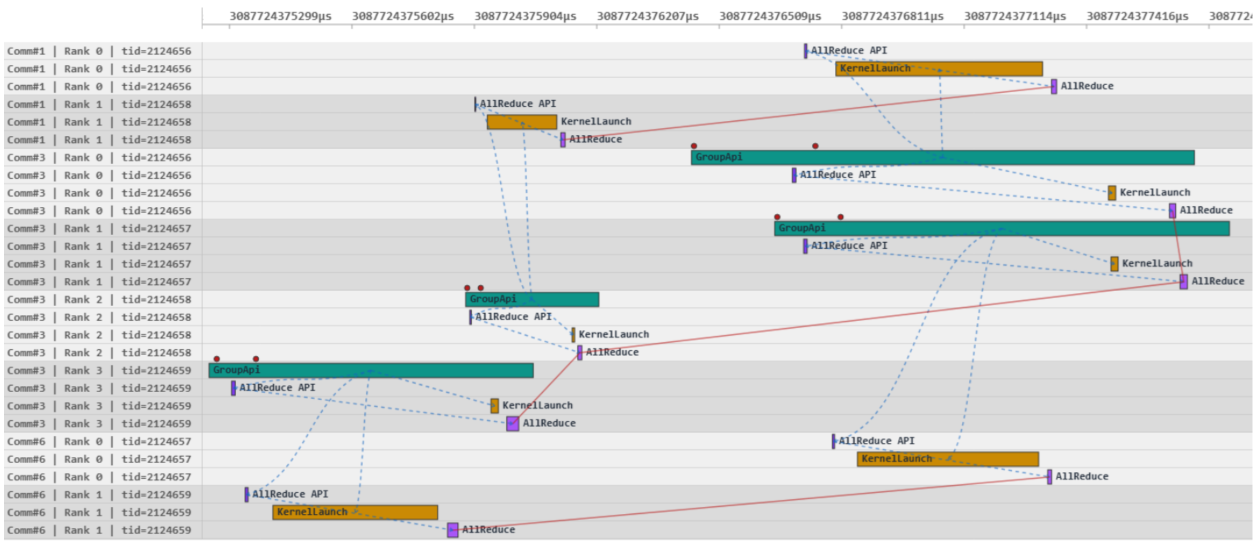


Figure 13: An example illustrating how `parentObj` and `seqNumber` can aid in understanding nccl collective operations.

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.

4.2 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**)

4.3 Potential Integration with Score-P

Score-P is a scalable measurement infrastructure for profiling, event tracing, and online analysis of parallel HPC applications[4]. It supports MPI, OpenMP, CUDA, and other paradigms, and produces OTF2 traces and CUBE4 profiles for tools such as Vampir and Scalasca. Developers can extend Score-P via plugins.

Substrate plugins are loaded as `libscorep_substrate_{name}.so` and register callbacks for Score-P's internal runtime events (region entries, MPI calls, etc.). Substrate plugins are *consumers* of events: they observe Score-P's event stream but do not inject new events.

In contrast, 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.

A potential integration strategy would involve writing a **dual-interface plugin:** A developer writes a shared library that simultaneously implements the NCCL profiler API and integrates with Score-P as a substrate plugin. Score-P loads the plugin as `libscorep_substrate_`*nccl*`.so`, while NCCL loads it via the environment variable `NCCL_PROFILER_PLUGIN`. Although substrate plugins primarily consume events, the plugin 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.

5 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

[4] `https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/`

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. A notable advantage is its low overhead: NVIDIA advertises their `inspector` implementation (**/ext-profiler/inspector**) 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.