

TECHNISCHE UNIVERSITÄT DRESDEN

DEPARTMENT OF COMPUTER SCIENCE
INSTITUTE OF COMPUTER ENGINEERING
CHAIR OF COMPUTER ARCHITECTURE
PROF. DR. WOLFGANG E. NAGEL

Hauptseminar
“Rechnerarchitektur und Programmierung”

NCCL Profiler Plugin API – A Feasibility Study

Alexander Moritz Van Le
(Mat.-No.: 4607469)

Professor: Prof. Dr. Wolfgang E. Nagel
Tutor: Bert Wesarg

Dresden, February 20, 2026

Contents

1 Abstract	2
2 Introduction to NCCL	2
2.1 Comparison to MPI	2
2.2 Relevant NCCL internals	3
3 Profiler Plugin	6
3.1 Profiler plugin loading mechanism	6
3.2 Profiler API	8
3.2.1 init	8
3.2.2 startEvent	9
3.2.3 stopEvent	11
3.2.4 recordEventState	14
3.2.5 finalize	14
3.2.6 name	15
4 Code examples and visualizations	15
4.1 One Device per Thread	17
4.1.1 Multiple Devices per Thread (ncclGroup)	19
4.1.2 Aggregated operations	21
5 Performance and scalability of the Profiler Plugin API	22
6 Discussion	24
6.1 Considerations for developers of a Profiler Plugin	24
6.2 Known limitations	26
6.3 Potential Integration with Score-P	26
7 Conclusion	27

¹ **NCCL Profiler Plugin API – A Feasibility Study**

² Alexander Moritz Van Le

³ February 20, 2026

4 **Contents**

5 **1 Abstract**

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

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

19 **2 Introduction to NCCL**

20 NCCL was first introduced by NVIDIA in 2015 at the Supercomputing Conference¹ with code being
 21 made available on GitHub². The release of NCCL 2.0 in 2017 brought support for NVLink, however
 22 this was initially only available as pre-built binaries. With the release of NCCL 2.3 in 2018, it returned
 23 to being fully open source. The NCCL Profiler Plugin API was even later introduced with NCCL 2.23
 24 in early 2025.

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

27 **2.1 Comparison to MPI**

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

- 30 • **MPI:** Communication is CPU-based. A rank corresponds to a single CPU process within a com-
 31 municator.
- 32 • **NCCL:** Communication is GPU-based, with CPU threads handling orchestration. A rank corre-

¹<https://images.nvidia.com/events/sc15/pdfs/NCCL-Woolley.pdf>

²<https://github.com/NVIDIA/nccl>

33 sponds to a GPU device within a communicator; the mapping from ranks to devices is surjective. A
 34 single CPU thread can manage multiple ranks (i.e., multiple devices) in a communicator using the
 35 functions `ncclGroupStart` and `ncclGroupEnd`. A CPU thread can also manage multiple
 36 ranks from different communicators (i.e same device allotted by multiple ranks from different com-
 37 municators) through communicator creation with `ncclCommSplit` or `ncclCommShrink`.
 38 This means the mapping from ranks to threads is also surjective.

39 **2.2 Relevant NCCL internals**

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

41 A typical NCCL application follows this basic structure:

- 42 • create nccl communicators
- 43 • allocate memory for computation and communication
- 44 • do computation and communication
- 45 • clean up nccl communicators

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

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

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

56 The ProxyProgress thread reads from this pool when calling `ncclProxyGetPostedOps` and pro-
 57 gresses the ops. See Fig. 3.

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

³<https://github.com/NVIDIA/nccl/tree/master/src/proxy.cc>

⁴<https://github.com/NVIDIA/nccl/tree/master/src/include/comm.h>

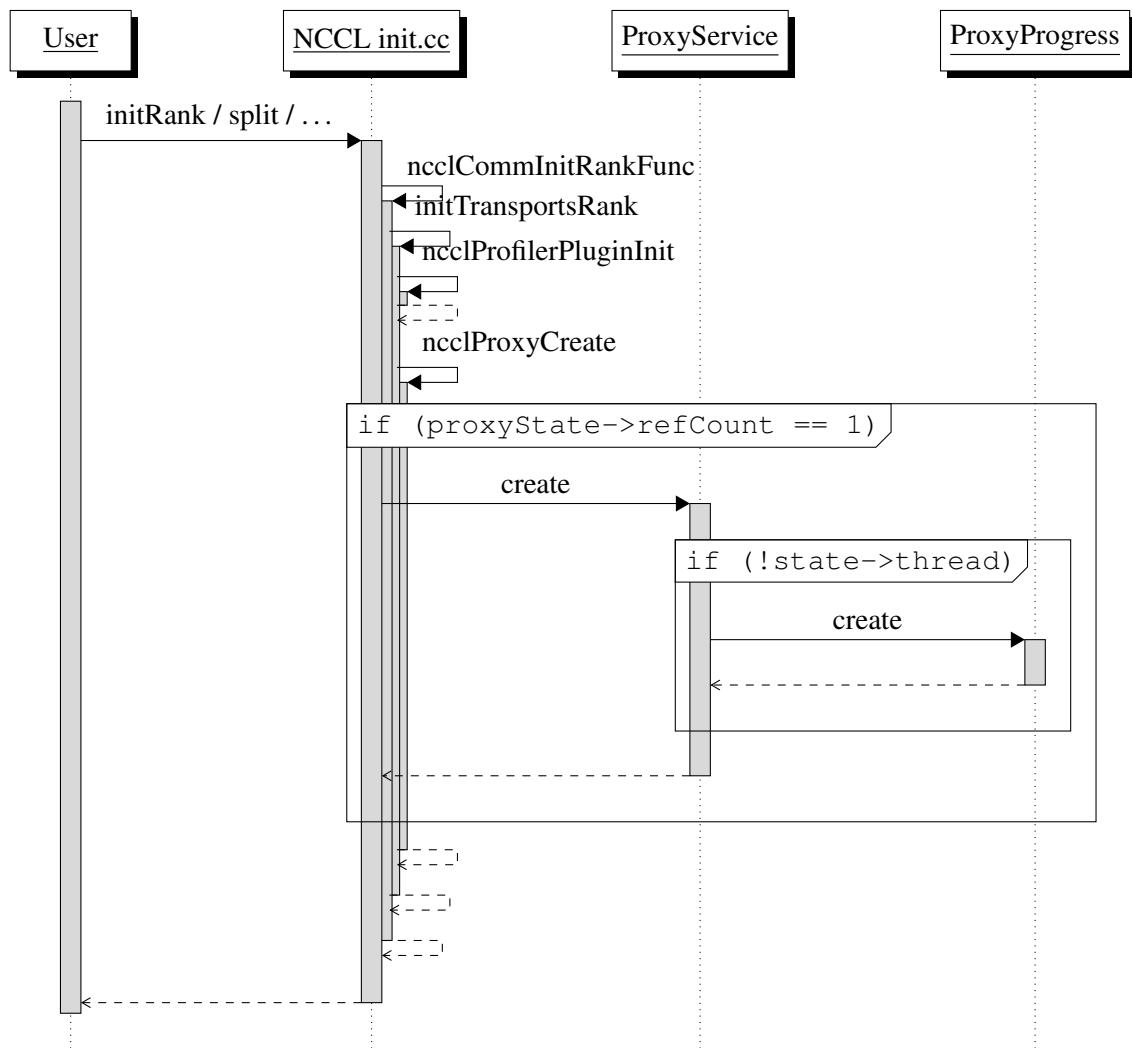


Figure 1: Thread creation: User API → NCCL internal init → create ProxyService → create ProxyProgress.

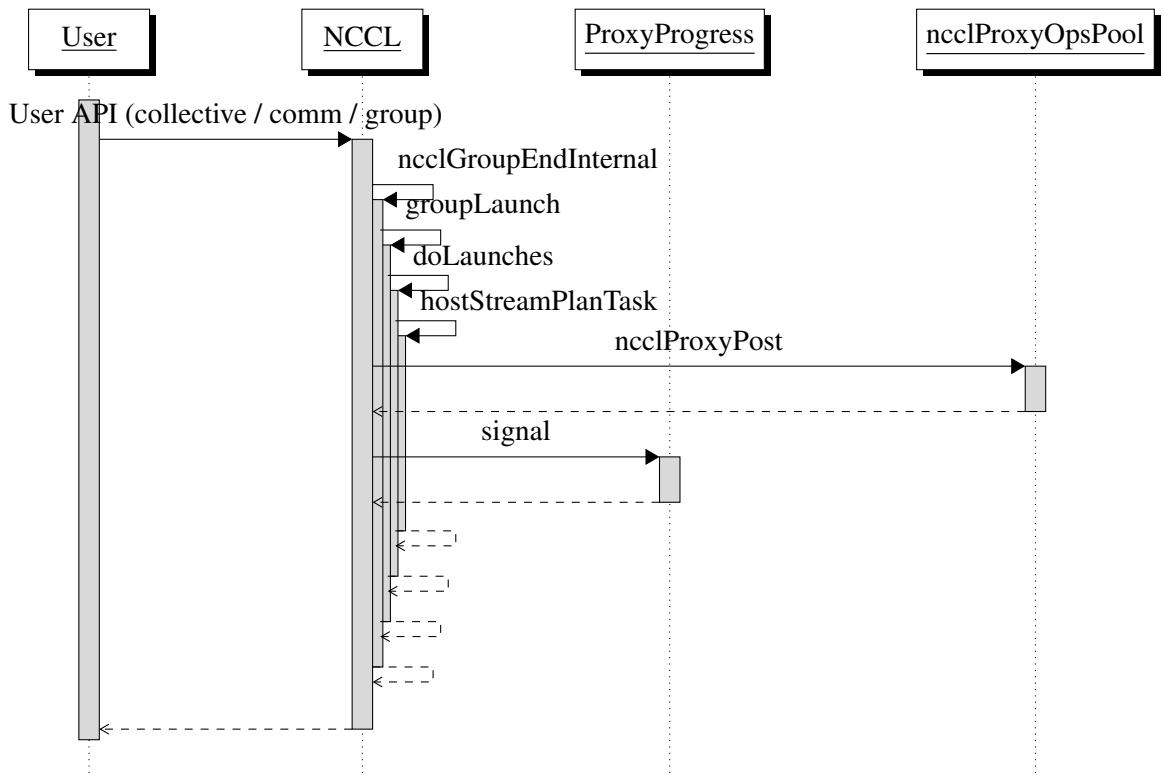


Figure 2: Flow from User API to ncclProxyPost

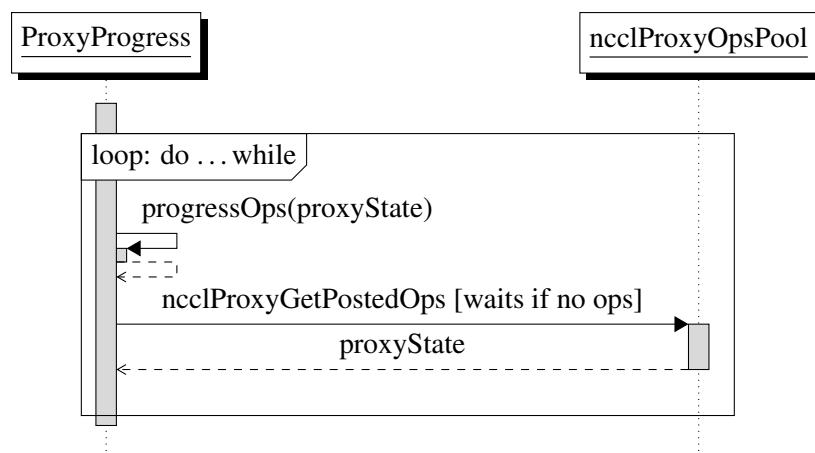


Figure 3: /src/proxy.cc ncclProxyProgress progressing loop: progress ops, then get posted ops (or wait).

60 3 Profiler Plugin

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

67 3.1 Profiler plugin loading mechanism

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

```
71
72 profilerName = ncclGetEnv("NCCL_PROFILER_PLUGIN");
73 // ...
74 handle* = dlopen(name, RTLD_NOW | RTLD_LOCAL);
75 // ...
76 ncclProfiler_v5 = (ncclProfiler_v5_t*)dlsym(handle, "ncclProfiler_v5");
```

78 If the library has already been loaded on the process, this procedure is skipped.
 79 A `profilerPluginRefCount` keeps track of the number of calls to this procedure to ensure correct
 80 unloading during finalization. See Fig. 4. The NCCL documentation⁷ also describes some further
 81 loading logic:

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

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

91 The profiler API has changed multiple times with newer NCCL releases. NCCL features a fallback

⁵<https://github.com/NVIDIA/nccl/tree/master/src/plugin/profiler.cc>

⁶https://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>

92 mechanism to load older struct versions. However one instance is known, where a profiler plugin being
 93 developed against the NCCL release 2.25.1 with Profiler API version 2, was unable to run with the latest
 94 NCCL release⁸. Around this time, the NCCL repository has undergone a refactor related to the profiler
 95 plugin.



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

⁸<https://github.com/variemai/ncclsee>

96 3.2 Profiler API

97 The plugin must implement a profiler API specified by NCCL by exposing a struct⁹. This struct should
 98 contain pointers to all functions required by the API. A plugin may expose multiple versioned structs
 99 for backwards compatibility with older NCCL versions.

```
100 ncclProfiler_v5_t ncclProfiler_v5 = {
101     const char* name;
102     ncclResult_t (*init)(...); // called when a communicator is created
103     ncclResult_t (*startEvent)(...); // at start of operations/activities
104     ncclResult_t (*stopEvent)(...); // at end of these operations/activities
105     ncclResult_t (*recordEventState)(...); // to record state of certain
106         operations
107     ncclResult_t (*finalize)(...); // called when a communicator is destroyed
108 };
109 }
```

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

115 Five functions must be implemented for the API. Internally NCCL wraps calls to the profiler API in
 116 custom functions which are all declared in a single file¹⁰.

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

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

121 3.2.1 init

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

```
126
127 ncclResult_t init(
128     void** context, // out param - opaque profiler context
129     uint64_t commId, // communicator id
130     int* eActivationMask, // out param - bitmask for which events are tracked
131     const char* commName, // user assigned communicator name
132     int nNodes, // number of nodes in communicator
```

⁹https://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>

```

133     int nranks, // number of ranks in communicator
134     int rank, // rank identifier in communicator
135     ncclDebugLogger_t logfn // logger function
136 );

```

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

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

142 The plugin developer should set `int* eActivationMask` to a bitmask¹¹, indicating which event
 143 types the profiler wants to track:

```

144
145 enum {
146     ncclProfileGroup = (1 << 0), // group event type
147     ncclProfileColl = (1 << 1), // host collective call event type
148     ncclProfileP2p = (1 << 2), // host point-to-point call event type
149     ncclProfileProxyOp = (1 << 3), // proxy operation event type
150     ncclProfileProxyStep = (1 << 4), // proxy step event type
151     ncclProfileProxyCtrl = (1 << 5), // proxy control event type
152     ncclProfileKernelCh = (1 << 6), // kernel channel event type
153     ncclProfileNetPlugin = (1 << 7), // network plugin-defined, events
154     ncclProfileGroupApi = (1 << 8), // Group API events
155     ncclProfileCollApi = (1 << 9), // Collective API events
156     ncclProfileP2pApi = (1 << 10), // Point-to-Point API events
157     ncclProfileKernelLaunch = (1 << 11), // Kernel launch events
158 };

```

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

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

168 3.2.2 startEvent

169 `startEvent` is called when NCCL begins certain operations:

```

170
171 ncclResult_t startEvent(
172     void* context, // opaque profiler context object

```

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

```

173     void** eHandle, // out param - event handle
174     ncclProfilerEventDescr_v5_t* eDescr // pointer to event descriptor
175 );

```

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

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

182 All User API calls to Collective or P2P operations will start a Group API event. When networking is
 183 required, ProxyCtrl Events may be emitted. Depending on the `eActivationMask` bitmask returned
 184 in the `init` function, further (child) events will be emitted in deeper regions of the nccl code base. It
 185 can be thought of as an event hierarchy¹³ with several depth levels:

```

186
187     Group API event
188     |
189     +- Collective API event
190     |   |
191     |   +- Collective event
192     |   |
193     |   +- ProxyOp event
194     |   |   |
195     |   |   +- ProxyStep event
196     |   |   |
197     |   |   +- NetPlugin event
198     |   |
199     |   +- KernelCh event
200     |
201     +- Point-to-point API event
202     |   |
203     |   +- Point-to-point event
204     |   |
205     |   +- ProxyOp event
206     |   |   |
207     |   |   +- ProxyStep event
208     |   |   |
209     |   |   +- NetPlugin event
210     |   |
211     |   +- KernelCh event
212     |
213     +- Kernel Launch event
214
215     ProxyCtrl event

```

217 The `parentObj` inside `eDescr` will be a reference to the `eHandle` of the respective parent event

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

¹³<https://github.com/NVIDIA/nccl/tree/master/ext-profiler/README.md>

218 for the current event according to this hierarchy. Thus, if the eActivationMask set during init
219 enables tracking for event types lower in the hierarchy, NCCL always also tracks their parent event types.

220 **3.2.3 stopEvent**

221 ncclResult_t stopEvent (void* eHandle); // handle to event object
222

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

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

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

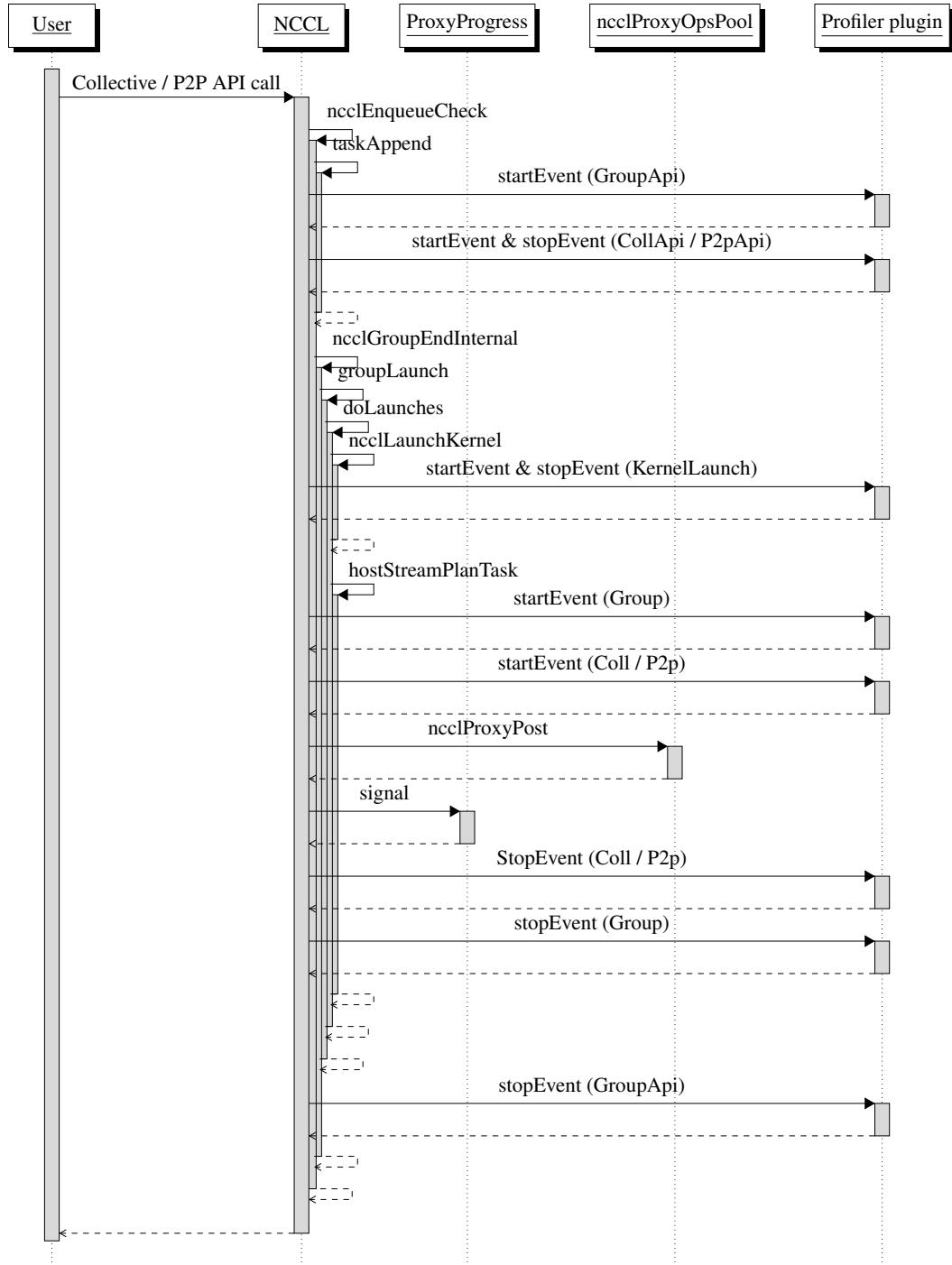


Figure 5: Flow from NCCL API calls to profiler events. In case of `ncclGroupStart` / `ncclGroupEnd`, multiple events of everything (except GroupApi) are called. Internally, some Collectives (e.g. `ncclAlltoAll`) are implemented as multiple p2p ops, triggering many P2pApi and P2p events.

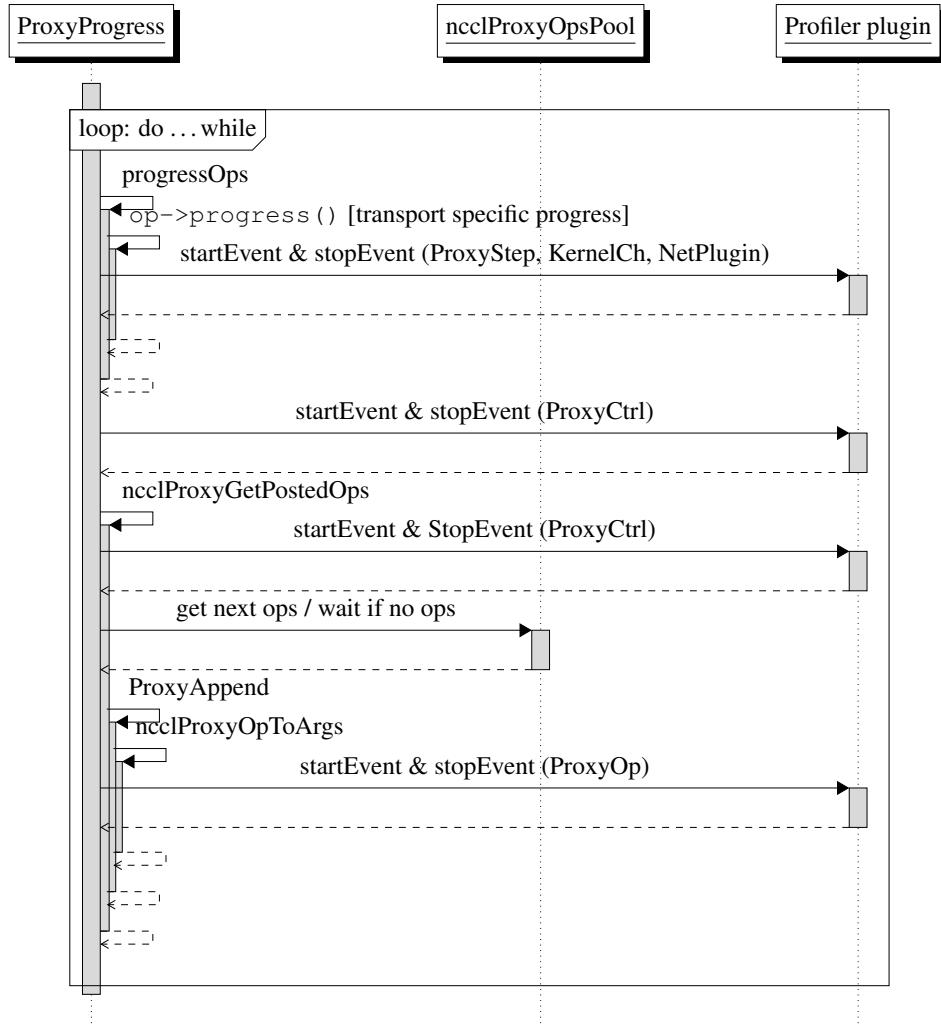


Figure 6: ncclProxyProgress: progressOps emits ProxyStep/KernelCh/NetPlugin events. getPostedOps emits ProxyOp events. Several events ProxyCtrl are also emitted

230 op->progress () progresses transport specific ops. This is implemented as a function pointer type¹⁴.
 231 Confusingly the variable is called ‘op’, although its type is ncclProxyArgs and *not* ncclProxyOp.

```

232
233     typedef ncclResult_t (*proxyProgressFunc_t) (struct ncclProxyState*, struct
234         ncclProxyArgs* );
235
236     struct ncclProxyArgs {
237         proxyProgressFunc_t progress;
238         struct ncclProxyArgs* next;
239         /* other fields */
240     }
  
```

242 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>

243 methods¹⁵¹⁶¹⁷¹⁸. Each implementations calls the profiler API to inform about a different event type
 244 (ProxyStep, KernelCh or Network plugin specific).

245 **3.2.4 recordEventState**

```
246 ncclResult_t recordEventState(
247     void* eHandle,
248     ncclProfilerEventState_v5_t eState,
249     ncclProfilerEventStateArgs_v5_t* eStateArgs
250 );
251 );
```

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

256 **3.2.5 finalize**

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

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

¹⁵<https://github.com/NVIDIA/nccl/tree/master/src/transport/net.cc>

¹⁶https://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>

¹⁹https://github.com/NVIDIA/nccl/tree/master/src/include/plugin/profiler/profiler_v5.h

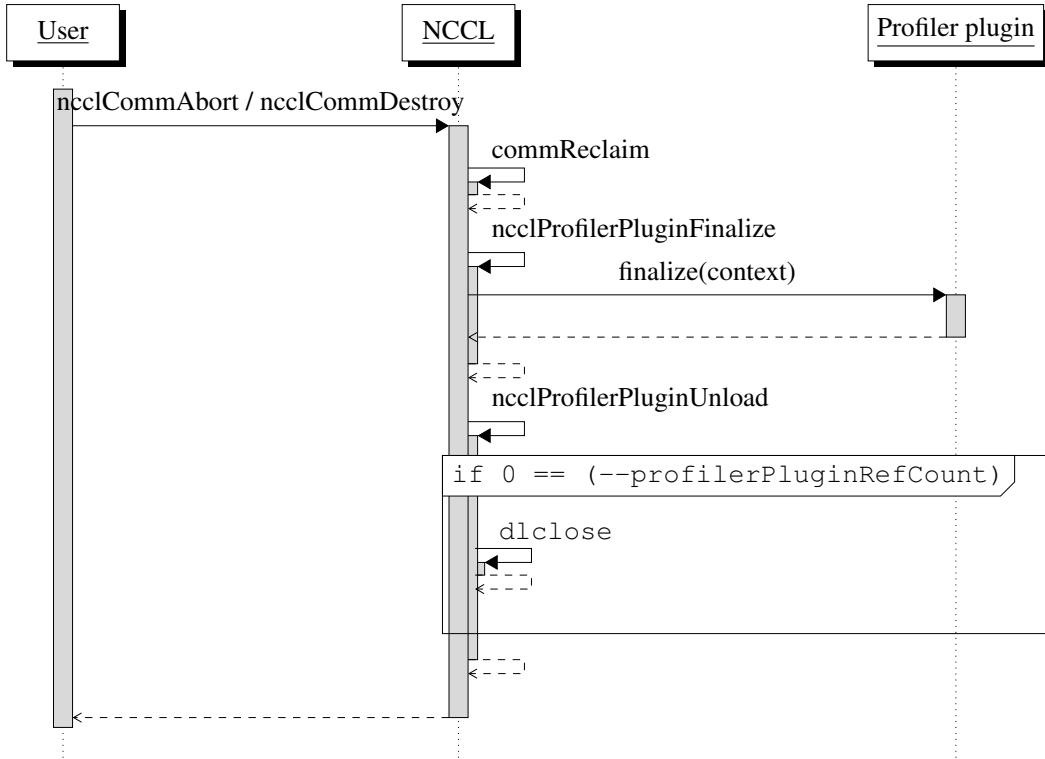


Figure 7: User API → commReclaim → finalize → plugin unload.

264 3.2.6 name

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

268 4 Code examples and visualizations

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

- 270 • One Device per Thread
- 271 • Multiple Devices per Thread via `ncclGroupStart` and `ncclGroupEnd`
- 272 • One Device per Thread and aggregated operations via `ncclGroupStart` and `ncclGroupEnd`

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

```

275
276 struct MyContext { /* custom context struct */ };
277 struct MyEvent { /* custom event struct */ };
278
279 MyEvent* allocEvent(args) { /* handles event allocation */ }

```

```
280 uint64_t getTime() { /* gets time */ }
281 void writeJsonl() { /* writes call details to process specific log file as
282     structured jsonl */ }
283
284 ncclResult_t myInit( /* args - **context, *eActivationMask, ... */ ) {
285     *context = malloc(sizeof(struct MyContext));
286     *eActivationMask = 4095; /* enable ALL event types */
287
288     writeJsonl(getTime(), "Init", args);
289     return ncclSuccess;
290 }
291
292 ncclResult_t myStartEvent( /* args - **eHandle, ... */ ) {
293     *eHandle = allocEvent(args);
294
295     writeJsonl(getTime(), "StartEvent", args);
296     return ncclSuccess;
297 }
298
299 ncclResult_t myStopEvent(void* eHandle) {
300     writeJsonl(getTime(), "StopEvent", eHandle);
301
302     free(eHandle)
303     return ncclSuccess;
304 }
305
306 ncclResult_t myRecordEventState( /* args - ... */ ) {
307     writeJsonl(getTime(), "RecordEventState", args);
308     return ncclSuccess;
309 }
310
311 ncclResult_t myFinalize(void* context) {
312     writeJsonl(getTime(), "Finalize", args);
313
314     free(context);
315     return ncclSuccess;
316 }
317
318 ncclProfiler_v5_t ncclProfiler_v5 = {
319     "MyProfilerPlugin",
320     myInit,
321     myStartEvent,
322     myStopEvent,
323     myRecordEventState,
324     myFinalize,
325 };
```

327 Alongside the logging profiler plugin, a visualization tool has been built, that ingests the profiler logs
 328 to inspect the exact behavior of internal calls from NCCL to the Profiler API. It displays the events
 329 as colored bars on a timeline and separates them on different lanes. Each lane also displays some
 330 information about the communicator, rank and thread corresponding to the event. Additionally, blue
 331 dotted lines indicate the relationship between events according to the `parentObj` field and red lines
 332 indicate which collective events belong to the same collective operation.
 333 Further, a hover feature was added to inspect all details of an event, however this feature is not used in
 334 the following illustrative examples.

335 **4.1 One Device per Thread**

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

```
340
341 // broadcast a commId
342
343 // ...
344
345 ncclCommInitRank(&rootComm, nRanks, commId, myRank);
346
347 // ...
348
349 ncclAllReduce(sendBuff, recvBuff, BUFFER_SIZE, ncclFloat, ncclSum, rootComm
350     , streams);
351
352 // ...
353
354 ncclCommDestroy(rootComm);
355
```

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

358 First, the profiler API `init` is called for each rank. This occurs during NCCL's internal communicator
 359 creation, when the application calls `ncclCommInitRank`. After the application calls `ncclAllReduce`,
 360 many Profiler API calls to `stateEvent`, `stopEvent`, and `recordEventState` are triggered: In-
 361 tially, `startEvent` for the `groupApi` (green bar) is called. Below it, the `startEvent` and soon the `stopEvent`
 362 for the AllReduce `collApi` event are called. The yellow bar shows when NCCL enqueues the GPU
 363 kernel launch (`KernelLaunch` event). The two bars below represent the `group` and `coll` events.
 364 NCCL also spawns a proxy progress thread per rank, which does additional profiler API calls. The first
 365 red `ProxyCtrl` event shows the proxy progress thread was asleep. Next, a new `ProxyCtrl` event
 366 shows time for the proxy thread to append proxy ops. Then, appended ops start progressing (`ProxyOps`
 367 events), which in `op->progress()` starts `ProxyStep` and `KernelCh` events that inform about

368 low level network activity in updates via `recordEventState` like `ProxyStepRecvGPUWait`
 369 (see Fig. 9). Network activity eventually completes and the AllReduce collective finishes. The next
 370 ProxyCtrl event only shows the proxy thread sleeping again. Finally, profiler `finalize` is called,
 371 which happens when the application cleans up NCCL communicators and no further communicators are
 372 tracked in the profiler in each respective thread.

373 ProxyStep events are emitted in cross node communication environments. If this type of communica-
 374 tion 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.

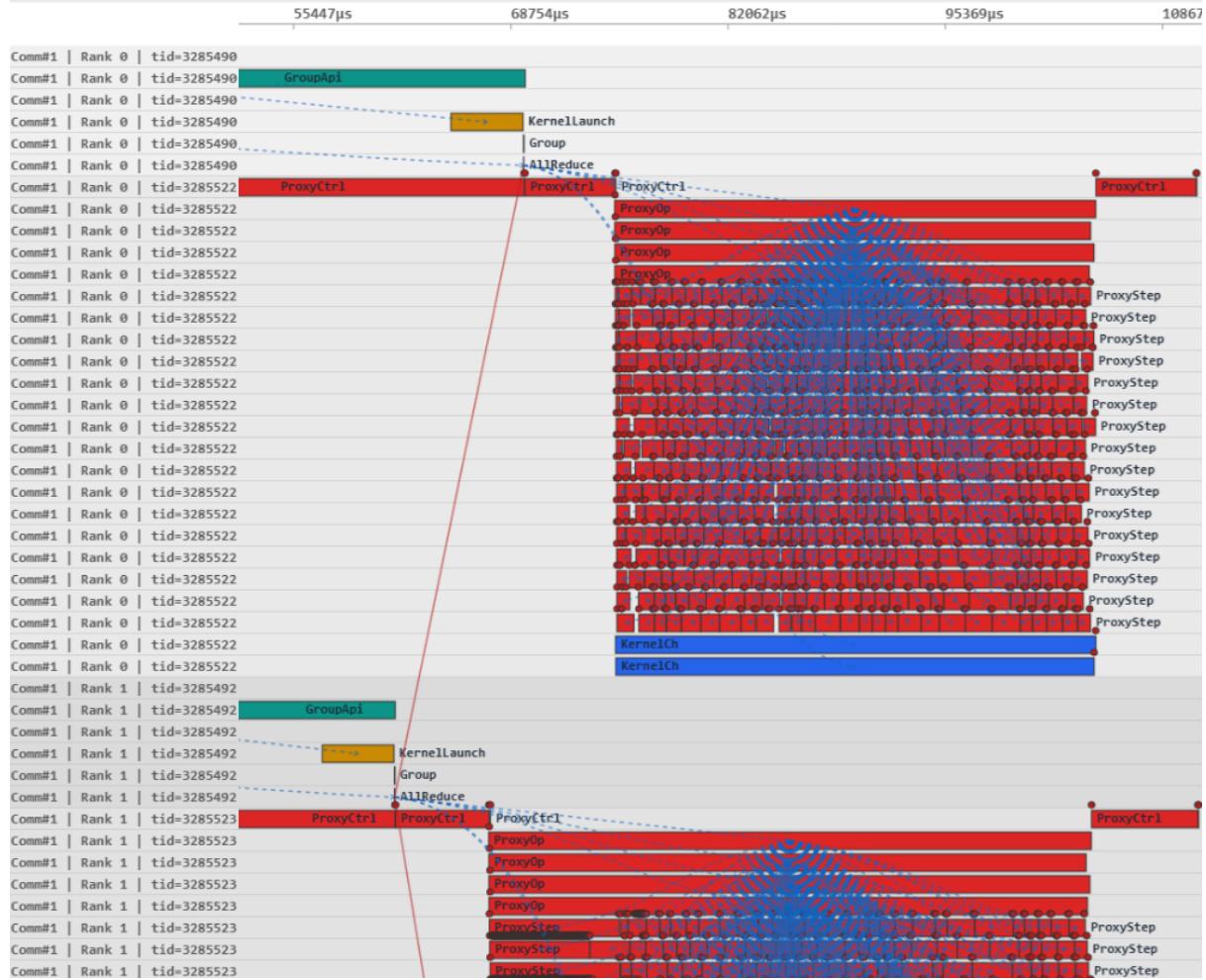


Figure 9: One device per thread: In Fig. 8 ProxyStep events have been omitted for visual clarity. However, in multinode settings, many additional profiler API calls for proxyStep events happen, informing about the low level network steps in their event details via recordEventState (indicated as red circles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

375 4.1.1 Multiple Devices per Thread (ncclGroup)

376 In this example²⁰, one NCCL thread manages all GPUs on the same node. This is achieved by wrapping
 377 communication initialization in ncclGroupStart and ncclGroupEnd for each managed GPU. In
 378 this orchestration setting, **NVIDIA’s documentation states that collective API calls should also be**
 379 **wrapped in ncclGroup**. Here, only one collective operation (per device) is inside the ncclGroup:

```

380
381 // broadcast a commId
382
383 // ...
384
  
```

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

```

385 ncclGroupStart();
386 for (int i=0; i<nGPUs; i++) {
387   cudaSetDevice(dev);
388   ncclCommInitRank(comms+i, nGPUs*nRanks, id, myRank*nGPUs+i);
389 }
390 ncclGroupEnd();
391
392 // alternatively to above method, NCCL provides the convenience function
393 // ncclCommInitAll();
394
395 // ...
396
397 ncclGroupStart();
398 for (int i = 0; i < num_gpus; i++) {
399   ncclAllReduce( /* ... */ );
400 }
401 ncclGroupEnd();
402
403 // ...
404
405 for (int i = 0; i < num_gpus; i++) {
406   ncclCommDestroy(comms[i]);
407 }

```

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

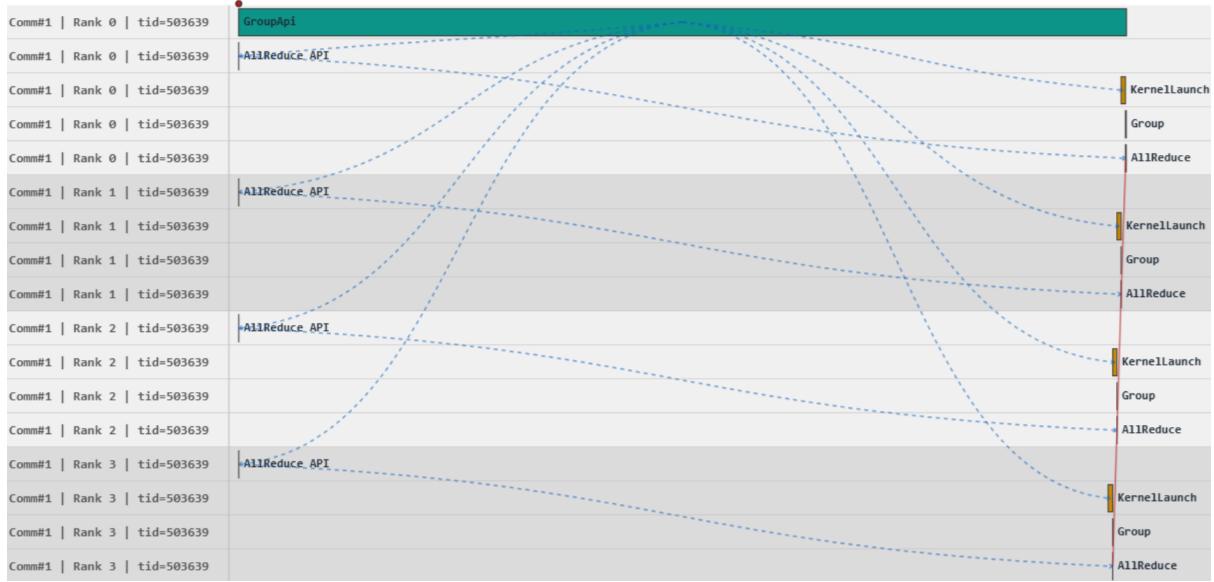


Figure 10: Multiple devices per thread: Events from the proxy thread as well as init and finalize calls are omitted. Collective API calls from multiple GPUs managed by a single thread only trigger a single GroupApi event.

414 4.1.2 Aggregated operations

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

```

417
418 // broadcast a commId
419
420 // ...
421
422 ncclCommInitRank(&rootComm, nRanks, rootId, myRank);
423
424 // ...
425
426 ncclGroupStart();
427 ncclAllReduce( /* ... */ );
428 ncclBroadcast( /* ... */ );
429 ncclReduce( /* ... */ );
430 ncclAllGather( /* ... */ );
431 ncclReduceScatter( /* ... */ );
432 ncclGroupEnd();
433
434 // ...
435
  
```

436 The behavior changes can be described as follow:

- 437 • single GroupApi event per thread
 438 • single KernelLaunch event per thread
 439 • single Group event per thread

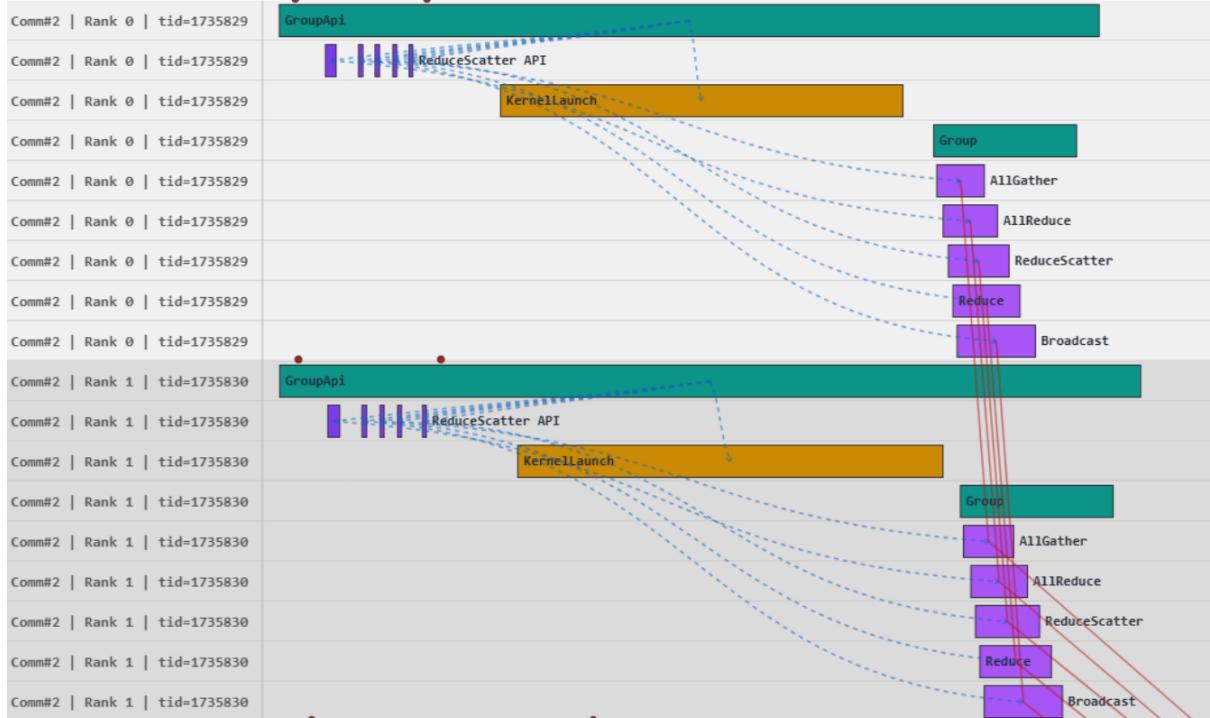


Figure 11: one GPU per thread with aggregated operations: multiple collective calls are grouped together and nccl does only a single kernel launch per thread.

440 5 Performance and scalability of the Profiler Plugin API

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

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

```
447
448 // an 'empty' NCCL Profiler Plugin
449
450 struct MyContext {
451     char dummy;
452 };
453
454 ncclResult_t myInit(void** context, uint64_t commId, int* eActivationMask,
```

```

455     const char* commName, int nNodes, int nranks, int rank,
456     ncclDebugLogger_t logfn) {
457     *context = malloc(sizeof(struct MyContext));
458     *eActivationMask = 4095; /* enable ALL event types */
459     return ncclSuccess;
460 }
461
462 ncclResult_t myStartEvent(void* context, void** eHandle,
463     ncclProfilerEventDescr_v5_t* eDescr) {
464     *eHandle = NULL;
465     return ncclSuccess;
466 }
467
468 ncclResult_t myStopEvent(void* eHandle) {
469     return ncclSuccess;
470 }
471
472 ncclResult_t myRecordEventState(void* eHandle, ncclProfilerEventState_v5_t
473     eState, ncclProfilerEventStateArgs_v5_t* eStateArgs) {
474     return ncclSuccess;
475 }
476
477 ncclResult_t myFinalize(void* context) {
478     free(context);
479     return ncclSuccess;
480 }
481
482 ncclProfiler_v5_t ncclProfiler_v5 = {
483     "EmptyProfiler",
484     myInit,
485     myStartEvent,
486     myStopEvent,
487     myRecordEventState,
488     myFinalize,
489 };

```

491 For testing the performance overhead in collective and P2P operations, **nccl-tests** from NVIDIA was
492 used²¹.

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

498 The Table 1 shows the average latency per operation (time in μ s) across iterations. The empty profiler

²¹<https://github.com/NVIDIA/nccl-tests>

499 adds roughly 8 to 9 μs overhead per operation in single-node runs (4 GPUs), but introduces negligible
500 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

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

503 6 Discussion

504 This section first discusses practical considerations for developers who implement or extend an NCCL
505 profiler plugin, as well as known limitations of the current profiling infrastructure, and then shows how
506 the plugin could be integrated with the Score-P measurement infrastructure for HPC-wide tracing and
507 analysis.

508 6.1 Considerations for developers of a Profiler Plugin

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

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

```
515 /src/include/comm.h
516
517 struct ncclComm {
518     uint64_t seqNumber[NCCL_NUM_FUNCTIONS];
519     /* other fields */
520 }
```

522 /src/plugin/profiler.cc

```
523
524 ncclResult_t ncclProfilerStartTaskEvents(struct ncclKernelPlan* plan) {
```

```

525  /* other code */
526  __atomic_fetch_add(&plan->comm->seqNumber[ct->func], 1, __ATOMIC_RELAXED)
527  ;
528  /* other code */
529 }
530

```

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

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

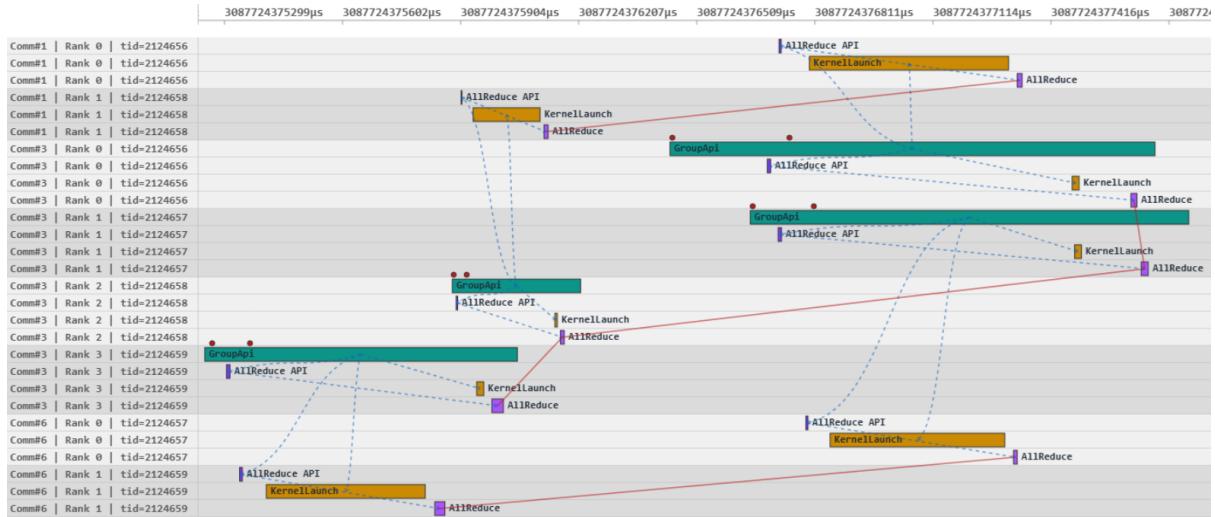


Figure 12: An example illustrating how `parentObj` and `seqNumber` can be used to better understand the timing of concurrent collective operations.

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

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

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

555 **6.2 Known limitations**

556 Kernel event instrumentation uses counters exposed by the kernel to the host and the proxy progress
 557 thread. Thus the proxy progress thread infrastructure is shared between network and profiler. If the
 558 proxy is serving network requests, reading kernel profiling data can be delayed, causing loss of accuracy.
 559 Similarly, under heavy CPU load and delayed scheduling of the proxy progress thread, accuracy can be
 560 lost.
 561 From profiler version 4, NCCL uses a per-channel ring buffer of 64 elements. Each counter is comple-
 562 mented by two timestamps (ptimers) supplied by the NCCL kernel (start and stop of the operation in
 563 the kernel). NCCL propagates these timestamps to the profiler plugin so it can convert them to the CPU
 564 time domain.

565 (Source: [/ext-profiler/README.md](#))

566 **6.3 Potential Integration with Score-P**

567 The Score-P measurement infrastructure²² is a highly scalable and easy-to-use tool suite for profiling
 568 and event tracing of HPC applications. It supports a number of analysis tools. Currently, it works
 569 with Scalasca, Vampir, and Tau and is open for other tools and produces OTF2 traces and CUBE4
 570 profiles. Integrating NCCL into Score-P allows developers to see communication collectives alongside
 571 the application logic.
 572 A prerequisite for distributed tracing is the unique identification of process groups. NCCL achieves this
 573 via `ncclGetUniqueId`²³ without a central coordinator. To establish a communicator, one process
 574 generates a handle containing a random 64-bit `magic` value from `/dev/urandom` and the socket
 575 address of a new listening socket (IP, port), whose port is chosen by the operating system. This
 576 combination avoids collisions across a cluster, ensuring the ID is globally unique in practice. A Score-P
 577 integration can use these to accurately define Process Groups.

578 The integration could be achieved in two ways, either using a direct Profiler API mapping or via an
 579 indirect NVTX/CUPTI annotation:

²²<https://www.vi-hps.org/projects/score-p/overview>

²³<https://github.com/NVIDIA/nccl/tree/master/src/init.cc>

580 A direct integration would potentially involve implementing a NCCL profiler plugin that translates the
 581 `startEvent` and `stopEvent` callbacks into Score-P regions: The plugin maps NCCL event descrip-
 582 tors (e.g., `ncclAllReduce`) to Score-P regions using the instrumentation macros (e.g.,
 583 `SCOREP_USER_REGION_BY_NAME_BEGIN/END`).
 584 Alternatively, the NCCL profiler plugin can act as a bridge to NVIDIA’s Tools Extension (NVTX). If
 585 Score-P has been built with CUDA support it can intercept NVTX ranges. The NCCL profiler plugin
 586 would emit `nvtxRangePush`²⁴ and `nvtxRangePop` around NCCL operations. Score-P records
 587 these as labeled regions without requiring the plugin to link directly against Score-P libraries. This
 588 approach decouples the NCCL plugin from the Score-P build environment and instead relies on Score-
 589 P’s internal NVTX-to-OTF2 mapping logic.
 590 The plugin can utilize `cuptiActivityPush/PopExternalCorrelationId` to capture GPU
 591 activity during the `startEvent` and `stopEvent` of `KernelLaunch` events, while incrementing a
 592 thread-safe correlation ID (see Fig. 13). CUPTI can be initialized and cleaned up within the profiler
 593 plugin’s `init` and `finalize` functions.

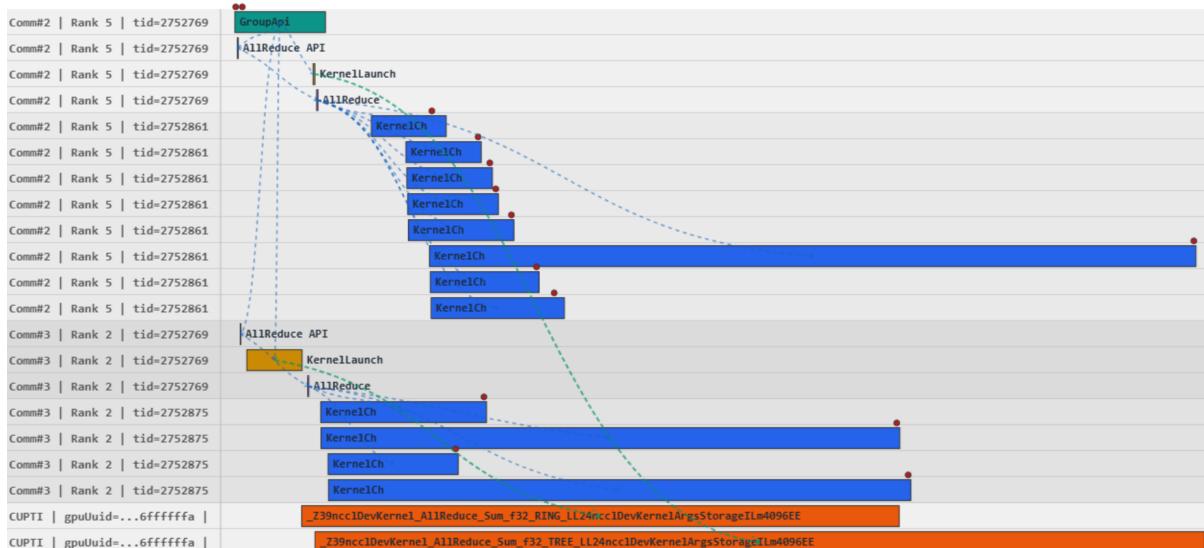


Figure 13: CUPTI activity is visualized as orange event bars. With a unique correlation Id, it is possible to trace the activity back to `KernelLaunch` events

594 7 Conclusion

595 This study examined the NCCL Profiler Plugin API and its suitability for integration with Score-P. It
 596 provided background on NCCL and its design, explained how the profiler plugin is loaded and described
 597 the API definition with its five core callbacks `init`, `startEvent`, `stopEvent`,
 598 `recordEventState` and `finalize`. Code examples and visualizations illustrate the event flow

²⁴https://nvidia.github.io/NVTX/doxygen/group__m_a_r_k_e_r_s__a_n_d__r_a_n_g_e_s.html

599 from API calls to NCCL’s internal profiler callbacks. Performance experiments showed that an empty
600 profiler adds roughly $8\text{--}9\ \mu\text{s}$ overhead per operation in single-node runs but introduces negligible over-
601 head in multi-node runs, and scaling to many GPUs across nodes required no changes to the profiler
602 plugin. The discussion covered developer considerations, known limitations, and a potential integration
603 strategy with Score-P.

604 The NCCL Profiler API allows for highly customized plugins tailored to the analysis needs, whether for
605 simple timing, kernel tracing via CUPTI, or integration with external tools such as Score-P. A notable
606 advantage is its low overhead: NVIDIA advertises their `inspector`²⁵ implementation as efficient
607 enough for “always-on” profiling in production. On the downside, profiler plugins may require mainte-
608 nance and active development, since NCCL is actively developed. API versions evolve and new features
609 are being introduced.

²⁵<https://github.com/NVIDIA/nccl/tree/master/ext-profiler/inspector>