

NCCL Profiler Plugin API – eine Machbarkeitsstudie

Inhaltsverzeichnis

1	TODO / Struktur (aus Markdown)	2
1.1	Table of Contents	2
1.2	Main content chunks / concepts	2
2	Abstract	3
3	Introduction	3
3.1	Comparison to MPI (TODO)	3
3.2	NCCL Concepts	3
4	The Profiler API	7
4.1	How nccl detects the profiler plugin (1.1)	7
4.2	The profiler API definition (1.2)	8
4.2.1	init	8
4.2.2	startEvent	9
4.2.3	stopEvent	10
4.2.4	recordEventState	10
4.2.5	finalize	10
4.2.6	name	11
4.3	Callback perspective from viewpoint of multi-node cluster	11
4.3.1	Callback-Verhalten bei verschiedenen Communicator-Initialisierungsstrategien	11
4.3.2	Callback-Verhalten bei spezifischen Knotenkonfigurationen	11

5	What can you do with it	12
5.1	Logging	12
5.2	Tracking & running metrics	12
5.3	Kernel tracing with CUPTI	12
6	Why would you use it? pros & cons	12
6.1	NCCL_DEBUG	13
7	Known limitations	13
8	TODO	13

1 TODO / Struktur (aus Markdown)

1.1 Table of Contents

- 0. Abstract – GPU communication profiling/tracing motivieren
- 0. Introduction – vorausgesetztes Verständnis soweit nötig (z. B. MPI, SLURM, NCCL)
- 1. Die Profiler API
 - 1.1 Einbindung des Plugins in NCCL. (first draft; TODO final draft)
 - 1.2 “rohe” API-Definition, kurz und knapp
 - 1.3 Der Codeflow: Application NCCL User API → Profiler API
 - 1.4 Codeflow++: ncclGroup, multi GPU streams, multi-threaded, multi-node
- 2. Was einem die Profiler Plugin API (nicht) ermöglicht (Logging, running metrics, CUPTI, ...)
- 3. Warum (nicht) die Profiler Plugin API in Erwägung ziehen? (Experimente, Genauigkeit, Vor-/Nachteile)
- 4. Conclusion – Nützlichkeit für P-Score-Messsystem

1.2 Main content chunks / concepts

- Einfache Code-Beispiel-Walkthroughs
- Swim-Lane-Diagramme (User API → init/finalize; start/stop/recordEventState)
- Benchmarking, Messungen
- Conclusion

2 Abstract

- AI – großer Use Case für HPC
- Teure Workloads; Wunsch, Anwendungsperformance zu verstehen und zu optimieren
- Großer Teil von AI-Workloads ist GPU-Kommunikation (oft über viele GPUs)
- NCCL – die Bibliothek, die Kommunikationsroutinen für NVIDIA-GPUs implementiert
- Bietet eine Schnittstelle, einen eigenen Profiler in NCCL einzubinden und Performancedaten zu extrahieren

3 Introduction

TODO: nach Bedarf erwähnen – MPI-Konzepte, NCCL-Konzepte, SLURM-Konzepte.

3.1 Comparison to MPI (TODO)

MPI

- Zentriert um CPU-Prozesse
- Rank = CPU-Prozess
- 1 Ranks/Tasks pro 1 CPU-Prozess

NCCL

- Zentriert um GPUs; CPU-Threads helfen kommunikation zu orchestrieren
- Rank = GPU-Device
- mehrere Ranks pro CPU-Thread möglich (? TODO true if i get this to work: check slurm 2694771 multi gpu per task)

3.2 NCCL Concepts

Bevor die Profiler Plugin API genauer betrachtet wird, ist es hilfreich zu verstehen, was NCCL intern tut, wenn eine Anwendung die NCCL User API aufruft.

Eine typische NCCL-Anwendung folgt dieser Grundstruktur:

```

// create nccl communicators
createNcclComm();

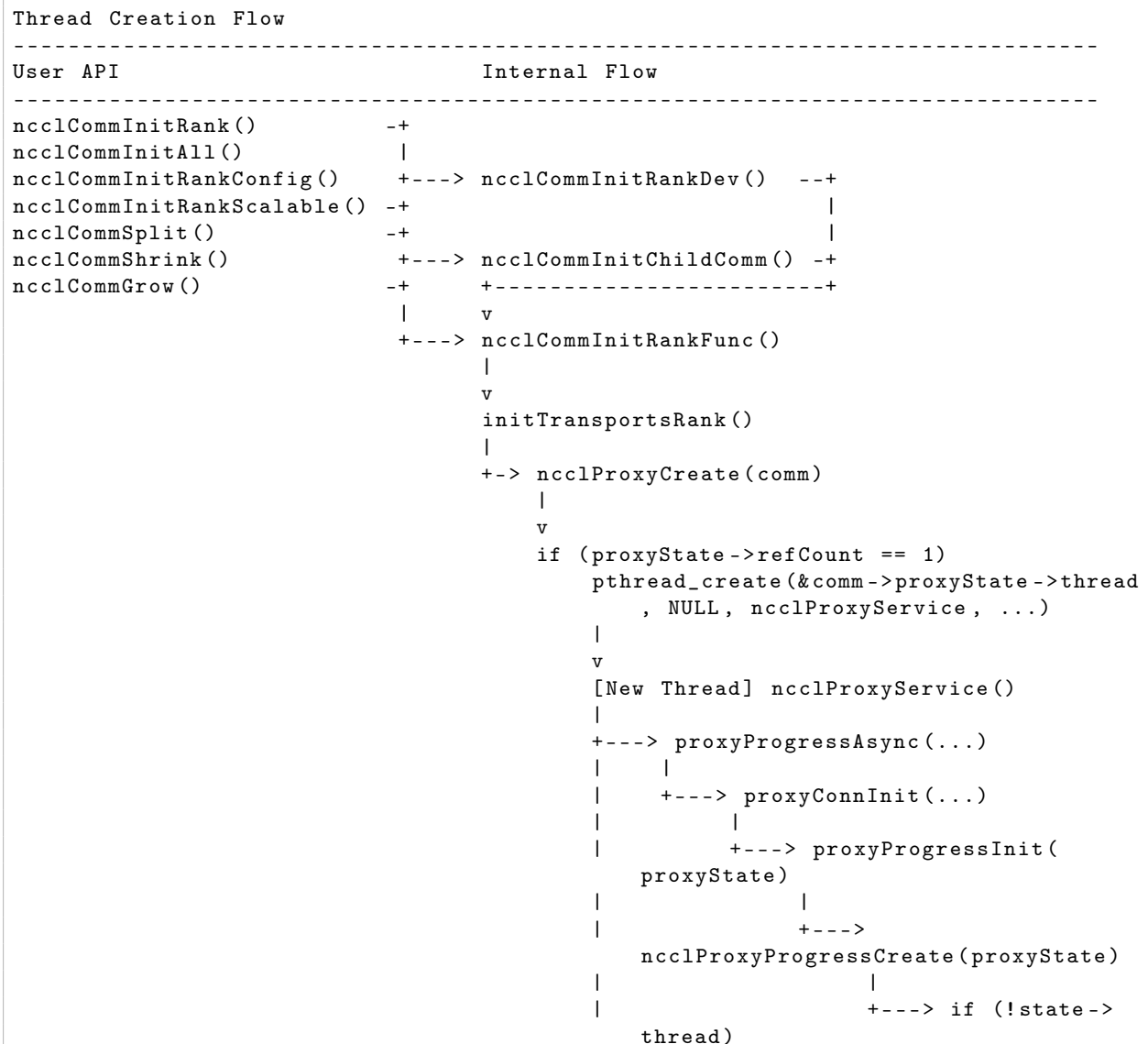
// allocate memory for computation and communication
prepareDeviceForWork();

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

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

```

Bei der Erstellung von NCCL-Communicators startet NCCL intern einen Thread namens **ProxyService**. Dieser startet verzögert einen weiteren Thread **ProxyProgress**, der Netzwerkanfragen für GPU-Kommunikation bei Collective- und P2P-Operationen bearbeitet.



```

|
| pthread_create(&state->thread, NULL,
| ncclProxyProgress, proxyState)
|
| v
| [New Thread]
| ncclProxyProgress()
|
+---> proxyServiceInitOp(...)
      +---> proxyProgressAsync(...) (
          same path as above)

```

Die Bedingungen `if (proxyState->refCount == 1)` und `if (!state->thread)` stellen sicher, dass diese Threads nur einmal pro geteilter Ressource (`struct ncclSharedResources`) erzeugt werden. Ausschnitte der relevanten Structs:

`/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;
    // other fields
}

```

Standardmäßig hat jeder NCCL-Communicator seine eigene Shared Resource. Ruft die Anwendung `ncclCommSplit()` oder `ncclCommShrink()` auf, wobei der ursprüngliche Communicator mit einem `ncclConfig_t` mit `splitShare` bzw. `shrinkShare = 1` initialisiert wurde, teilt sich der neue Communicator die Shared Resource (und die Proxy-Threads) mit dem Eltern-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. */

```

(Zitat aus `/src/proxy.cc`)

Später entscheidet NCCL bei jedem Aufruf der NCCL User API intern über die Netzwerkoperationen und hängt sie an einen Pool. Der ProxyProgress-Thread liest diese Operationen aus dem Pool und führt sie aus. Die folgenden Pfade führen zu `ncclProxyPost()`, wo Ops in einen Pool geschrieben und der Proxy-Progress-Thread signalisiert wird.

Flow from User API Calls to `ncclProxyPost()`

User API

```

-----
ncclCommInitAll()          -+      ncclAllGather()          -+
ncclCommInitRankConfig()  |      ncclAlltoAll()           |
ncclCommInitRankScalable() |      ncclAllReduce()          |
ncclCommFinalize()         |      ncclBroadcast()          |
ncclCommDestroy()         |      ncclGather()           |
ncclCommRevoke()          |      ncclReduce()            |
ncclCommAbort()           |      ncclReduceScatter()       |
ncclCommSplit()           |      ncclScatter()           |
ncclCommShrink()          |      ncclSend()             |
ncclCommGrow()            |      ncclRecv()              -+
ncclDevCommCreate()       |                               |
ncclCommWindowRegister()  |                               |
ncclGroupSimulateEnd()    -+                               |
                          |                               |
Internal Flow            |                               v
                          |                               ncclEnqueueCheck()
                          +-----+
                          v
                          ncclGroupEndInternal()
                          +----+
                          |      v
                          |      groupLaunchNonBlocking()
                          +----+
                          v
                          groupLaunch() -> doLaunches()
                          +--> ncclLaunchPrepare() -> ...
                          +--> ncclProxyStart() -> ncclProxySaveOp() -> ...
                          v
                          ncclProxyPost() (proxy.cc)
                          +--> [Posts Ops to pool]
                          +--> [Signals Proxy Progress Thread]

```

Der Proxy-Progress-Thread liest aus diesem Pool bei `ncclProxyGetPostedOps()` und führt die Ops aus.

`/src/proxy.cc`

```

ncclProxyProgress() progressing loop
-----
ncclProxyProgress(proxyState)
+--> do {
    +--> progressOps(proxyState, ...)
    |   +--> while (op) {
    |       op->progress(proxyState, op);
    |       op = op->next;
    |   }
    +--> ncclProxyGetPostedOps()
        +--> [reads Ops or thread will wait]
    } while (...)

```

Dieses Verhalten zu verstehen ist nützlich für die Profiler Plugin API und netzwerkbezogene Aktivität im nächsten Abschnitt.

4 The Profiler API

4.1 How nccl detects the profiler plugin (1.1)

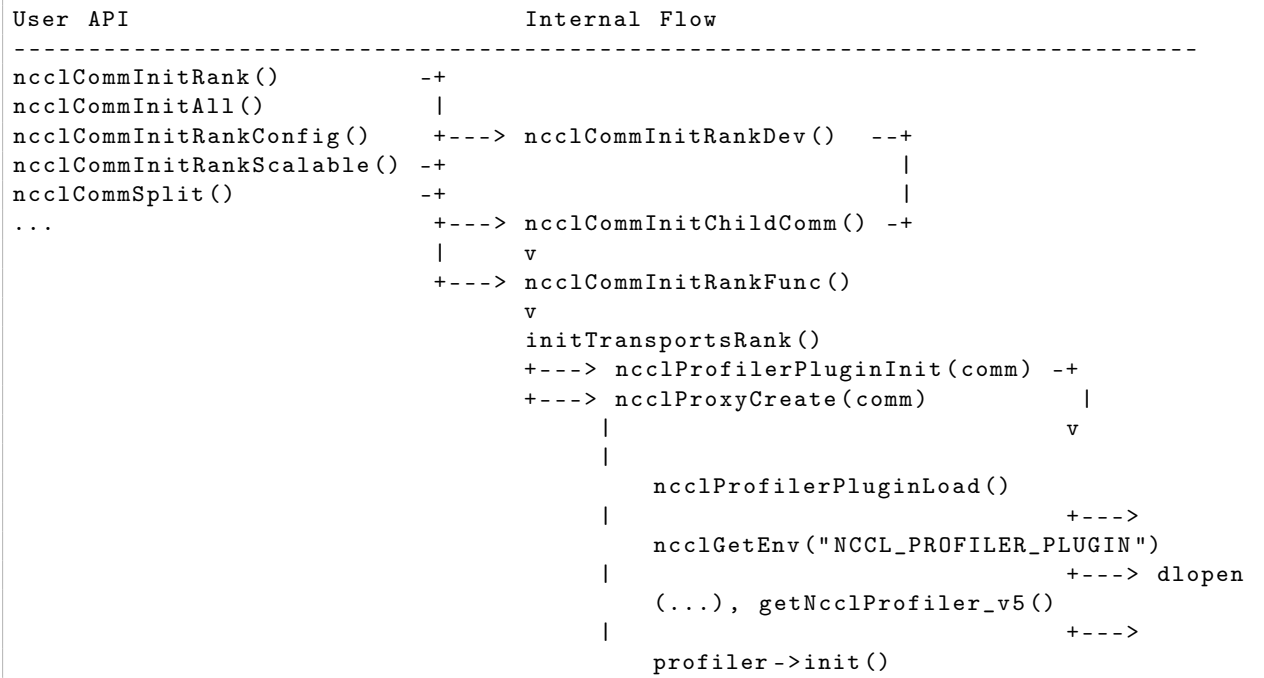
Bei der Erstellung eines NCCL-Communicators sucht NCCL nach einer Shared Library, die das Profiler-Plugin repräsentiert, und prüft die Umgebungsvariable `NCCL_PROFILER_PLUGIN`: `profilerName = ncclGetEnv("NCCL_PROFILER_PLUGIN")`.

Dann werden `handle* = dlopen(name, RTLD_NOW | RTLD_LOCAL)` und `ncclProfiler_v5 = (ncclProfiler_v5_t*)dlsym(handle, "ncclProfiler_v5")` aufgerufen, um die Library sofort mit lokaler Symbol-Sichtbarkeit zu laden.

- Wenn `NCCL_PROFILER_PLUGIN` gesetzt ist: Lade die Library mit dem angegebenen Namen; schlägt das fehl, versuche `libnccl-profiler-<NCCL_PROFILER_PLUGIN>.so`.
- Wenn `NCCL_PROFILER_PLUGIN` nicht gesetzt ist: versuche `libnccl-profiler.so`.
- Wurde kein Plugin gefunden: Profiling ist deaktiviert.
- Wenn `NCCL_PROFILER_PLUGIN` auf `STATIC_PLUGIN` gesetzt ist, werden die Plugin-Symbole in der Programmbinary gesucht.

(Quelle: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-profiler-plugin>)

Der Ablauf vom User-API-Aufruf bis zum Laden des Profiler-Plugins:



Das Profiler-Plugin wird beim Erstellen eines Communicators geladen (vor der Proxy-Thread-Erstellung). Der Plugin-Lademechanismus erwartet, dass der Struct-Variablenname der Konvention `ncclProfiler_v{versionNum}` folgt; damit ist auch die API-Version angegeben.

Die Profiler-API hat sich mit neueren NCCL-Releases mehrfach geändert; die Abwärtskompatibilität zu älteren Plugins scheint begrenzt (TODO: Faktencheck). Die genaue Implementierung steht in `/src/plugin/plugin_open.cc` und `/src/plugin/profiler.cc`.

4.2 The profiler API definition (1.2)

Das Plugin muss eine von NCCL spezifizierte Profiler-API implementieren, indem es einen Struct exportiert. Dieser Struct enthält Zeiger auf alle vom API geforderten Funktionen. Ein Plugin kann mehrere versionierte Structs exportieren (Abwärtskompatibilität).

```
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
};
```

Die vollständige API liegt unter `/src/include/plugin/profiler/profiler_v{versionNum}.cc`. Ab NCCL v2.29.1 ist Version 6 die neueste; fünf Funktionen müssen implementiert werden. Intern kapselt NCCL Aufrufe in eigene Funktionen (u. a. in `/src/include/profiler.h`).

NCCL ruft die Profiler-API auf verschiedenen Ebenen auf, um Start/Ende von Groups, Collectives, P2P, Proxy-, Kernel- und Netzwerkaktivität zu erfassen. Im Folgenden werden die API-Funktionen und die Stellen, an denen NCCL sie aufruft, erläutert.

4.2.1 init

`init` initialisiert das Profiler-Plugin. NCCL übergibt u. a.:

```
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
);
```

`init()` wird unmittelbar nach erfolgreichem Plugin-Load in `ncclProfilerPluginLoad()` aufgerufen (siehe Diagramm oben) – und bei jeder Communicator-Erstellung (anders als die Proxy-Threads). Gibt `init` nicht `ncclSuccess` zurück, deaktiviert NCCL das Plugin.

Sobald NCCL das Plugin und das richtige `ncclProfiler`-Symbol findet, ruft es die `init`-

Funktion auf. So kann das Plugin seinen internen Kontext für die Profilierung von NCCL-Events initialisieren.

(Quelle: `/ext-profiler/README.md`)

`void** context` ist ein opaker Handle, auf den der Plugin-Entwickler ein beliebiges Kontextobjekt zeigen kann; dieser Zeiger wird bei `startEvent` und `finalize` wieder übergeben. Das Kontextobjekt ist pro Communicator getrennt.

Der Plugin-Entwickler soll `int* eActivationMask` auf eine Bitmaske setzen, die angibt, welche Event-Typen getrackt werden. Die Zuordnung steht in `/src/include/plugin/nccl_profiler.h`; intern ist die Maske standardmäßig 0 (keine Events). 4095 = alle Events.

TODO: Verwendung von `ncclDebugLogger_t logfn`?

4.2.2 startEvent

`startEvent` wird aufgerufen, wenn NCCL bestimmte Operationen beginnt:

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

Ab Release v2.29.1 ignoriert NCCL den Rückgabewert. `void** eHandle` kann auf ein benutzerdefiniertes Event-Objekt zeigen; dieser Zeiger wird bei `stopEvent` und `recordEventState` wieder übergeben. `eDescr` beschreibt das gestartete Event; Details in `/src/include/plugin/profiler/`. Das Feld `void* parentObj` im Event-Deskriptor ist der `eHandle` eines Parent-Events (oder null).

Alle User-API-Aufrufe zu Collectives oder P2P starten ein Group-API-Event; bei Netzwerkbedarf (multi-node) können ProxyCtrl-Events emittiert werden. Je nach `eActivationMask` werden weitere (Kind-)Events in tieferen Codebereichen emittiert – eine Event-Hierarchie mit mehreren Tiefen:

```
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
```

(Quelle: `/ext-profiler/README.md`)

Wenn das Profiler-Plugin Tracking für Event-Typen weiter unten in der Hierarchie aktiviert, trackt NCCL auch die Parent-Typen. Die folgenden Diagramme zeigen, wo NCCL `startEvent` und `stopEvent` emittiert (siehe Markdown/Quellcode für vollständige ASCII-Diagramme). Implementierung: `/src/init.cc`, `/src/plugin/profiler.cc`.

Der ProxyProgress-Thread emittiert ebenfalls `startEvent/stopEvent` beim Fortschreiten der Ops (z. B. ProxyStep, KernelCh, NetPlugin-Events). Die transport-spezifischen Progress-Funktionen stehen in `/src/transport/net.cc`, `coll_net.cc`, `p2p.cc`, `shm.cc`.

Ohne `NCCL_PXN_DISABLE=0` (Standard 1) kann wegen PXN (PCIe x NVLink) ein Teil der Proxy-Ops in einem Proxy-Thread eines anderen Prozesses laufen. Dann ist `parentObj` im `eDescr` nicht sicher dereferenzierbar; das `eDescr` für ProxyOp-Events enthält die PID des Ursprungs, die der Profiler mit der lokalen PID vergleichen kann. ProxyStep liefert dieses Feld nicht; der Profiler kann über getrackte Kontexte prüfen, ob der übergebene `context` zum lokalen Prozess gehört.

4.2.3 stopEvent

`stopEvent` teilt dem Plugin mit, dass das Event beendet ist.

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

NCCL ignoriert den Rückgabewert. `stopEvent` wird in denselben Funktionen wie `startEvent` aufgerufen (außer beim GroupApi-Event, siehe Diagramm).

4.2.4 recordEventState

Einige Event-Typen können durch `recordEventState` aktualisiert werden (Zustand und Attribute). Unterstützte Zustände: `/src/include/plugin/profiler/profiler_v{versionNum}.h`.

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

Aufruf an denselben Stellen wie `startEvent`.

4.2.5 finalize

Nach einem User-API-Aufruf zum Freigeben der Communicator-Ressourcen wird `finalize()` in `ncclProfilerPluginFinalize()` aufgerufen; danach wird das Plugin per `dlclose(handle)` in `ncclProfilerPlugin` entladen.

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

User API		Internal Flow
<code>ncclCommAbort()</code>	-+	
<code>ncclCommDestroy()</code>	+----->	<code>commReclaim()</code>
		+---> <code>ncclProfilerPluginFinalize()</code>
		+---> <code>ncclProfiler->finalize()</code>
		+---> <code>ncclProfilerPluginUnload()</code>

Details: `/src/init.cc`, `/src/plugin/profiler.cc`, `/src/plugin/plugin_open.cc`.

4.2.6 name

Der Profiler-Plugin-Struct hat zusätzlich ein Feld `name`.

Das Feld `name` soll auf eine Zeichenkette mit dem Namen des Profiler-Plugins zeigen. Es wird für alle Log-Ausgaben verwendet, besonders bei `NCCL_DEBUG=INFO`.

(Quelle: `/ext-profiler/README.md`)

TODO: Copy-Engine-basierte Events?

4.3 Callback perspective from viewpoint of multi-node cluster

Neben den durch NCCL-API-Aufrufe ausgelösten Profiler-API-Aufrufen wird NCCL bei jeder Communicator-Erstellung mehrfach initialisiert. Die Multi-Node-Umgebung beeinflusst die Gesamtzahl der Callbacks. Verschiedene Szenarien (Initialisierungsstrategien, Knotenkonfigurationen) werden im Mark-down detailliert beschrieben; hier eine kompakte Übersicht.

4.3.1 Callback-Verhalten bei verschiedenen Communicator-Initialisierungsstrategien

TODO: Verhalten bei `one_device_per_process_mpi` vs. `one_device_per_thread`; aus NCCL-Sicht: Identifikation über `ncclUniqueId` bis auf Thread-Ebene.

- Strategie `one_device_per_process_mpi`
- Strategie `one_device_per_thread`
- Strategie mit eigenem Netzwerk-Socket-Code statt MPI

4.3.2 Callback-Verhalten bei spezifischen Knotenkonfigurationen

Konfigurationen: n Knoten, m GPUs/Knoten, p Tasks auf m GPUs:

- `tasks/node < gpus/node` (verschiedene Unterfälle; z. B. 1 Task/node \Rightarrow ähnlich `one_device_per_thread`; n tasks/node kann zu Crashes bei Collectives führen)
- `tasks/node = gpus/node` \Rightarrow Verhalten wie `one_device_per_process_mpi`
- `tasks/node > gpus/node` (kann bei Communicator-Init crashen; TODO)

(Vollständige Aufzählung und TODOs siehe `uni_nccl_report.md`.)

5 What can you do with it

Aufgrund der asynchronen Natur von NCCL-Operationen sind Events zu Collectives und P2P nicht präzise abgrenzbar. `stopEvent` bei Collectives bedeutet nur, dass das Collective enqueued wurde. Ohne Proxy- und/oder Kernel-Aktivität kann das Plugin nicht feststellen, wann ein Collective endet. Mit aktivierten Proxy-/Kernel-Events kann das Plugin das Ende schätzen.

(leicht umformuliert nach `/ext-profiler/README.md`)

5.1 Logging

- Logging-Funktion aus `init` (TODO)
- Code-Snippet: eigene Logging-Infrastruktur, Timestamping

5.2 Tracking & running metrics

- Code-Snippet: CRUD des benutzerdefinierten Kontextobjekts
- Code-Snippet: CRUD des benutzerdefinierten Event-Objekts

5.3 Kernel tracing with CUPTI

- CUPTI-Ext.-ID-Mechanismus kurz erläutern
- Code-Snippet: CUPTI init/cleanup und Nutzung
- Veränderung des Profiling-Verhaltens zur Laufzeit (TODO: `example_profiler`, Inspector prüfen)

6 Why would you use it? pros & cons

- Anpassbar (customizable)

- Kann Wartung / aktive Weiterentwicklung erfordern, da NCCL aktiv weiterentwickelt wird
- Overhead: NVIDIA wirbt damit, dass ihre **inspector**-Implementierung effizient genug für “always-on” in Produktion sei

6.1 NCCL_DEBUG

<https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/env.html#nccl-debug>

NCCL bringt bereits Debug-Logging mit verschiedenen Granularitätsstufen mit:

- INFO – Debug-Informationen
- TRACE – Replay-fähige Trace-Informationen bei jedem Aufruf
- Weitere Optionen (v2.2.12 NCCL_DEBUG_FILE, v2.3.4 NCCL_DEBUG_SUBSYS, v2.26 Timestamp-Format/Levels)

7 Known limitations

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

Die Instrumentierung von Kernel-Events nutzt Zähler, die der Kernel an den Host und den Proxy-Progress-Thread liefert. Die Proxy-Progress-Thread-Infrastruktur wird also von Netzwerk und Profiler geteilt. Wenn der Proxy Netzwerkanfragen bedient, kann das Auslesen der Kernel-Profiling-Daten verzögert werden (Genauigkeitsverlust). Ebenso bei starker CPU-Last und verzögerter Planung des Proxy-Progress-Threads.

Ab Profiler-Version 4 verwendet NCCL einen Ringpuffer von 64 Elementen pro Kanal. Jeder Zähler wird durch zwei vom NCCL-Kernel gelieferte Timestamps (ptimers) ergänzt (Start/Ende der Operation im Kernel). NCCL übergibt diese Timestamps an das Profiler-Plugin, damit es sie in die CPU-Zeitdomäne umrechnen kann.

8 TODO

- Code-Snippets mit Quellcode-Dateien und Zeilennummern verlinken
- Nutzen PyTorch (+ JAX, TensorFlow) die Profiler-Plugin-API?