

PR #49274 完整报告

vllm-project/vllm

[Bugfix] Make metadata send non-blocking in GroupCoordinator.isend_tensor_dict

合并时间: 2026-08-31 07:51

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/49274>

执行摘要

- 一句话: PP 元数据发送改异步, 修死锁与 UAF 并提升吞吐
- 推荐动作: 值得精读。这是分布式通信层一次小而关键的语义修正: `_RetainedHandle` 的幂等 `wait`、以设备句柄为闸门而 `gloo` 句柄仅作 `backstop` 的完成判定策略、以及 `FIFO + 惰性收割` 的自保留设计, 都是处理异步通信 `buffer` 生命周期问题的可借鉴模式。重点关注 `vllm/distributed/parallel_state.py` 的 `isend_object` / `_reap_completed_isends` 与 `gpu_worker.py` 的 `fire-and-forget` 改动; 也建议留意 `depthfirst-app` 两条未解决的 `gloo` 完成判定评论, 后续若 CPU 张量或 `metadata-only` 发送进入 PP 路径需先补上。

功能与动机

PR body 明确列出两个缺陷: Bug 1 是阻塞性元数据发送——`isend_tensor_dict` 名义上非阻塞, 但元数据经同步 `send_object` (`gloo send`) 投递, `gloo` 在 `eager threshold` 之上是 `rendezvous` 语义, 会一直阻塞到对端发起匹配 `recv`; 虚拟流水线 `fold-point` 发送 (如 `vllm-ascend`) 中匹配 `recv` 要到后续迭代才出现, 导致两端在元数据发送内部互相死锁, 与 `isend_` 的承诺完全相反。Bug 2 是 `fire-and-forget` 调用方缺少保留——`pickle` 出的 `size_tensor/object_tensor` 是局部临时量, 调用方丢弃返回句柄后 `gloo` 后台线程拷贝期间 `buffer` 可能被释放 (`use-after-free`); 且 `record_stream` 只在 `tensor.is_cuda` 时执行, 非 CUDA 设备上的在途发送 `buffer` 可能被后续计算释放或覆写。作者预期修复对 PP + `chunked prefill` 有性能收益。

实现拆解

1. 新增 `_RetainedHandle` 与 `isend_object` (`vllm/distributed/parallel_state.py`): 把同步 `send_object` 拆成「先 `isend` 8 字节长度、再 `isend pickle` 内容」的两段式 `gloo` 非阻塞发送, 保持 `FIFO` 顺序使对端 `recv_object` 无需改动; `_RetainedHandle` 持有两个序列化 CPU 张量与底层 `work`, `wait()` 幂等 (`gloo` 点对点 `Work.wait()` 单次性已实测), 排空后释放保留引用。
2. `isend_tensor_dict` 以 `isend_object` 打底并自保留: 句柄列表首元素改为 `isend_object(metadata_list)` 的复合句柄, 替代阻塞的 `send_object`; 同时把 (`handles`, `sent_tensors`) 追加到新增的 `_pending_isends` `FIFO`, 解决 `fire-and-forget` 调用方的 `buffer` 存活问题。该属性延迟初始化, 以兼容未调用 `super().__init__()` 的子类。
3. 新增 `_reap_completed_isends` 惰性收割: 每次 `isend_tensor_dict` 调用前先收割。完成判定只用设备后端 `tensor` 句柄 (`handles[1:]`), 因为 `gloo Work.is_completed()` 可能在后

台拷贝完成前就返回 True; tensor 全部完成后对 metadata 句柄 wait() 仅作 backstop 并释放引用。纯 metadata 条目 (无 tensor 句柄) 没有可靠完成信号, 按 best-effort 直接弹出。

4. 同步路径清理: send_tensor_dict 在 wait 完所有句柄后, 通过 pending[-1][0] is handles 判定并弹出自己的保留条目, 避免同步调用也滞留引用。
5. worker 侧配套与测试: vllm/v1/worker/gpu_worker.py 删除 _pp_send_work 字段及 execute_model 开头对上一次发送的 wait, PP 中间张量完全 fire-and-forget 投递; tests/distributed/test_comm_ops.py 新增 4 个单测, 覆盖 size-then-object 顺序与 wait 释放、复合 metadata 句柄、fire-and-forget 自保留与 FIFO 收割、同步路径不留残留。

关键文件:

- vllm/distributed/parallel_state.py (模块 分布式通信; 类别 source; 类型 core-logic; 符号 _RetainedHandle, isend_object, _reap_completed_isends, isend_tensor_dict) : 核心修改文件: 新增 _RetainedHandle、isend_object、_reap_completed_isends, 重写 isend_tensor_dict 的元数据通路 with 自保留逻辑, 并让 send_tensor_dict 同步路径弹出保留条目。这是修复死锁与 UAF 的关键所在。
- tests/distributed/test_comm_ops.py (模块 通信测试; 类别 test; 类型 test-coverage; 符号 test_isend_object_posts_size_then_object_and_releases_on_wait, test_isend_tensor_dict_includes_metadata_handle, test_isend_tensor_dict_self_retains_for_fire_and_forget, test_send_tensor_dict_sync_path_does_not_self_retain) : 新增 4 个单测覆盖 size-then-object 顺序与 wait 释放、复合 metadata 句柄、fire-and-forget 自保留与 FIFO 收割、同步路径不留残留; _DummyWork 增加 is_completed 以支持完成状态模拟。
- vllm/v1/worker/gpu_worker.py (模块 执行器; 类别 source; 类型 core-logic; 符号 execute_model, init) : PP 中间张量发送从「存句柄 + step 开头 wait」改为完全 fire-and-forget, 是本次语义变更在 v1 执行路径上的直接消费方。

关键符号: isend_object, _reap_completed_isends, isend_tensor_dict, send_tensor_dict, _RetainedHandle.wait

关键源码片段

vllm/distributed/parallel_state.py

核心修改文件: 新增 _RetainedHandle、isend_object、_reap_completed_isends, 重写 isend_tensor_dict 的元数据通路 with 自保留逻辑, 并让 send_tensor_dict 同步路径弹出保留条目。这是修复死锁与 UAF 的关键所在。

vllm/distributed/parallel_state.py: 核心改动一 —— 幂等句柄与对象级非阻塞发送

```
class _RetainedHandle:
```

```
    """持有源张量直到所有已提交的通信工作完成。
```

```
    用于 ``isend_object``, 保证 gloo 后台线程在拷贝期间序列化后的 CPU 张量不会被释放; ``wait`` 在排空后丢弃引用。
```

```
    ``wait`` 必须幂等: gloo 的 ``Work.wait()`` 对点对点操作是单次的, 对已
```

完成的 send 再次调用会永久阻塞（已实测验证）。worker 的 step 开头排空与 ``\_reap\_completed\_isends`` 可能 wait 同一个条目的 metadata handle，因此第二个调用者必须直接返回，不能重复 wait 底层 work。

"""

```
def __init__(self, works: list[Any], retained: tuple[torch.Tensor, ...]) -> None:
    self._works = works
    self._retained = retained # 序列化源 buffer, wait 后排空
    self._waited = False
```

```
def is_completed(self) -> bool:
    # 注意: gloo 的 is_completed 在后台拷贝结束前就可能返回 True,
    # 因此该方法只适合同步路径或设备后端句柄, 不作收割闸门使用
    return all(w.is_completed() for w in self._works)
```

```
def wait(self) -> None:
    # 幂等: 重复 wait 直接返回, 避免对单次 gloo work 二次 wait 死锁
    if self._waited:
        return
    for w in self._works:
        w.wait()
    self._waited = True
    self._retained = ()
```

```
def isend_object(self, obj: Any, dst: int) -> Handle:
    """非阻塞版 ``send_object``: 在 CPU group 上先 isend 长度、再 isend pickle 对象。
```

返回的 ``\_RetainedHandle`` 持有序列化源张量, 直到 ``wait`` 排空两个 send。

"""

```
assert dst < self.world_size, f"Invalid dst rank ({dst})"
assert dst != self.rank_in_group, (
    "Invalid destination rank. Destination rank is the same "
    "as the current rank."
)
```

```
# 先发 8 字节长度、再发 uint8 内容, 保持与 send_object 一致的
# gloo FIFO 顺序, 对端 recv_object 无需改动
object_tensor = torch.frombuffer(pickle.dumps(obj), dtype=torch.uint8)
size_tensor = torch.tensor(
    [object_tensor.numel()], dtype=torch.long, device="cpu"
)
```

```
retained = (size_tensor, object_tensor)
works: list[Any] = []
for tensor in retained:
    work = torch.distributed.isend(
        tensor, dst=self.ranks[dst], group=self.cpu_group
    )
```

```
if work is not None:
    works.append(work)
```

```
return _RetainedHandle(works, retained)
```

vllm/distributed/parallel_state.py: 核心改动二 —— FIFO 惰性收割

```
def _reap_completed_isends(self) -> None:
```

```
    """惰性收割已完成的 ``isend_tensor_dict`` 自保留条目，按 FIFO 从旧到新。
```

gloo 的 ``Work.is\_completed()`` 不可靠（后台拷贝完成前就可能报告完成），因此 metadata handle (``handles[0]``) 绝不作完成闸门；改为以设备后端的 tensor handle (``handles[1:]``) 为准。所有 tensor send 完成后，对端必然已发起匹配 recv；而接收方先消费 metadata 再收 tensor，此时 gloo metadata send 必然已完成，对它 ``wait()`` 近似瞬时，仅用于释放保留引用。

纯 metadata 条目 (``handles[1:]`` 为空，如空 ``IntermediateTensors``) 没有可靠完成信号，按 best-effort 直接丢弃。

```
    """
```

```
    pending = getattr(self, "_pending_isends", None)
```

```
    if pending is None:
```

```
        # 部分子类复制了 __init__ 但未调用 super().__init__()，需要兜底
        self._pending_isends = pending = deque()
```

```
    while pending:
```

```
        handles, _ = pending[0]
```

```
        tensor_handles = handles[1:]
```

```
        if not tensor_handles:
```

```
            pending.popleft() # 纯 metadata 条目，best-effort 丢弃
            continue
```

```
        if not all(h.is_completed() for h in tensor_handles):
```

```
            break # FIFO: 最老的 send 未完成，后面的不会更靠前，停止收割
```

```
        handles[0].wait() # backstop: 等 gloo metadata 并释放保留引用
        pending.popleft()
```

vllm/distributed/parallel_state.py: 核心改动三 —— fire-and-forget 自保留

```
def isend_tensor_dict(
```

```
    self,
```

```
    tensor_dict: dict[str, torch.Tensor | Any],
```

```
    dst: int | None = None,
```

```
    all_gather_group: "GroupCoordinator | None" = None,
```

```
    all_gather_tensors: dict[str, bool] | None = None,
```

```
) -> list[Handle]:
```

```
    """异步发送张量字典。
```

返回的句柄还会与发送源张量一起被自保留在 ``self.\_pending\_isends``，并在后续调用中惰性收割，因此 fire-and-forget 调用方即使丢弃句柄，也不会在发送期间释放或覆写 buffer；保留句柄的调用方仍可像以前一样 ``wait()``。

```

"""
if self.world_size <= 1:
    return []

if dst is None:
    dst = (self.rank_in_group + 1) % self.world_size
    assert dst < self.world_size, f"Invalid dst rank ({dst})"

if self.use_cpu_custom_send_recv:
    if self.device_communicator is None:
        raise ValueError("No device communicator found")
    # 自定义设备通信器路径是同步的
    self.device_communicator.send_tensor_dict(tensor_dict, dst) # type: ignore
    return []

group = self.device_group
metadata_group = self.cpu_group

metadata_list, tensor_list = _split_tensor_dict(tensor_dict)
tensor_keys = [k for k, v in tensor_dict.items() if isinstance(v, torch.Tensor)]
assert len(tensor_keys) == len(tensor_list)

# 先收割再投递，把自保留 FIFO 控制在有界范围内
self._reap_completed_isends()
handles: list[Handle] = [self.isend_object(metadata_list, dst=dst)]

sent_tensors: list[torch.Tensor] = []
for key, tensor in zip(tensor_keys, tensor_list):
    if tensor.numel() == 0:
        continue

    if self._should_use_all_gather(
        key, tensor.numel(), all_gather_group, all_gather_tensors
    ):
        tensor = tensor.reshape(all_gather_size, -1)[all_gather_rank]

    comm_group = metadata_group if tensor.is_cpu else group
    handle = torch.distributed.isend(
        tensor, dst=self.ranks[dst], group=comm_group
    )
    if tensor.is_cuda:
        tensor.record_stream(torch.cuda.current_stream(tensor.device))
    handles.append(handle)
    sent_tensors.append(tensor)

# fire-and-forget 自保留：句柄与源张量一起入队，发送完成后由
# _reap_completed_isends 惰性收割；非 CUDA 设备没有 record_stream
# 等价机制，因此必须统一兜底
pending = getattr(self, "_pending_isends", None)

```

```

if pending is None:
    self._pending_isends = pending = deque()
    pending.append((handles, sent_tensors))
    return handles

```

tests/distributed/test_comm_ops.py

新增 4 个单测覆盖 size-then-object 顺序与 wait 释放、复合 metadata 句柄、fire-and-forget 自保留与 FIFO 收割、同步路径不留残留；_DummyWork 增加 is_completed 以支持完成状态模拟。

tests/distributed/test_comm_ops.py: 验证 fire-and-forget 自保留与 FIFO 收割

```

def test_isend_tensor_dict_self_retains_for_fire_and_forget(
    monkeypatch: pytest.MonkeyPatch,
) -> None:
    works: list[_DummyWork] = []

    # 用假 isend 记录所有投递的 work，便于手动控制完成状态
    def fake_isend(t: torch.Tensor, *args: Any, **kwargs: Any) -> _DummyWork:
        w = _DummyWork()
        works.append(w)
        return w

```

```

monkeypatch.setattr(torch.distributed, "isend", fake_isend)

```

```

g = _make_group_for_unit_test(rank_in_group=0, world_size=2)
td = {"a": torch.arange(4, dtype=torch.float32, device="cpu")}

```

```

# fire-and-forget: 调用方丢弃返回句柄，group 必须自保留句柄与源张量
g.isend_tensor_dict(td, dst=1)
assert len(g._pending_isends) == 1
handles0, tensors0 = g._pending_isends[0]
assert len(handles0) == 2 # metadata 复合句柄 + 一个 tensor 发送
assert tensors0 == [td["a"]]

```

```

# 第一次发送仍在途，下一次调用不会收割它
g.isend_tensor_dict(td, dst=1)
assert len(g._pending_isends) == 2

```

```

# 纯 metadata 发送（无 tensor）没有可靠完成信号，收割时 best-effort 丢弃
g.isend_tensor_dict({"meta": "no-tensors"}, dst=1)
assert len(g._pending_isends) == 3

```

```

# 全部完成后，下一次调用把三个旧条目全部收割，只保留自己新投递的
for w in works:
    w.completed = True
g.isend_tensor_dict(td, dst=1)
assert len(g._pending_isends) == 1
# 收割过的条目其 metadata 句柄已被 wait 兜底，保留引用已释放

```

```
assert handles0[0]._retained == ()
```

评论区精华

- 基准前提 (njhill) : njhill 要求作者交代基准 env/cmd, 并明确只关心 VLLM_USE_V2_MODEL_RUNNER=1 (MRV2) 下的收益, 因为 MRV1 的 PP 已知次优; 随后 njhill 在 4x GB200 复现: PP2+TP2 的 16k in/1 out 配置 +47%, PP4 同配置 +65%, 持续 prefill 与混合负载 PP4 分别 +3.0%/+5.4%, 确认收益不依赖 MRV1。
- CPU 张量走 gloo 时收割闸门不可靠 (depthfirst-app, MEDIUM, 未解决) : `_reap_completed_isends` 假设 `handles[1:]` 都是设备后端句柄, 但 `tensor.is_cpu` 的张量实际走 gloo, 其 `is_completed()` 同样不可靠, 可能提前收割仍被 gloo 后台拷贝读取的 CPU buffer; 建议按 gloo/device 分开跟踪句柄, 收割时对 gloo 句柄统一 `wait()`。
- 纯 metadata 条目弹出前未 wait (depthfirst-app, LOW, 未解决) : metadata-only 条目被直接 `popleft()`, `_RetainedHandle` 及其保留的序列化 buffer 可能在 gloo 后台拷贝期间被释放, 正是本 PR 想修的 Bug 2 的残余路径; 建议弹出前先 `handles[0].wait()` (幂等且负载极小) 。
- 风格与范围控制 (njhill) : `retained` 改为 tuple、注释精简均已采纳; `vllm/config/vllm.py` 中被带入的 MRV1 相关改动按要求回退。
 - CPU 张量经 gloo 发送时收割闸门不可靠 (correctness): 合并前未修复; 当前 PP 中间张量基本为 CUDA 张量, 实际触发概率低, 但该边界问题仍遗留。
 - 纯 metadata 条目弹出前未 wait 的 UAF 残余 (correctness): 未解决; 设计上有意为之 (PR body 写明 metadata-only 条目 best-effort 丢弃), 但严格意义上仍是残留风险。
 - 性能基准需以 MRV2 为前提 (testing): 已解决: njhill 自行复现并确认收益与 MRV1 无关。
 - `retained` 改为 tuple 与注释精简 (style): 已解决。
 - 回退 `vllm/config/vllm.py` 的 MRV1 改动 (design): 已解决。

风险与影响

- 风险:
 - CPU 张量路径的提前收割风险 (未解决) : `_reap_completed_isends` 把所有 `handles[1:]` 当作可靠完成信号, 但 `tensor.is_cpu` 的张量发送走 gloo, `is_completed()` 可能过早返回 True, 存在与 PR 目标同类的 use-after-free 残余风险 (depthfirst-app MEDIUM 评论, 合并前未修复)。当前 PP 中间张量几乎都是 CUDA 张量, 实际触发概率低。
 - 纯 metadata 条目的 best-effort 丢弃 (未解决) : metadata-only 条目 (如空 IntermediateTensors) 在收割时直接弹出, 不等待 gloo 后台拷贝完成; 负载很小, 但严格意义上仍是未修复的 UAF 通道 (depthfirst-app LOW 评论) 。
 - FIFO 只在调用时收割: `_pending_isends` 依赖「下一次 `isend_tensor_dict` 调用」触发收割; 若上游长期不再调用, 已完成条目会滞留占用少量内存, 内存上界与调用频率挂钩。
 - gloo 行为假设依赖实测: `Work.wait()` 单次性、`is_completed()` 不可靠等结论来自实测与代码注释, 跨 gloo 版本或后端 (CPU/XPU 等) 行为可能不同; `parallel_state.py` 是全集群共享核心, 影响面大。

- 注释与实现不一致: `gpu_worker.py` 新增注释提到的 `isend_tensor_dict(is_async=True)` 参数并不存在于签名中, 可能误导后续维护者。
- worker 语义变更: `execute_model` 不再显式等待上一次 PP 发送完成, 改为信任 FIFO 自保留与延迟收割; 目前新增测试均为单测 (`fake_isend` 模拟), 缺少真实多卡集成测试覆盖该语义。
- 影响:
 - 用户 / 性能: 所有启用流水线并行 ($PP > 1$) 且使用 v1 执行路径的部署直接受益, `chunked prefill` 与短请求场景最明显 (作者实测 5.2%, njhill 复现 PP2+TP2 短请求最高 +47%、PP4 最高 +65%); `vllm-ascend` 等虚拟流水线场景的元数据发送死锁被消除。
 - 系统: `GroupCoordinator.isend_tensor_dict` 是分布式通信核心原语, 行为从「元数据同步 + 张量异步」变为全异步 + 自保留; 返回句柄可 `wait()` 的契约与 `send_tensor_dict` 的同步语义均保持不变, 现有调用方无需改动。
 - 团队: 为后续「异步 PP 发送」设计提供了可复用的句柄 / 保留 / 收割模式; 同时遗留两条关于 gloo 完成判定的边界问题, 需要后续 PR 在 CPU 张量或 `metadata-only` 发送进入 PP 路径前收敛。
 - 风险标记: 核心通信路径变更, gloo 完成判定不可靠, CPU 张量收割边界未覆盖, `metadata-only` 条目 `best-effort` 丢弃, 缺真实多卡集成测试

关联脉络

- PR #52068 [KV Offload] Preserve KV event metadata until final residency removal: 同属「元数据 / 缓冲在异步传输期间必须存活到完成」的生命周期问题, 与本 PR 修复的 `use-after-free` 属同一类缺陷模式, 可对照阅读。
- PR #54293 [Perf][KV Connector] Pin token_indices before non_blocking H2D in hf3fs helper: 同样处理异步拷贝前 host buffer 的存活 / 固定问题, 与本 PR 的自保留思路互补。