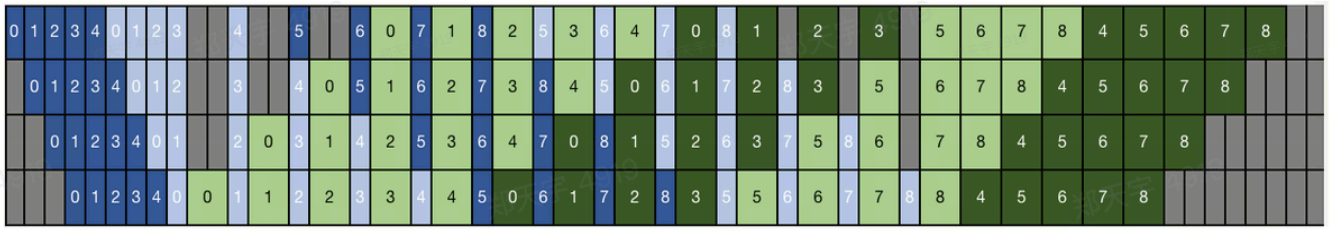
VPP(Any1f1b，显存峰值优化版)

1.针对当前Any1f1b的分析

1.方法分析

这里以pp=4，num_macro_batch=9来分析（Any1f1b的实现可以具体看此文档：Pipeline支持任意num_microbatch）。



当前的做法如下代码所示：

代码块

```
num_group = num_microbatches // pp_size
batch_remain = num_microbatches % num_group
num_group_batches = num_microbatches // num_group
group_batches = [num_group_batches + (i < batch_remain) for i in
range(num_group)]
```

首先根据pp_size对num_microbatch进行分组，并且将余数，首先按轮循的方式顺序将多余的micro分给每个group，直到分完，所以可以发现，第一个group的microbatch一定是最多的。

也可以自己配置group的分配大小，但要保证两点：

1.group_batches中的每个group的size都要大于pp数

2.前面的group的size都要大于等于后面的group

上述代码严格满足这两个要求。

2.显存分析

因为我们知道传统vpp的峰值显存为 (vpp-1)*pp+2*(pp-rank-1) + 1，**注意这里满足global_step>=warm_up_step的条件，否则就直接等于global_step个显存单位，即退化为FthenB** 前者是warmup的步数，+1是1f1b位置的一个forward。

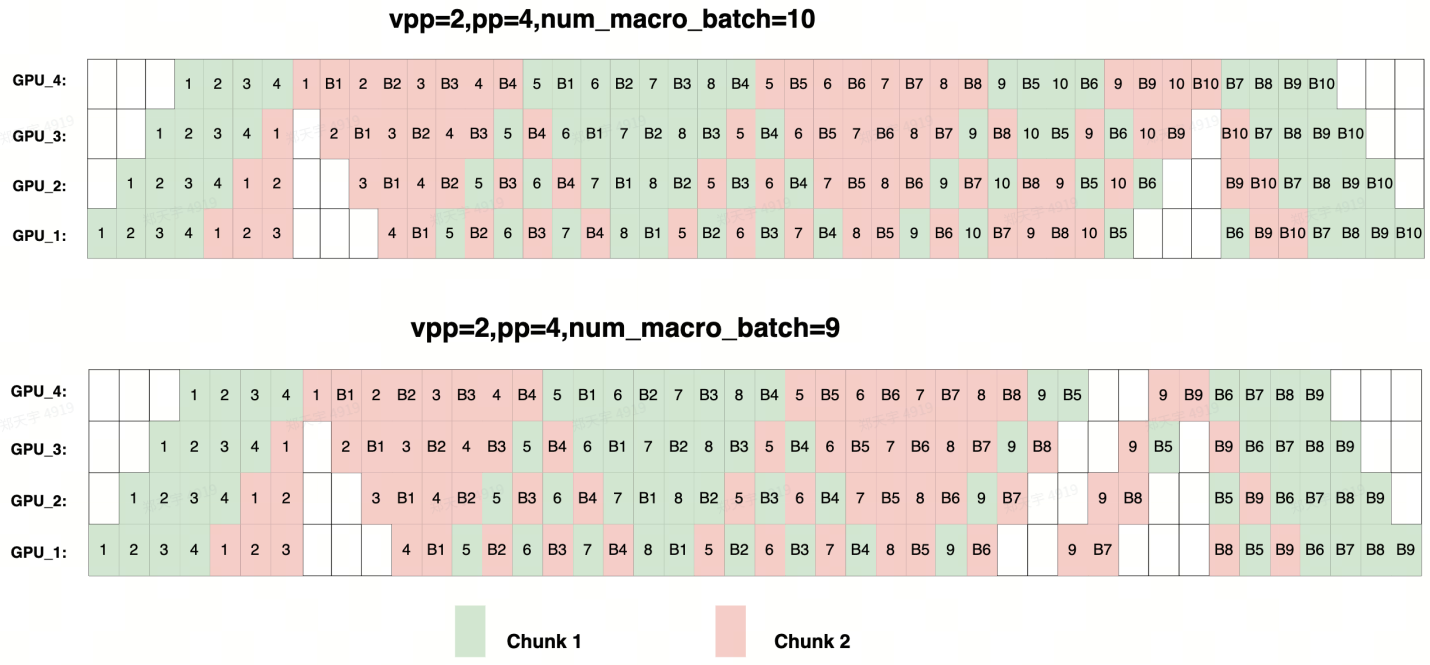
那么现在warm_up编排的个数不再是pp个，而是会多于pp个，即group_size里面的最大值，可转换为：(m // (m // p) +1)，即先除以p向下取整，得到分组数，然后用m再除以分组数，向上取证，即得到group_batches里面的最大size。最终表达式为：(m // (m // p) +1) * (v-1) + 2*(p-rank)-1, 当m=2p-1时，得到最大，即(2pv-v)个激活单位，激活单位为L/p/v，这里L表示不使用pp时，一个数据完整做一次前向和反向的最大激活显存。所以最终换算得到最大激活显存为：2L-L/p。（**所以也可以发现，p越大，越接近2L，在超大规模训练时，显存可能成为当前Any1f1b的一个瓶颈**）

2.Min_memory_Any1f1b编排设计

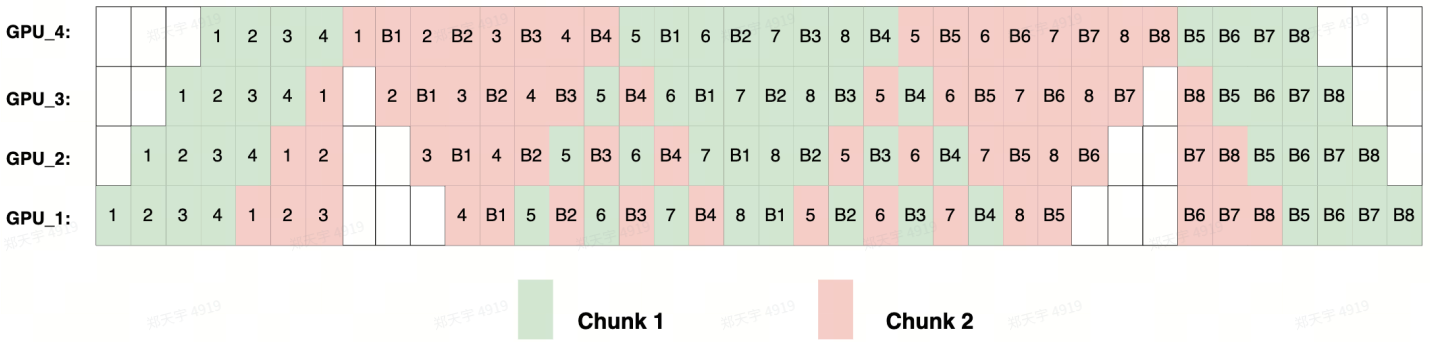
1.考虑一个通用编排方案，使得显存峰值始终为L。

此处编排均按照一个统一的方案进行编排，从而避免引入case by case的额外开销，提升编排的便捷性。注意此方案在 **2*remainder>=pp**时，相比于整除的情况(num_macro_batches % pp == 0)不会引入新的bubble时间，反之则会引入新的buble时间，新增加的bubble时间为: (pp//2-remain)* (vpp-1) 个 T(forward+backward)。**注意这里2*remain<pp 时，还能优化成不额外引入bubble，但当前仅考虑一个通用的编排方案（其实个人感觉这里我们可以控制micro_batch数，或者控制使用的GPU数来满足这个关系，相比于整除的关系，上述关系更容易满足）。**

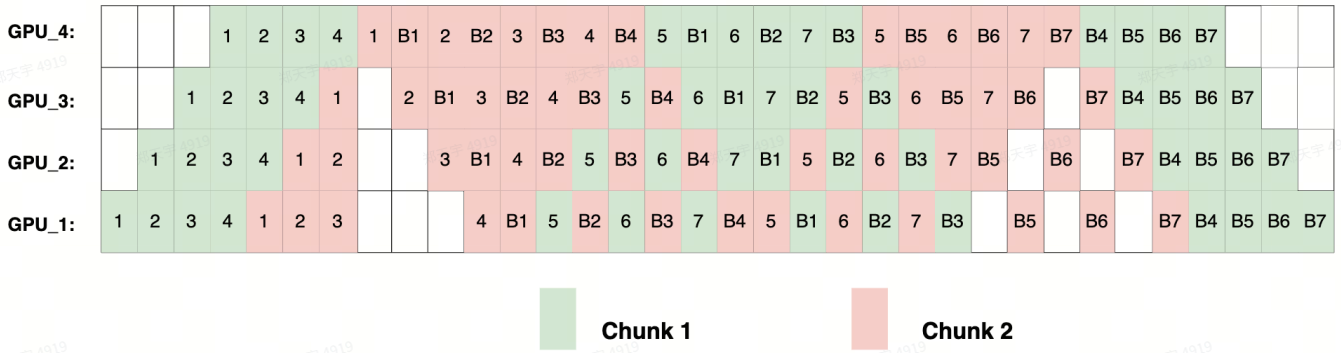
首先分析，vpp=2，pp=4下的各种编排示例：



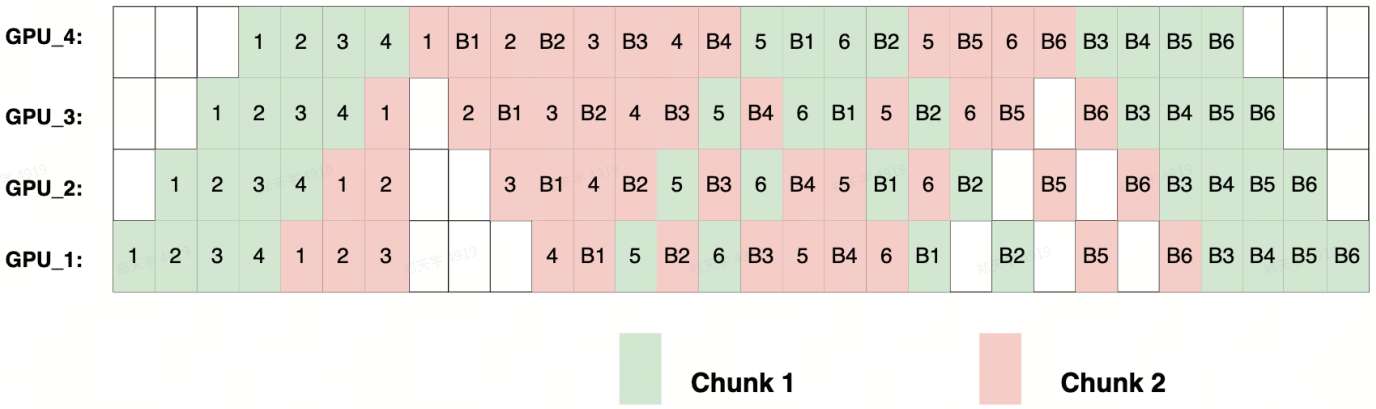
vpp=2,pp=4,num_macro_batch=8



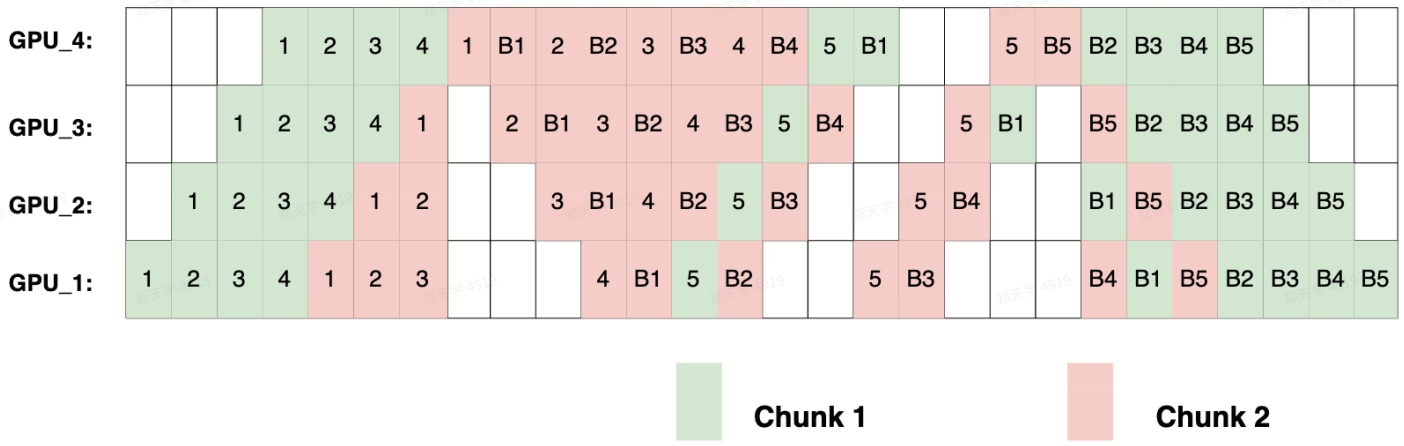
vpp=2,pp=4,num_macro_batch=7



vpp=2,pp=4,num_macro_batch=6



vpp=2,pp=4,num_macro_batch=5



编排方法总结

1. 首先获得余数部分remain，计算 num_micro_batches % pp = remain。
2. 构建fwd_groups，base_group = num_micro_batches // pp，即得到整除部分的分组数，每组分配的micro_batch的数量为 pp 个，而最后一组有余数部分组成，有remain个micro_batch，所以fwd_groups构建如下：

```
代码块
1   fwd_groups = [p] * base_group + ([remain] if remain > 0 else [])
```

3. 构建fwd_groups_vpp，指示编排时，forward需要按照什么样的规则在同一个rank上切换到不同的chunk。因为存在vpp，因此每组micro_batch都需要经过vpp个model chunk，因此fwd_groups每组micro_batch都需要编排vpp次，因此fwd_groups_vpp构建如下：

```
代码块
1   fwd_groups_vpp = each element repeated v times   # e.g. [4,3] -> [4,4,3,3] for
    v=2
```

4. 构建bwd_groups_vpp，指示编排时，backaward需要按照什么样的规则在同一个rank上切换到不同的chunk。这里为了让中间的1F1B能够完全匹配，我们需要对fwd_groups_vpp进行旋转操作，即向左旋转(v-1)步，因此bwd_groups_vpp构建如下：（需要注意的是，对于某个rank而言，fwd从chunk_0开始编排，而bwd则从chunk_v-1 开始编排，v为vpp数）

```
代码块
1   bwd_groups_vpp = fwd_groups_vpp rotated left by (v-1)  # e.g. [4,3,3,4] for v=2
```

5. 构建warm_up、1F1B(steady)、cooldown三个阶段编排的forward，backward数，这里遵循非overlap的1F1B公式，如下：

```
代码块
1     global_step    = v * n
2     warm_up_step   = min((v-1)*p + (p - rank_id - 1), global_step)
3     steady_step    = global_step - warm_up_step
4     cool_down_step = warm_up_step
```

6. 根据step和fwd_groups_vpp/bwd_groups_vpp 进行编排。

Demo

这里设num_macro_batches=n,pp=p,vpp=v，这里给出了一个num_macro_batches=7，pp=4，vpp=2的编排示例，可以直接运行得到每个rank的编排，每一个step的编排表示为：f-n_i-stage_j，表示第i个num_macro-batch在第j个vpp_stage上做forward。

代码块

```
"""
Interleaved Pipeline Parallel (VPP) Schedule Generator

Given: n (num_macro_batches), p (pp_degree), v (vpp), rank_id
Generates the full schedule for a given rank, including warmup, steady (1F1B), and cooldown phases.

Key formulas:
  remain = n % p
  base_group = n // p
  fwd_groups = [p] * base_group + ([remain] if remain > 0 else [])
  fwd_groups_vpp = each element repeated v times   # e.g. [4,3] -> [4,4,3,3] for v=2
  bwd_groups_vpp = fwd_groups_vpp rotated left by (v-1)  # e.g. [4,3,3,4] for v=2

  global_step    = v * n
  warm_up_step   = min((v-1)*p + (p - rank_id - 1), global_step)
  steady_step    = global_step - warm_up_step
  cool_down_step = warm_up_step
"""

def generate_pipeline_schedule(n: int, p: int, v: int, rank_id: int):
    """
    Args:
        n:       num_macro_batches
        p:       pp_degree (pipeline parallel size)
        v:       vpp (virtual pipeline parallel degree, chunks per device)
        rank_id: current rank (0-indexed)

    Returns:
        schedule: list of tuples (phase, op, micro_batch_id, vpp_stage_id)
        info:     dict with group lists and phase sizes
    """
    assert 0 <= rank_id < p, f"rank_id must be in [0, {p-1}]"
    assert n >= 1 and p >= 1 and v >= 1

    remain = n % p
    base_group = n // p

    fwd_groups = [p] * base_group
```

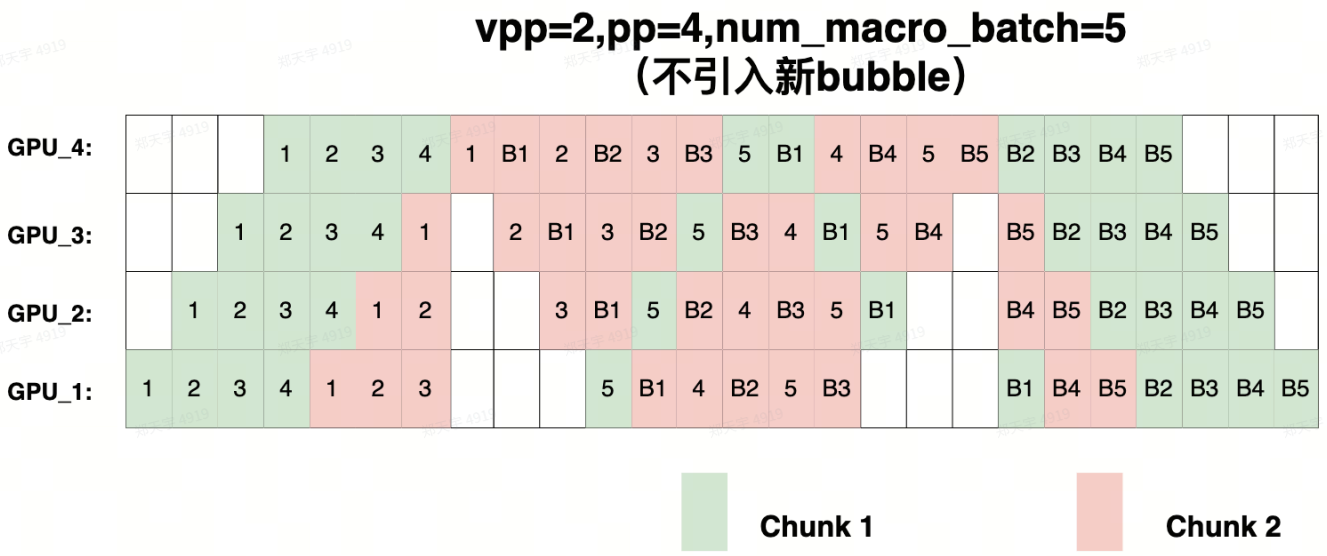
```
        if remain > 0:
            fwd_groups.append(remain)

        # Expand by v: each element repeated v times to cover all virtual stages
        fwd_groups_vpp = [g for g in fwd_groups for _ in range(v)]

        # Backward groups: rotate left by (v-1) positions
        shift = v - 1
        bwd_groups_vpp = fwd_groups_vpp[shift:] + fwd_groups_vpp[:shift]

        # ---- Build forward schedule: [(micro_batch_id, vpp_stage_id), ...] ----
        # Each group in fwd_groups_vpp maps to a stage cycling 0, 1, ..., v-1.
        fwd_schedule = []
        fwd_mb = [0] * v
        for i, g in enumerate(fwd_groups_vpp):
            s = i % v
            for _ in range(g):
                fwd_schedule.append((fwd_mb[s], s))
                fwd_mb[s] += 1

        # ---- Build backward schedule ----
        # Each group in bwd_groups_vpp maps to a stage cycling v-1, v-2, ..., 0.
        bwd_schedule = []
        bwd_mb = [0] * v
        for i, g in enumerate(bwd_groups_vpp):
            s = (v - 1) - (i % v)
            for _ in range(g):
                bwd_schedule.append((bwd_mb[s], s))
                bwd_mb[s] += 1

        # ---- Phase sizes ----
        global_step = v * n
        warmup_steps = min((v - 1) * p + (p - rank_id - 1), global_step)
        steady_steps = global_step - warmup_steps
        cooldown_steps = warmup_steps

        # ---- Assemble final schedule ----
        schedule = []
        fi, bi = 0, 0

        for _ in range(warmup_steps):
            mb, s = fwd_schedule[fi]; fi += 1
            schedule.append(("warmup", "f", mb, s))

        for _ in range(steady_steps):
            mb, s = fwd_schedule[fi]; fi += 1
            schedule.append(("steady", "f", mb, s))
```

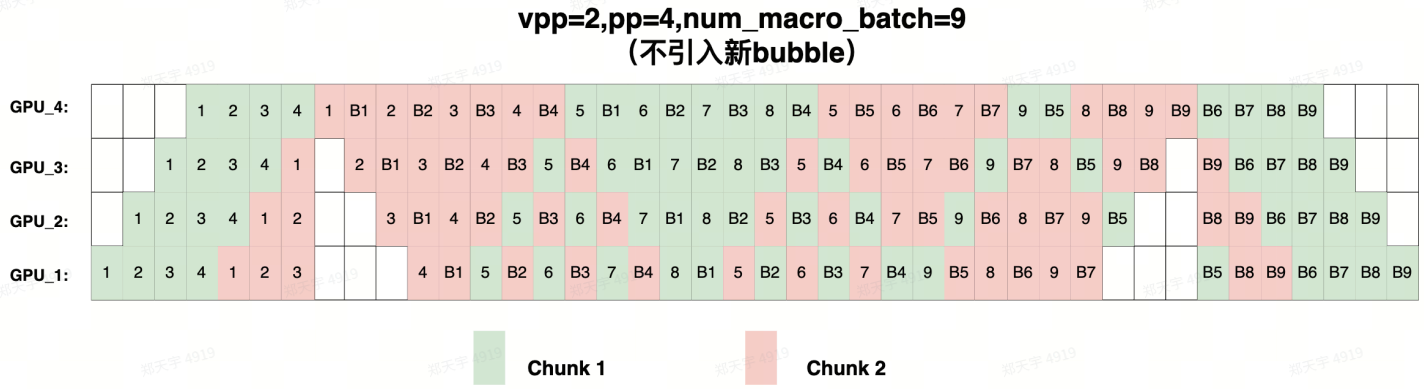
```
            mb, s = bwd_schedule[bi]; bi += 1
            schedule.append(("steady", "b", mb, s))

        for _ in range(cooldown_steps):
            mb, s = bwd_schedule[bi]; bi += 1
            schedule.append(("cooldown", "b", mb, s))

        assert fi == global_step, f"Forward index mismatch: {fi} != {global_step}"
        assert bi == global_step, f"Backward index mismatch: {bi} != {global_step}"

        info = {
            "fwd_groups": fwd_groups,
            "fwd_groups_vpp": fwd_groups_vpp,
            "bwd_groups_vpp": bwd_groups_vpp,
            "warmup_steps": warmup_steps,
            "steady_steps": steady_steps,
            "cooldown_steps": cooldown_steps,
            "global_step": global_step,
        }
        return schedule, info

    def format_op(op, mb, stage):
        return f"{op}-n_{mb}-stage_{stage}"

    def print_schedule(n, p, v, rank_id):
        schedule, info = generate_pipeline_schedule(n, p, v, rank_id)

        print("=" * 70)
        print(f"  Pipeline Schedule: n={n}, p={p}, v={v}, rank={rank_id}")
        print("=" * 70)
        print(f"  remain       = {n} % {p} = {n % p}")
        print(f"  base_group   = {n} // {p} = {n // p}")
        print(f"  fwd_groups       = {info['fwd_groups']}")
        print(f"  fwd_groups_vpp   = {info['fwd_groups_vpp']}")
        print(f"  bwd_groups_vpp   = {info['bwd_groups_vpp']}")
        print(f"  global_step  = {info['global_step']}")
        print(f"  warmup_steps = {info['warmup_steps']}  (forward only)")
        print(f"  steady_steps = {info['steady_steps']}  (1F1B)")
        print(f"  cooldown_steps = {info['cooldown_steps']}  (backward only)")
        print("-" * 70)

        current_phase = None
        step_idx = 0
        for phase, op, mb, stage in schedule:
            if phase != current_phase:
                current_phase = phase
                phase_label = {
```

```
                "warmup": "WARMUP (forward only)",
                "steady": "STEADY (1F1B)",
                "cooldown": "COOLDOWN (backward only)",
            }[phase]
            print(f"\n  [{phase_label}]")

        tag = format_op(op, mb, stage)
        print(f"    step {step_idx:>3d}:  {tag}")
        step_idx += 1

    print("\n" + "=" * 70)

    # Validation: each (micro_batch, stage) pair appears exactly once for F and B
    fwd_pairs = [(mb, s) for (ph, op, mb, s) in schedule if op == "f"]
    bwd_pairs = [(mb, s) for (ph, op, mb, s) in schedule if op == "b"]
    expected = {(m, s) for m in range(n) for s in range(v)}
    assert set(fwd_pairs) == expected, "Forward coverage mismatch!"
    assert set(bwd_pairs) == expected, "Backward coverage mismatch!"
    assert len(fwd_pairs) == len(expected), "Duplicate forwards!"
    assert len(bwd_pairs) == len(expected), "Duplicate backwards!"
    print("  [PASS] All (micro_batch, stage) pairs covered exactly once.\n")

def print_all_ranks(n, p, v):
    print(f"\n{'#' * 70}")
    print(f"#  All ranks for n={n}, p={p}, v={v}")
    print(f"{'#' * 70}\n")
    for rank in range(p):
        print_schedule(n, p, v, rank)

if __name__ == "__main__":
    import sys

    if len(sys.argv) == 5:
        n = int(sys.argv[1])
        p = int(sys.argv[2])
        v = int(sys.argv[3])
        rank_id = int(sys.argv[4])
        print_schedule(n, p, v, rank_id)
    elif len(sys.argv) == 4:
        n = int(sys.argv[1])
        p = int(sys.argv[2])
        v = int(sys.argv[3])
        print_all_ranks(n, p, v)
    else:
        print("Usage:")
```

```
178          print("  python pipeline_schedule.py <n> <p> <v> <rank_id>   # single
     rank")
179          print("  python pipeline_schedule.py <n> <p> <v>             # all
     ranks")
180          print()
181          print("Examples:")
182          print("  python pipeline_schedule.py 7 4 2 0")
183          print("  python pipeline_schedule.py 9 4 2")
184          print()
185
186          print("=" * 70)
187          print("  Demo: n=7, p=4, v=2")
188          print("=" * 70),
189          print_all_ranks(7, 4, 2)
190
```

2.针对2*remain<pp的编排优化





可以看到上述的编排方式，均可将2*remain<pp的编排同样转换为不引入新bubble的编排，且峰值显存仍然是L，并且其实是存在编排处理规律的。

~~**TODO：**总结2*remain<pp时，不引入新bubble的编排方式。~~

见 VPP(Any1f1b，显存峰值优化版)

3.显存优化性能分析

以下v表示vpp_size，p表示pp_size，m表示micro_batches:

当前Any1f1b：

峰值显存公式：((m // (m // p) +1) * (v-1) + 2*(p-0-1)+1) * L/p/v

最大峰值显存：2L-L/p (m=2p-1时最大)

Min_Memory_Any1f1b:

峰值显存公式：((v-1)*p + p -1 +1) * L/p/v = L

最大峰值显存：L

当p非常大时，**最大峰值显存**优化前为2L，优化后为L，降低了50%。

3.通信编排分析

1.原始Any1f1b的通信编排分析

1.warmup阶段

pp0：需要先安排min(group_batches[0],pp_size)个None，用来表示，这几个forward，不需要recv数据，是直接拿到初始数据做的，而后续的forward，只要发现vpp_rank不是最后一个，则说明，vpp_rank+1的rank一定要做recv，所以就将fwd_recv_chunk_ids中的元素更新，并且

model_chunk_id+1，表示vpp_rank+1的rank需要做recv，在warmup阶段，直接根据forward_k查表。

对pp0的特殊处理：

并且如果开启了overlap，其他pp_stage都是会做一个偏移，主要是为了计算i的时候，通信i+1，而pp0则不需要，因为它的fwd_recv_chunk_ids已经做了特殊处理。

其他pp：直接用fwd_recv_chunk_ids查表，但是第0步forward，发recv0和1，而后续的发recv（k+1）

2.在1F1B阶段

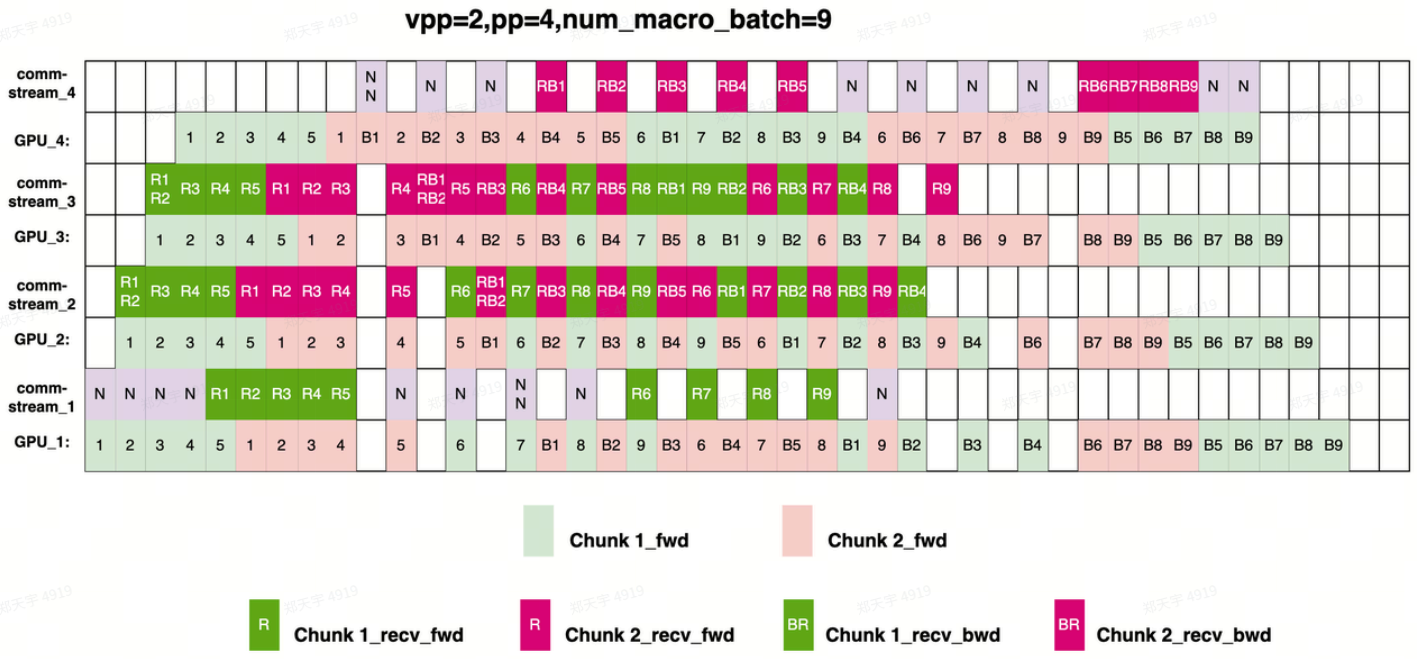
pp0和其他pp都一样，第k步forward做第k+1步的recv，这里也直接查fwd_recv_chunk_ids[k+1]的结果即可。（但需要注意，pp0的表是做了偏移的）

1F1B阶段，这里主要通过prefetch_one_tensor来判断是否要多编排一个recv，只有pp0需要，而num_warmup_microbatches是针对pp=1的特殊边界处理。并且注意，最后一个forward无需再同时接收数据。

3.cooldown阶段

和warmup阶段是对称，pp_last_stage做特殊处理，而其他pp，使用bwd_prefetch_one_tensor做位移

4.通信总览图



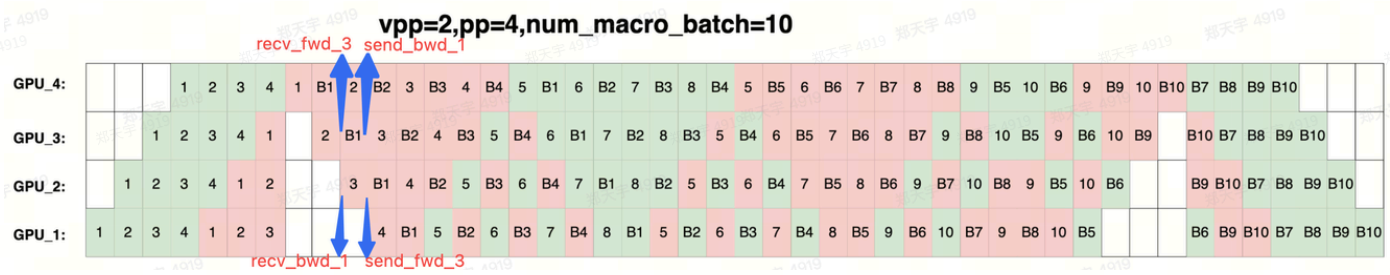
这里在last_pp_stage上只画了backward,在first_pp_stage只画了forward，中间的pp_stage同时画了forawrd和backward，比较直观，可以看到，在做overlap的时候，理念是，一旦当前rank接收数据所需要的那个rank计算完成，就立即调用recv提前把数据接收过来，因此注意，图中的通信和计算，实际还会有一些偏移，但能够在下一个计算之前，完成通信，从而实现计算和通信的overlap。

注意这里画的只是发起创建recv op的时刻，连接依赖关系的时候，有一些区别，比如图中GPU2的1只和R1关联，即R1做完，就可以立刻做1的forward计算，所以发起recv op ，和构建 recv和compute的依赖关系是有区别的。

2.Min_memory_Any1f1b的通信编排分析

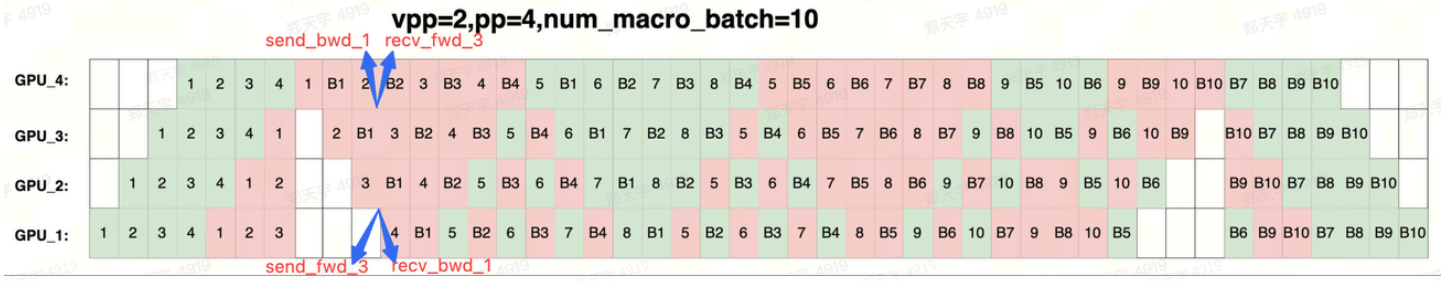
1.当前通信的问题分析

1.fwd和bwd的recv提前



保持**与原始Any1f1b同样的通信**，此时因为GPU_3提前做了recv，而recv_fwd和send_bwd共用一个stream，所以需要先等到接收到micro_batch_3的数据，才会继续send backward的数据；而GPU_2先recv_bwd_1再send_fwd_3，需要先接收到micro_batch_1的backward数据，才会发送micro_batch_3的数据，**两边都在等待对方send，导致hang住。**

2.fwd和bwd的recv都不提前



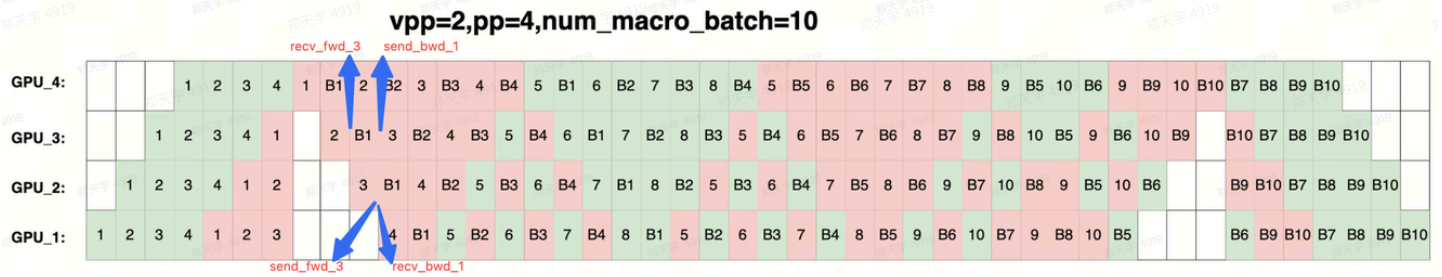
再考虑一种情况，**我们不将recv或者send提前，都在计算前调用，此时仍然会hang住**，如图，GPU_3和GPU_2同时向对方发送数据，都在等待对方接收，导致hang住。

2.方案1: 依赖异步通信解决hang

优势：可以用一种通用通信编排方案适配所有pp和micro_batch的数量组合，但可能存在异常hang住的风险，无法开启CUDA_LAUNCH_BLOCKING来排查hang异常。

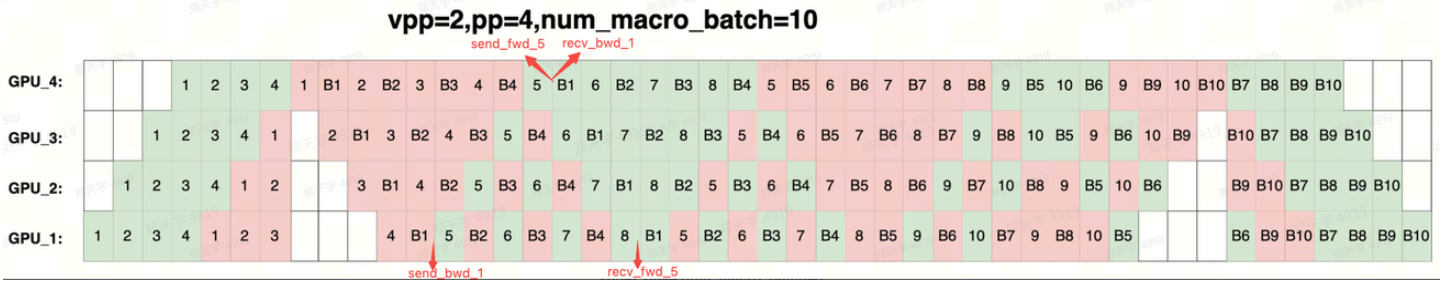
解决方法如下：

1.fwd的recv提前，而bwd的recv不提前(解决非（first_stage与last_stage）之间相邻rank间的通信死锁问题)

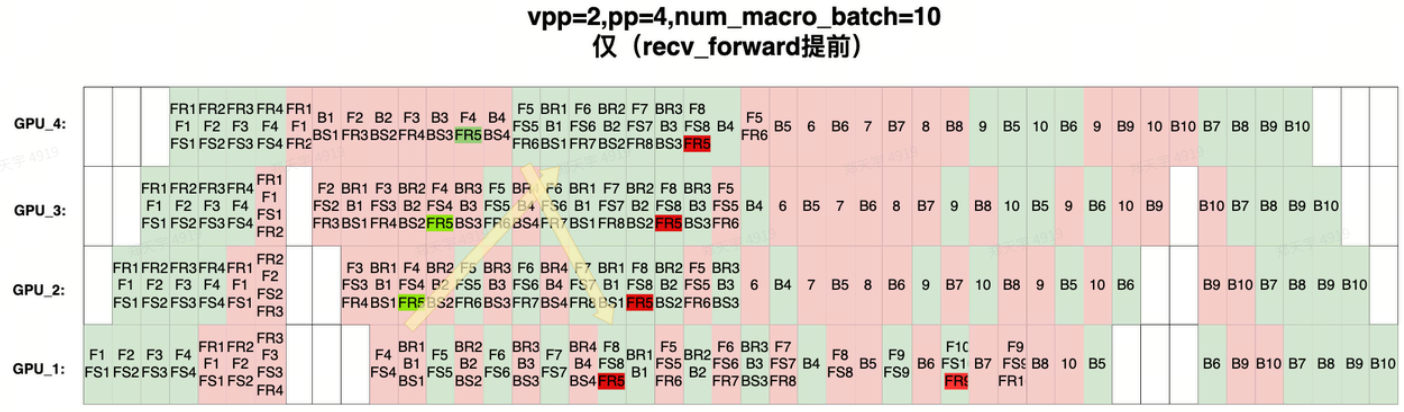


可以看到上述通信的排列，因为此时forward与backward刚好是交叉排列的（即在同一个timeline，不同相邻rank交替编排forward和backward），所以这样做的目的是让forward和backawrd交错通信，即backward计算后的结果对应的send操作，排在下一个forward计算需要的recv通信后面，这样就可以先接收forward数据，再发送backward的数据，解决中间rank的死锁，但**first_stage与last_stage存在特殊通信死锁**。

2.first_stage与last_stage的特殊死锁问题

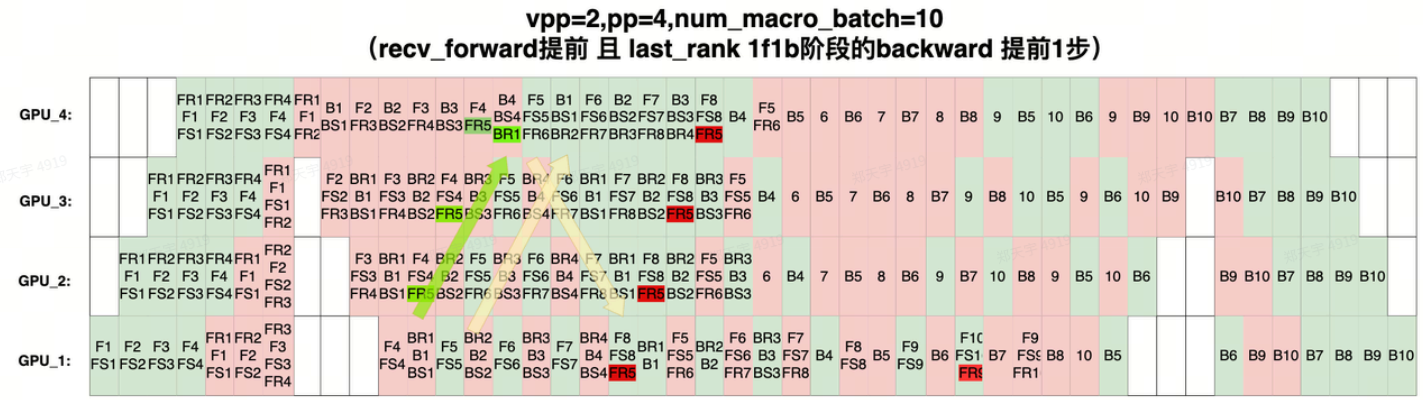


这时候虽然解决了rank i 和rank i+1相邻rank之间的冲突，但是此时，rank0和rank pp-1 仍然存在冲突，如图所示，GPU1发送了bwd的数据，等待GPU_4接收，而GPU4需要先等待GPU1将send_5的数据接收，这就导致存在了一个循环hang的问题，为了更清楚体现出现通信问题的位置和过程，在下图展现了具体通信hang住的流程(仅画出部分通信)。

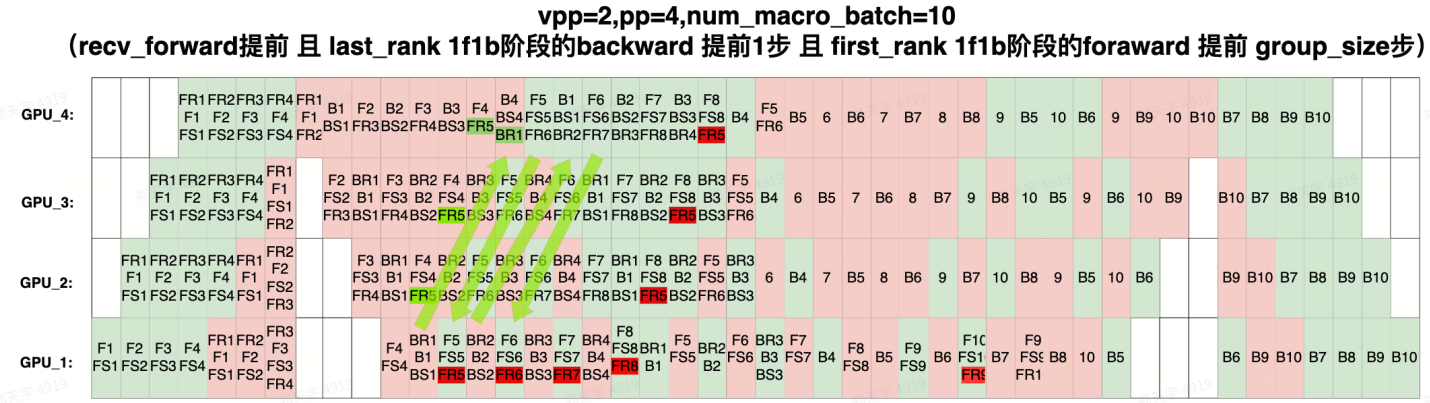


对于这个问题，我们分析可知，这里造成死锁的主要原因在于，最后一个rank和第一个rank通信，即切换vpp_stage的过程，并且**注意一个性质：最后一个rank的recv_backward只会来自于rank1，第一个rank的recv_forward只会来自于rank4，且first_stage的初始forward不需要通信数据，而last_stage的初始backward不需要接收通信数据**。而这里主要是forwrad_send和backward_send后都在等待对方接收造成的hang，而我们发现，其实recv_bwd_1早就可以执行拿到

结果了，所以我们仅在**最后一个rank的1F1B阶段做recv_bwd的提前**，这里由于前面性质的原因，是不会出现hang住的现象，且**解决了上图的死锁问题，但是这里仍然会存在hang住的问题**，如下：



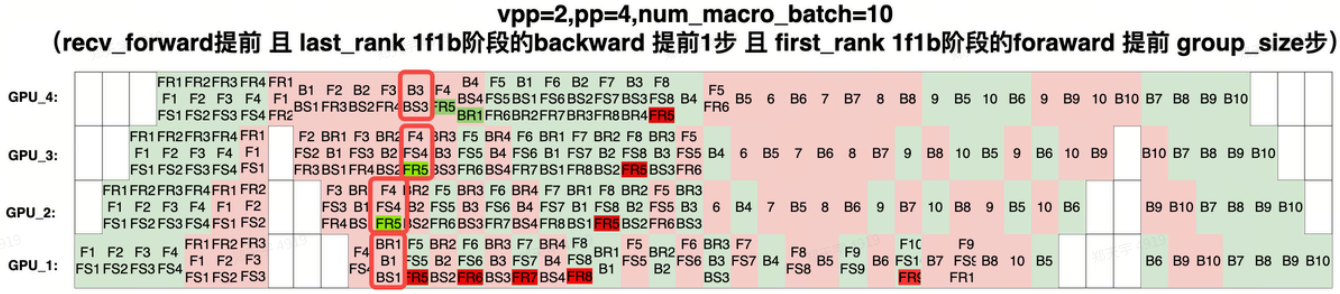
可以看到上述方法仍然存在死锁，其主要原因在于，forward的recv时间太晚了，其实当我们first rank 做了一个forward，且当前不是last_vpp_stage，则一定会从last_rank接收forward的数据，进入到下一个vpp_stage，因此我们可以在做完forrward后，就立即发起，编排效果如下：



此时，first_rank和last_rank的通信，是一个平行状态，就不会再hang了，这里的操作，即让first_stage 在1f1b阶段的forward不再是提前1步，而是提前group_size步，即vpp_stage提前发起vpp_stage+1的 recv_forward 的通信，这里使用group_size的原因在于，我们对于余数会单独放一个group做处理，因此需要根据group_size来提前最后一组forward的步数，当然也可以直接在代码中硬编码，因为只有最后一组的group_size不一样。

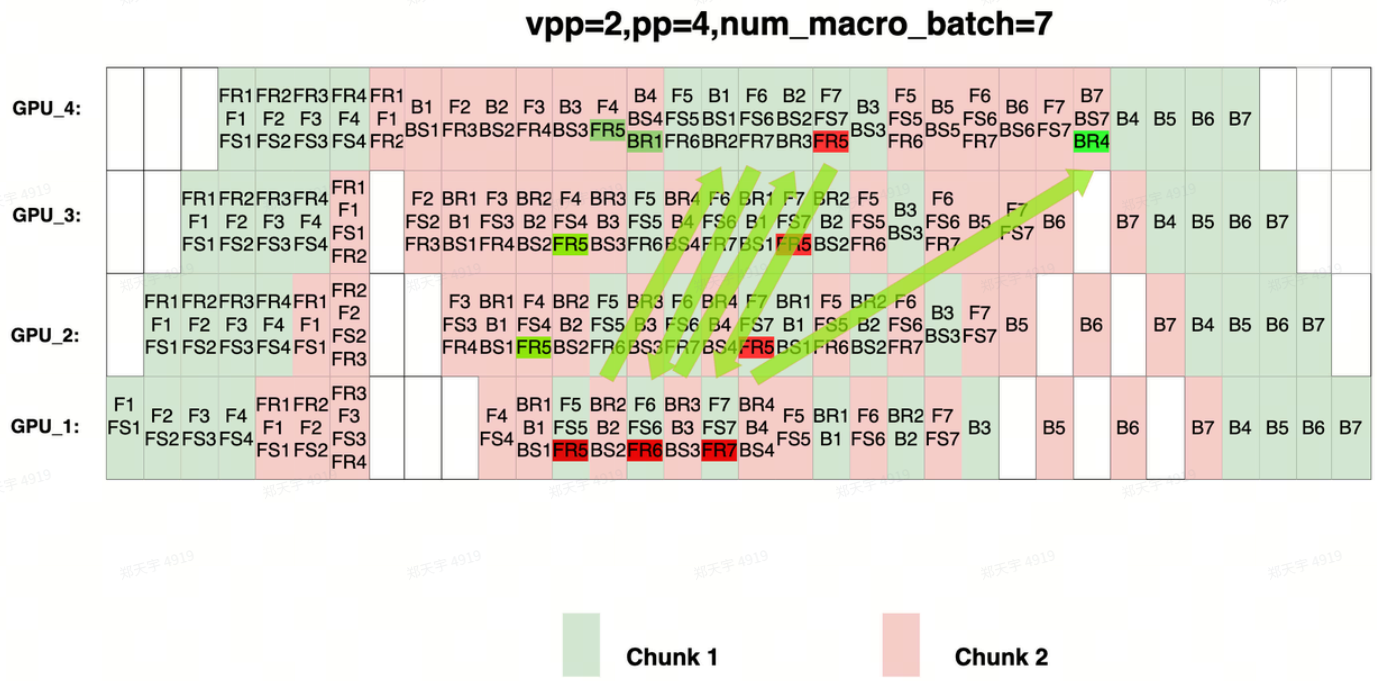
注意：我们上述对first rank和 last rank的改动，last rank 只改动了recv_bwd的编排，只与first rank有关，所以不会产生相邻rank之间新的通信死锁，同理对first rank也是，这里与性质对应。

3.说明无法开启CUDA_BLOCKING的排查的原因：



考虑一个问题，如果开了CUDA_BLOCKING，即同步阻塞的通信，则发起通信后，会等待通信完成后再launch下一个kernel，这时候，GPU1 send backward 1 数据的通信会阻塞GPU1 接下来的forward计算，即使二者没有关联，此时其余的GPU无法recv forward 5的数据，而最后一个GPU 4也会卡在send backawrd 3的位置。

当前方法在pp模拟器下任意pp_size和num_micro_batches测试通过，实际跑训练遇到如下问题：



~~TODO: 当前划分为[4,3],[4,4,3]的组，会hang住，目前还未定位到原因。由于当前的方法，主要是通信利用异步isend，irecv来实现recv send的编排错位，从而解决通信hang的问题，因此无法开启cuda_blocking来定位，需要后续再分析。~~

异步通信Min_Memory_Any1f1b hang的问题解决

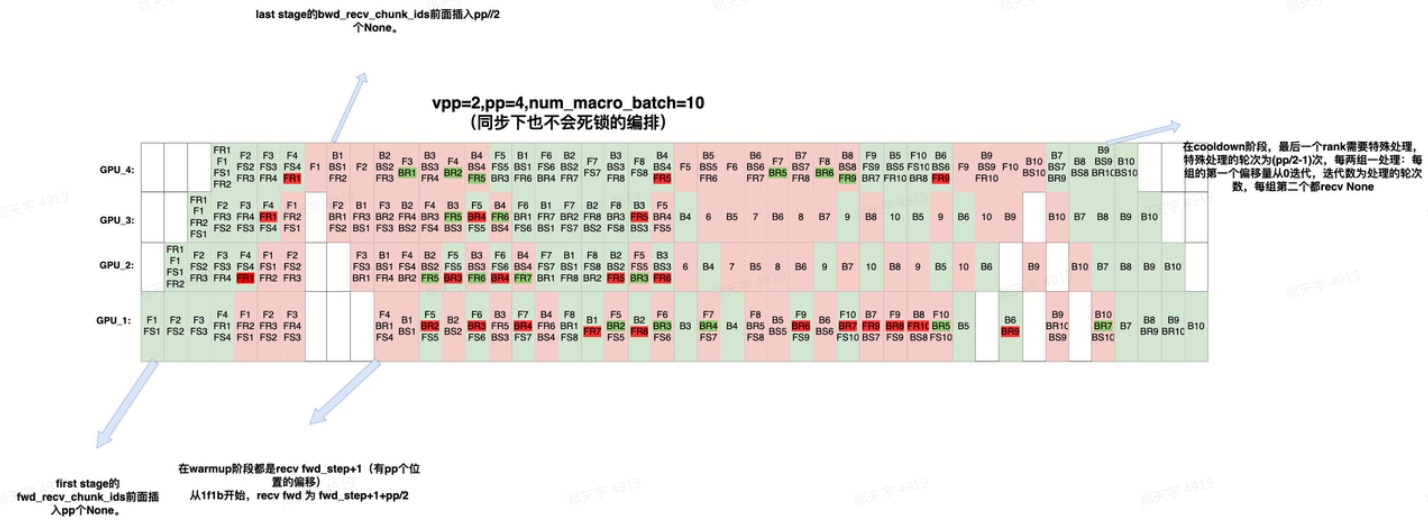
~~4.从stream上下手，使用batch p2p（当前不考虑）~~

从stream上解：一种简单的做法是直接把stream分成四份，让bwd_send，fwd_send，fwd_recv，bwd_recv各占一个stream，这样会出现带宽抢占（另一种方法是，编排fwd_recv和backward_send，而不再是提前fwd和bwd的recv。）：

3.方案2: 不依赖异步通信解决hang

优势：即使开了CUDA_LAUNCH_BLOCKING，同样可以正常执行，方便排查问题，没有hang的风险，但目前方案需要设计两类，一类支持remainder*2>=pp，另一类支持remainder*2<pp的情况。

1.remainder*2>=pp



这里与异步通信解决hang的区别在于对first_rank和last_rank的特殊处理，采用即发即收的方式，如编排图所示，first_rank与last_rank在通信过程中，每一个send在同一时刻会有对应的recv接收，注意，因为数据并不是立即用到，因此在这里也是异步处理，来overlap时间,主要是加快first和last stage的处理速度。

这里pp//2的2，指的是1个forward，1个backward，所以与vpp，reaminder是无关的。

具体逻辑如下：

一、remainder>=pp//2

首先要assert一下，确保remainder*2>=pp。这里需要严格对齐，即remainder=pp%2,remiander*2>=pp。（原因见 VPP(Any1f1b，显存峰值优化版)，针对remainder<pp//2的分析）

二、在Any1F1BSchedule中

针对first_stage的fwd_recv_chunk_ids，按照非enable_min_memory的方法，处理fwd_recv_chunk_ids（但注意，直接加pp_size个None，不需要跟非enable_min_memory方法那样选min），同时按照非enable_min_memory的方法处理bwd_recv_chunk_ids，但这里同样要特殊对待数量，只创建pp//2个None即可。

三 、编排阶段

在编排阶段，首先都需要先_fwd_recv(0)，这里调用send_forward_recv_forward来处理通信，然后开始warmup阶段。

1. warmup阶段，每次recv_k都等于k+ 1，k为循环num_warmup_microbatches时的循环数，这里先做forward计算，然后同样调用send_forward_recv_forward来处理通信。

接着进入steady阶段，也就是1f1b阶段，这里顺序和非enable_min_memory就不一样了。

2. 1f1b阶段，每轮迭代，先做forward计算，然后recv_bwd的步数就等于k，k为循环num_microbatches_remaining时的循环数，调用send_forward_recv_backward，然后计算

backward，然后计算recv_fwd的步数：

3. 如果是parallel_state.is_pipeline_first_stage(ignore_virtual=True)，则recv_fwd=forward_k+1+pp/2；否则，就是recv_fwd=forward_k+1，然后调用send_backward_recv_forward。

4. cooldown阶段：

都先做一次backward的计算，然后计算recv_bwd的recv_k，这里需要区别对待：

i. 如果是parallel_state.is_pipeline_last_stage(ignore_virtual=True)，设置一个post_recv_step=0,然后设置一个count=pp//2-1，表示要特殊处理的组数，然后开始迭代，每次先计算一次backward，然后计算recv_back_step=recv_k-post_recv_step，然后做一次_bwd_recv，里面就用send_backward_recv_backward即可，然后接下来一次，的recv，需要直接recv None，这里可以直接用_bwd_recv(0)来固定，然后post_recv_step+1，count--，接着继续，做上述操作，即每2组处理一次，每次其实主要处理第二次的这个recv固定为_bwd_recv(0)即None，并且更新post_recv_step和count。当count为0以后，post_recv_step就固定了，每次按recv_back_step=recv_k-post_recv_step处理即可。

ii. 如果是非parallel_state.is_pipeline_last_stage(ignore_virtual=True)，则在进入backward之前，要先_bwd_recv（num_microbatches_remaining），然后再进入backward：

每次迭代，先计算backward，然后recv_k=此时的k+1,k也是在(num_microbatches_remaining, total_num_microbatches)循环的范围里，然后调用_bwd_recv。也是用里面的send_backward_recv_backward即可。

所以注意，进入_bwd_recv和_fwd_recv前，都要判断一下recv_k是不是已经大于等于total_num_microbatches了，是的话，就直接都是recv None对应的处理方式，因为说明不需要再提前recv了。

四、p2p的奇偶交错

因为这里在1f1b阶段是send_bwd_recv_fwd和send_fwd_recv_bwd，因此需要新增一组奇偶交错的逻辑，如编排图所示，奇偶编排遵循：

i. 奇数：

recv_fwd -> recv_bwd -> send_fwd -> send_bwd

ii. 偶数：

send_fwd -> send_bwd -> recv_fwd -> recv_bwd

五、编排理论解释

i. fwd_recv_chunk_ids和bwd_recv_chunk_ids说明

forward：因为首个group的forward一定确定为pp_size，若小于pp_size会退化为标准的any1f1b。因此一定有pp_size个forward 是不需要recv，因此插入pp_size个None

backward:在原来的any1f1b编排时，同样会先插入pp_size个None,与forward原因相同。然而因为我们要做到即发即收，所以当经过pp_size//2个backward后，此时由于1f1b的编排，第一个backward已经到达第一个rank，下一对FB可以开始接收第一个backward从rank0发到最后一个rank的backward数据了，实现backward的即发即收。

其余处理与常规any1f1b相同，即first_stage的chunk_id为vpp-1的，recv都设置为None，而last_stage的chunk_id为0的，recv都设置为None。

ii. 编排阶段说明

在warmup阶段，以及1f1b阶段，采用奇偶交错的通信方式，来解决相邻rank间通信hang的问题（除第一个rank和最后一个rank），因此不需要特殊处理，recv_fwd为k+1，recv_bwd为k。

因为前面通过设置chunk_ids，我们已经保证了让backward的数据在first_rank和last_rank之间即发即收，现在还需保证forward在chunk切换时也能即发即收，而首先通过对fwd_recv_chunk_ids的处理，forward的recv其实延迟了pp_size步，而recv_k+1，将其变为了pp_size-1步，这时，我们只需要加pp_size//2,即变为了pp_size//2-1步，这时候我们其实可以发现，在1f1b阶段1步对应1个forward和1个backward，经过pp_size//2步，forward即可从第一个rank传入到最后一个rank计算完毕，因此剔除在first_rank的forward这一步计算，就刚好是pp_size//2-1步。从而实现forward在第一个rank和最后一个rank，切换chunk时的即发即收。

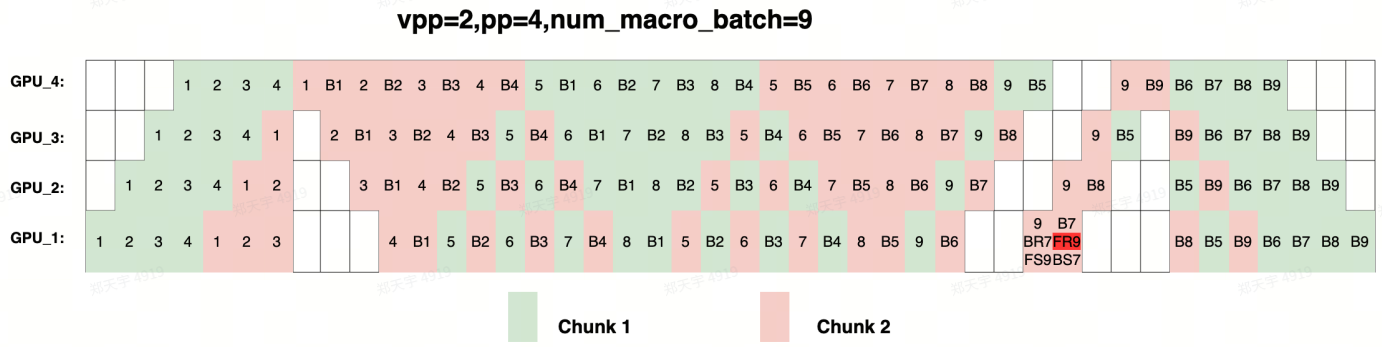
iii. cooldown阶段

为了在cool down也能实现即发即收，我们会发现最初我们设置的是pp//2个none，那么我们会发现原来的**bwd_recv_chunk_ids**其实长度增加了pp//2，而在1F1B结束时，一定存在pp//2个backward的数据还需要接收，因此我们在此处做特殊处理，注意特殊处理的轮数为(pp//2-1)次，每两个一组，原因是以两两backward模拟1F1B的数量，只在每组的第一个backward做recv，并且增加偏移量（增加偏移量是因为此时处理的都是backward，需要跳过模拟用到的F——其实对应的是B），从而实现在cooldown阶段也即发即收。

六、相关MR

https://code.byted.org/seed/mariana/merge_requests/14248/checks?to_version=10&ck_commit_sha=58e1bf3c351a&check_run_id=768732584489538

2.remainder*2<pp



首先我们分析一下，为什么remainer*2<pp_degree的时候,无法用上述的编排方法，因为在**pp >= remainder*2**的方案有提到，对于rank0来说，现在是让forward在计算后的`pp_size//2-1`步就被接收，如图，由于remainder小于`pp_size//2`，所以rank0的bubble期间对应的位置，本应该有一对FB执行并在此期间接收chunk 1 的micro_batch_9的forward，但是没有接收，导致，chunk2的micro_batch_9在计算前并未收到数据（注意我们的forward_recv在backaward计算结束后执行），所以对于**remainder*2<pp** 的情况，我们不能再按照原来的方式提前recv，而对应我们上图的编排，每个vpp_stage的forward做完会立即进入下一个vpp_stage，因此我们只需要判断group_id为余数组的id时，在计算forward的时候之前接收forward的数据即可。

那么应该提前多少步，上面的例子比较少，无法直接看出来，我们继续分析一些例子:

pp_size=6,vpp_size=2,micro_batches=8:

此时是需要在上一个vpp_stage计算完成后，延迟2个step接收并计算vpp_stage+1

pp_size=6,vpp_size=2,micro_batches=7:

此时是需要在上一个vpp_stage计算完成后，延迟1个step接收并计算vpp_stage+1

pp_size=8,vpp_size=2,micro_batches=9:

此时是需要在上一个vpp_stage计算完成后，延迟1个step接收并计算vpp_stage+1

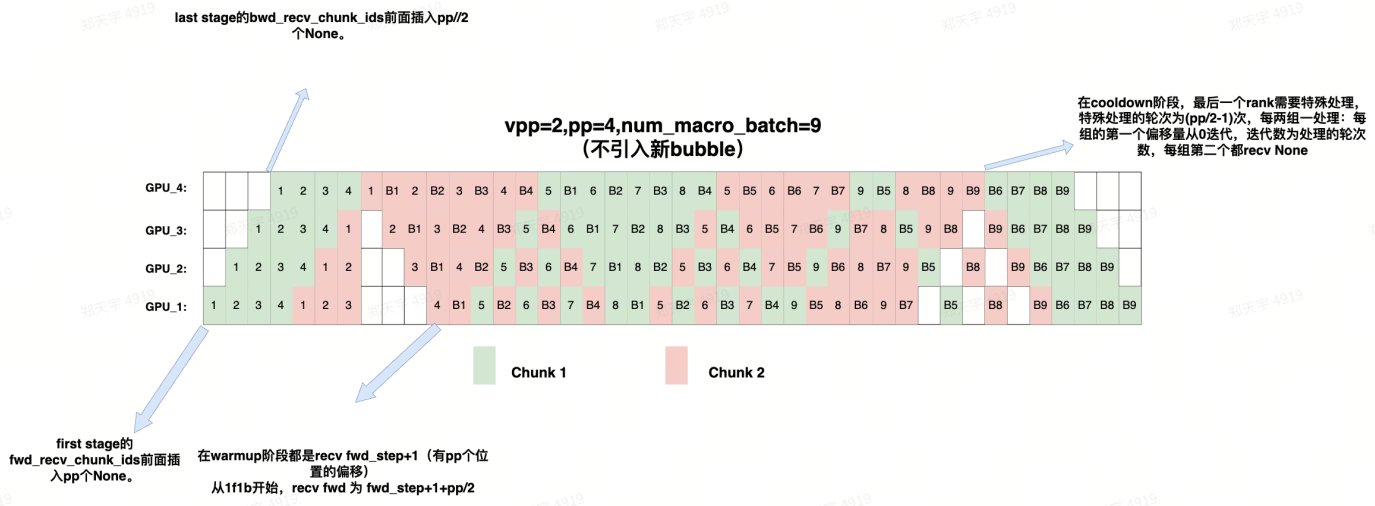
我们可以归纳出在group为余数组时，forward其实只需要延迟remainder个step接收，即可适配。注意上面的例子有一些未遵循 `remainder>=pp_size` 下的编排。主要是为了在不用启发式算法优化前，即按顺序处理micro_batch时，按最低bubble率来编排。若遵循，则bubble可能更大，当然这并不影响我们探讨对forward的特殊处理，同时若遵循 `remainder>=pp_size` ，backward过程也会出错，如下举一个例子（这里例子也可以直观看出来，若遵循 `remainder>=pp_size` 编排，bubble会增加的更多）：

pp_size=4,vpp_size=3,micro_batches=5:

遵循 `remainder>=pp_size` 的编排：

注意此时为了遵循remainer*2>=pp_degree时的编排，即forward为[4,4,4,1,1,1]的分组，则backward为[4,1,1,1,4,4]的分组，则会发现backward在此时因为我们插入了pp//2个None，所以紫色B0才刚刚将绿色B3的数据接收完，而黄色B0未被接收，此时backward计算缺失数据就会出错。我们当然可以仍然按照上述顺序编排的最低bubble率的编排方式来做，如下图所示：

但是我们其实可以看到，在**remainder*2<pp** 时，不仅bubble较大，且通信编排非常复杂，需要特殊处理，无法统一编排，所以我们希望可以找到一种方法，将这种高bubble率的编排降低到低bubble率（即之前提到的，不新增bubble），如下图所示：



我们也能发现，通过消除1F1B阶段产生的bubble，即不引入新buble时的编排，完全符合remainder*2>=pp时的通信编排方法，因此，我们只需要针对remainder*2<pp的情况，归纳一个统一特殊编排处理的模版，即可复用通信编排。而这里需要处理的是末尾余数部分的micro_batch，需要将其forward非last stage的计算提前处理，才能防止其延迟关键路径的时间。启发式算法见 启发式算法优化remainder*2<pp的bubble

3.支持overlap(warmup,1F1B,cooldown)

Min_Memory_Any1F1B overlap(warmup&cooldown)

4.MR链接：

https://code.byted.org/seed/mariana/merge_requests/14248/checks?to_version=7&ck_commit_sha=9349ec5b47ee&check_run_id=768152543696320