HyDra: Demystifying and Taming Dynamic Context Parallelism at Production Scale

Zihao Fan[†‡], Yunzhuo Liu[‡], Bo Jiang[†], Changgang Zheng[§], Lin Zheng[‡], Ray Ying[‡], Key Zhang[‡]
[†]Shanghai Jiao Tong University [‡]Tencent Hy [§]Nanjing University

Abstract

Long-context training runs on sequences whose lengths span orders of magnitude, and dynamic context parallelism (DCP) gives each sequence its own CP degree. Existing DCP systems either do not scale or perform poorly on mainstream models, leaving Megatron-Core (Mcore) DCP as the only option at production scale. Mcore DCP, however, sizes each degree to fit memory, which grows linearly with length while attention grows quadratically, so comparable token counts hide unequal computation. In our production 256K-context training job on more than **11K** GPUs under Mcore DCP, per-rank microbatch times differ by up to 5× at similar token counts. The skew leads to a 46% pipeline bubble and a 13% data-parallel bubble. We present HyDra, a scalable load-driven DCP system. Its scheduler balances computation by pulling every rank toward one load target, and balance in turn makes that target solvable in closed form. It places sequences with lazy heaps rather than whole-pool scans. That balance asks for CP degrees larger than memory requires, so its CP engine nests an inner Ulysses group in a shallow outer ring, letting a higher degree lower computation and communication together. Evaluation at both scales shows consistent gains. On a **512**-GPU testbed, HyDra raises throughput over Mcore DCP by 1.18× on average at 32K context and 2.48× at 256K. On a **2,048**-GPU production job, it shrinks the pipeline bubble from 36% to 14%, cuts scheduling time by 2.3×, and raises throughput by 1.10–1.43× (avg. 1.25×) over Mcore DCP and 1.33–1.90× (avg. 1.59×) over static CP.

1 Introduction

Long-context capabilities are increasingly important for frontier language models [2–5, 27, 29, 43]. The data used for long-context training, however, do not form a uniform stream of long sequences. Real corpora mix short documents with a small fraction of very long ones, so sequence lengths span orders of magnitude [13, 41, 42]. Dynamic context parallelism (DCP) is an effective way to train on such data, because it adapts the context-parallel (CP) degree [15, 28] to each sequence instead of fixing it for the whole job [13, 41].

In practice, however, every existing DCP system falls short in a different way [13, 17, 23, 24, 41]. The first limitation is scalability. FlexSP [41], Hydraulis [23], and DCP [17] all plan every iteration by solving a global optimization problem, which becomes prohibitively expensive at large scale. The second limitation is performance. ByteScale [13] does scale, but its ring-based CP [28] is designed for grouped-query attention (GQA) [1], where the few KV heads keep the exchange cheap. Mainstream models now use multi-head latent attention (MLA) [6–8], which materializes K and V for every head during training, so the same exchange moves far more data and performance degrades sharply at large scale.

What remains is the DCP shipped in Megatron-Core (Mcore) [24], which is today the only implementation usable at production scale. Its scheduler, however, sizes the CP group of each sample from a memory budget, namely the largest number of tokens a rank can hold. Memory grows linearly with sequence length while attention computation grows quadratically, so a CP size chosen to fit memory leaves the computation imbalanced, and the imbalance worsens as sequences get longer. We confirm this on a 256K-context training job, running Mcore DCP across more than **11K** GPUs in our model production (§3). Per-rank token counts are comparable, yet microbatch execution times differ by 5× and CP-group loads by 2.4×. This imbalance leads to a 46.0% pipeline (PP) bubble and a data-parallel (DP) bubble that adds up to 12.8% to every iteration. Planning costs another 9%, because the scheduler rescans the entire pool of sequences on every iteration.

We present HyDra, a load-driven DCP system that keeps both planning and execution efficient at production scale. **(1) For execution**, HyDra derives one per-rank load target for the whole iteration from the quadratic attention computation, and sizes every sequence's CP degree to meet it, pulling every rank toward the same load and shrinking both the PP and DP bubble. The target is the lowest one the longest sequence can meet, which also makes it the easiest for short sequences to reach. Long sequences reach that target only at large CP degrees, which far exceed the upper bound of attention-head count in Ulysses. Ring does support such large CP degrees, but its per-rank traffic does not shrink as the degree grows, so communication becomes the bottleneck, preventing long sequences from reaching the target. HyDra therefore factorizes each CP degree into an inner Ulysses group inside a shallow outer ring, and overlaps the exposed all-to-alls with computation. A higher degree now lowers both computation and communication per rank, so even the longest sequences can reach the load target. **(2) For planning**, HyDra derives these quantities in closed form and places sequences with heaps. Iteration time is a graph of waits, each a maximum over ranks, so it has no closed form. Balance removes those maxima and leaves one unknown, the largest CP degree allowed, in a convex trade-off between the

pipeline bubble and the per-MB overhead. HyDra therefore solves that trade-off in closed form, and the load target and MB count follow from the degree by substitution. Placement is the only cost left, and a lazy heap per CP degree turns its whole-pool scans into logarithmic lookups.

We implement HyDra in approximately 3K LoC of Python on Megatron-LM [37]. On a **512**-GPU testbed, it improves end-to-end throughput over Mcore DCP by 1.09–1.25× (avg. 1.18×) at 32K context and 1.42–3.08× (avg. 2.48×) at 256K, keeping the compute stream busy over 76% of each iteration at both lengths. In production 256K-context training on **2,048** GPUs, it improves throughput by 1.10–1.43× (avg. 1.25×) over Mcore DCP and 1.33–1.90× (avg. 1.59×) over static CP. It shrinks the pipeline bubble from 36.4% to 13.6% of each iteration and cuts scheduling time by 2.3×, rising to 11.6× at 16K GPUs, while its training-loss curves track static CP over roughly 8,000 iterations. In simulation, it averages 1.28× over the state-of-the-art (SOTA) ByteScale [13] on a GQA model and 1.53× on a communication-heavy MLA model.

This paper makes the following contributions:

- We characterize production Mcore DCP on more than **11K** GPUs, where up to a 46.0% PP bubble, a 12.8% DP bubble, and a 9% scheduling stall dominate: balancing tokens instead of attention leaves MB times 5× and CP-group loads 2.4× apart.
- We propose HyDra, a load-driven DCP system that collapses the PP and DP bubbles with one load target. HyDra co-designs planning with execution, so the CP degrees that target demands stay affordable at production scale.
- We design a scheduler that derives the load target in closed form and places sequences with lazy heaps rather than whole-pool scans. We break the ring–Ulysses dilemma by running a Ulysses group inside a shallow ring, so a larger degree lowers computation and communication together.
- We implement HyDra in Megatron-LM and evaluate it at 32K and 256K context: on a **512**-GPU testbed it raises throughput over Mcore DCP by 1.09–1.25× (avg. 1.18×) at 32K and 1.42–3.08× (avg. 2.48×) at 256K, on **2,048** production GPUs by 1.10–1.43× (avg. 1.25×) over Mcore DCP and 1.33–1.90× (avg. 1.59×) over static CP while cutting scheduling time by 2.3×; in simulation to 40K GPUs it leads the SOTA baseline by 1.28–1.53×.

2 Background

2.1 5D Parallelism for Distributed LLM Training

Large-scale LLM training spans five parallelism dimensions, which planners combine under model, memory, and topology constraints [16, 18, 32, 37, 44]. Tensor parallelism (TP) shards operators within a node [36, 37]; pipeline parallelism (PP) splits layers into microbatched stages, trading bubbles against activation memory [14, 31]; expert parallelism (EP) shards mixture-of-experts (MoE) weights and routes tokens through all-to-all [11, 21]; and data parallelism (DP) replicates the model, shards the batch, and synchronizes gradients [26, 35]. Context parallelism (CP) shards each sequence across ranks, so each rank holds only a fraction of its activations and long contexts become feasible [10, 15, 22, 25, 28]. Because attention couples all tokens, CP circulates key and value blocks around a ring or switches sequence and head shardings through all-to-all [15, 28]. This work fixes TP, PP, and EP at their deployment settings and jointly manages CP and DP to place variable-length sequences across ranks.

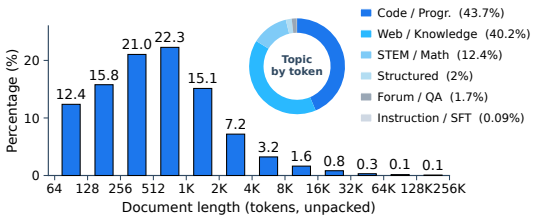


Figure 1. *Profile of the 256K-context training dataset.*

2.2 Long-Context Data Characteristics

Long-context corpora have highly skewed sequence-length distributions [13, 41]. In the benchmark dataset used for our measurements (Figure 1), lengths span four orders of magnitude up to 256K tokens, with more than 85% of sequences below 2K and fewer than 0.6% above 32K. The rare long ones are what teach long-context ability, so every batch mixes them with abundant short ones [29].

To fill each context window, systems pack short sequences together and mask attention across their boundaries [9, 20, 38], while ragged-tensor compilation and mini-sequence execution cut padding and non-attention memory further [12, 30]. Packing equalizes token counts, not attention work, which scales as $\sum_i O(s_i^2)$, so packed samples of the same size can differ widely in computation.

2.3 Dynamic Context Parallelism

DCP gives every sequence its own CP degree instead of one degree for the whole job [13, 17, 41]. Mcore ships a production scheduler that sizes each degree from a token budget, the most tokens a rank can hold, so every sequence goes to the fewest ranks that fit it [24]. Figure 2 shows the rule on four ranks with a 4K budget, where a 16K sequence

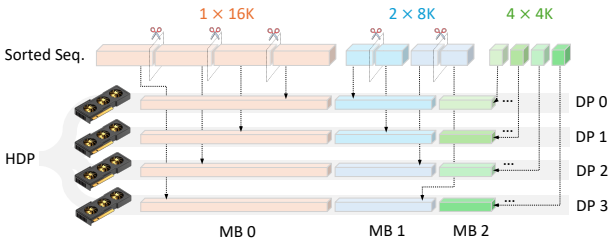


Figure 2. *DCP under a 4K per-rank token budget.*

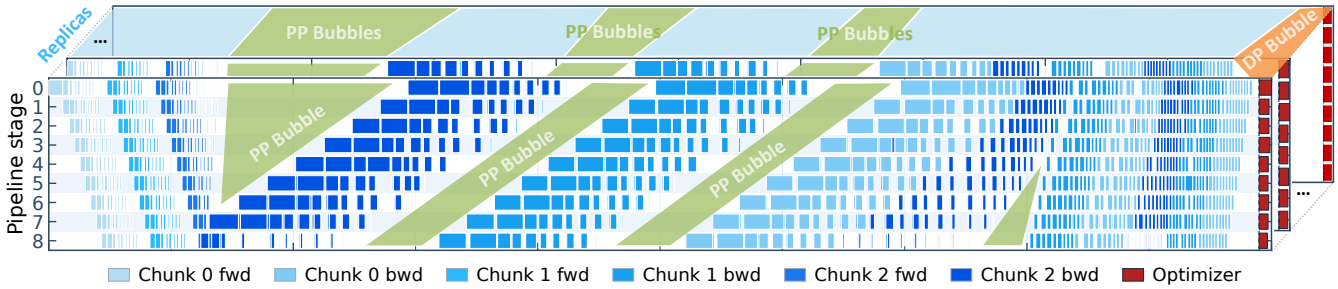


Figure 3. *Training timeline of one iteration under Mcore DCP, across nine PP stages and DP replicas.*

spans all four, an 8K sequence takes two, and a 4K sequence stays on one. Short sequences thus avoid the redundant communication a job-wide degree would force on them.

CP and DP partition the same batch in two directions, CP within one sequence and DP across sequences, so a static job organizes its ranks as a DP×CP mesh sized for the longest sequence. Per-sequence degrees leave no fixed mesh, so the DP×CP ranks become one pool, the *hybrid data-parallel* (HDP) domain, from which each group is cut at placement time [13, 24]. In Figure 2, the same four ranks serve as one group of four in MB 0, as two groups of two in MB 1, and as four groups of one in MB 2. In our production training, Mcore DCP improves throughput over static CP by about 8% on 32K data and 39% on 256K data. The budget, however, counts tokens, and Section 3 shows that this leaves quadratic attention work badly imbalanced.

3 Characterizing DCP at Production Scale

In this section, we measure Mcore DCP at production scale and identify its major bottlenecks.

3.1 Measurement Setup and Methodology

All measurements in this section are collected from a training job of Hy4-preview [40], an open-weight MoE model with 770B total and 49B activated parameters, running on more than **11K** GPUs at 256K context length. The model is parallelized with $tp = 2, pp = 9, ep = 32$, while the remaining ranks form the DP and CP dimensions managed by Mcore DCP [24], whose CP communication uses Ulysses-style all-to-all [15]. Traces are collected with Argus [46], a lightweight, always-on tracing system.

3.2 Inter-MB Imbalance

Figure 3 plots one iteration across all nine pipeline stages, with the forward and backward passes of every MB over three interleaved chunks and the closing optimizer step.

The heaviest MB bounds the iteration. Iteration time follows the *critical path* through this schedule: every stage must process every MB, so the longest-running one recurs stage after stage and dictates the iteration latency. Figure 3 traces one iteration of this job, where the heaviest MB, the first in the schedule, stretches every stage and idle time fans out around it. Under DCP each MB packs a different mix of variable-length sequences at a different CP degree, and in this example the heaviest takes more than 5× as long to execute as the lightest. We call this skew *inter-MB imbalance*; its footprint is the *PP bubble*, the green gaps in the figure, which consume 45.97% of this pipeline, nearly half of every iteration. A balanced schedule at pp=9 with three interleaved chunks would bubble only about 10% at this MB count, so skew inflates it 4.6×.

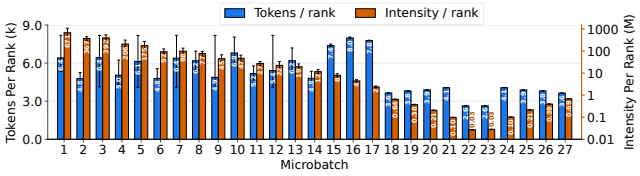


Figure 4. *Per-rank profile of each MB.*

The imbalance traces to a mismatch between how load is partitioned and how it scales. Figure 4 profiles each MB in this iteration with two per-rank quantities. The first is the token count, which grows linearly with length and is what the scheduler equalizes. The second is the *attention intensity* of a rank, $\mathcal{I} = \sum_i s_i^2/R$, the quadratic work of the sequences $\{s_i\}$ in the MB divided over the R ranks that carry them. The two diverge sharply. Token counts stay of the same order, averaging around 5K, while intensity spans four orders of magnitude, from ≈0.1 M to ≈1,000 M. Equal tokens need not imply equal computation, which has motivated FLOP-aware packing and scheduling [13, 42]; our point is that the skew survives *production* DCP, whose scheduler equalizes tokens and memory per rank but never the quadratic attention cost.

> ***Insight 1:*** *Under Mcore DCP, per-rank tokens are balanced yet attention computation intensity varies by orders of magnitude, opening a 5× MB execution-time gap and widening the PP bubble from its 10% floor to 45.97%. MBs must therefore be balanced by attention computation, not tokens alone.*

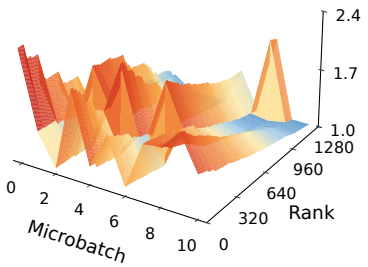
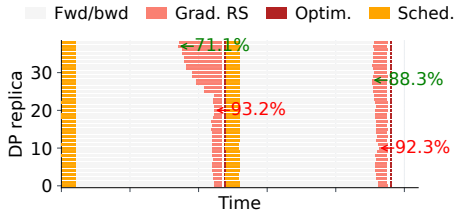
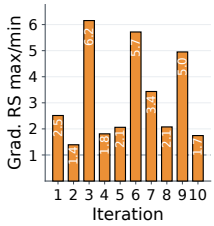


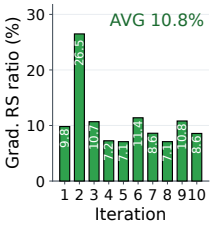
Figure 5. *Inter-CP group imbalance.*



(a) *Per-iteration DP timeline*



(b) *Grad. RS max/min*



(c) *Grad. RS ratio*

Figure 6. *DP bubble.*

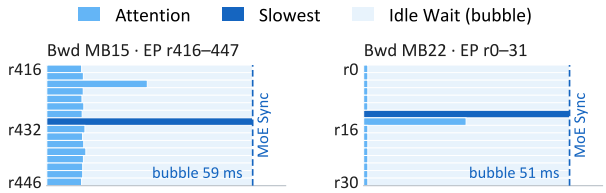


Figure 7. *Intra-EP group imbalance.*

3.3 Intra-MB Imbalance

Imbalance also arises *within* a single MB. It starts as a computation load gap between the CP groups that share the MB, and every MoE all-to-all stamps that gap onto whole EP groups. Nothing reconciles those groups until the iteration ends, so the gap may accumulate MB after MB into the *DP bubble*.

Inter-CP-group imbalance. An HDP domain consists of several CP groups, each responsible for a subset of an MB. For each CP group we calculate the computation of the sequences it holds, then normalize these per-group values within each MB to its least-loaded group (set to 1); Figure 5 plots the result for a representative subset of an iteration's MBs. The spread is large: within the same MB, the busiest CP group carries up to 2.4× the computation of the idlest. It is worst for MB 0, which is also the MB that paces the iteration, so both forms of imbalance land on the same MB.

Intra-EP-group imbalance. Every MoE layer opens with a dispatch all-to-all, where the CP groups sharing an EP group must rendezvous before any of them can proceed. Such a barrier cannot absorb the skew; it spreads it, stamping the slowest group's cost onto every group in the EP domain. In Figure 7, one rank's attention runs 70 ms while the rest of its 32-rank EP group finishes in ~12 ms and idles ~59 ms at the barrier. The stall bites hardest where the per-sequence CP degree is small: in the tail MBs an EP group spans many CP groups of widely varying sequence lengths, and the bubble ratio reaches 7.4–20.0%, versus 4–7% in the middle MBs and < 4% in the leading ones. Summed over one iteration, these stalls waste ~2.3% of aggregate GPU time.

DP bubble. Each EP group is therefore limited by its slowest CP group. Unless the CP degree exceeds the EP degree, different DP replicas do not synchronize again until the gradient reduction at the end of the iteration. The imbalance does not average out across MBs: a replica assigned a heavier share of sequences remains slower, so the gap between replicas grows throughout the iteration. As the first cross-replica barrier, gradient reduction exposes this accumulated gap. Figure 6a shows two representative iterations, one severely imbalanced and one mild: the forward/backward boundary is staggered across replicas, so early finishers idle through the gradient reduce-scatter (RS) window until the slowest arrives. The straggler is large and persistent: the slowest replica's gradient RS runs up to 6.2× as long as the fastest (Figure 6b), and the RS phase occupies 7% to 27% of the aggregate DP GPU-time (Figure 6c), stretching each iteration by 8–12.8%. Prior work has documented DP imbalance under static CP [13]. In DCP, we further show that the DP bubble arises from intra-MB imbalance accumulating across MBs.

> ***Insight 2:*** *Balancing MBs is not enough: within one MB, CP groups differ by up to 2.4× in attention computation, and no barrier in the iteration corrects it, so the gap accumulates MB after MB into an 8–12.8% DP bubble. Balancing must reach inside the MB, across CP groups and DP replicas.*

3.4 Scheduling Overhead

Beyond load imbalance, Mcore DCP also incurs overhead to compute each schedule. It runs on the CPU at the head of each iteration's data-fetch path and lies on the critical path: every steady-state iteration opens with a scheduling band spanning all DP replicas (Figure 6a), during which the GPUs idle. Across profiled iterations this stall is stable at 8–9% of the iteration time.

The cost is not the scheduling decision: choosing a sequence's CP degree and ranks is cheap. The plan is built incrementally, one sub-sample and one MB at a time. For each sub-sample, the scheduler linearly scans the entire HDP pool to choose ranks, rebalance load, and fill idle ranks. This

cost grows with both the number of sub-samples and the pool width. Long-context training makes both large: a batch holds tens of thousands of short sub-samples, while the pool spans tens of thousands of ranks. What is negligible at small scale thus compounds into the stall we observe.

> ***Insight 3:*** *Mcore DCP scans the entire rank pool once per sub-sample, tens of thousands of times per batch, idling GPUs for 8–9% of each iteration. Online planning must preserve placement quality without sweeping the entire pool for every sequence.*

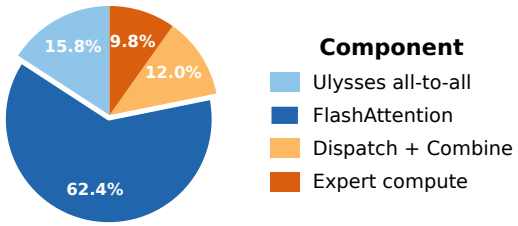


Figure 8. *Forward-pass latency breakdown of one transformer layer in the heaviest-loaded MB.*

3.5 CP Scaling Dilemma

The MB that paces the iteration is attention-bound. Figure 8 breaks down one transformer layer on that MB's bottleneck rank. The rank processes a single 256K-token sequence, to which Mcore's memory-driven policy assigns cp=32. FlashAttention dominates the layer, accounting for 62.4% of its latency and far exceeding either the Ulysses all-to-all or the entire MoE block. The iteration is therefore paced by the longest sequence's per-rank attention cost, S^2/cp; reducing iteration time requires lowering this term.

Balance asks for a larger degree, not a smaller one. Every MB uses the same fixed HDP rank pool. Reducing the heaviest MB's load therefore requires assigning more ranks to its long sequences, which lowers their S^2/cp cost. Mcore packs each sequence onto the fewest ranks that hold it, while ByteScale uses the minimum required number of devices [13, 24]. Solver-based systems choose group degrees from a finite set of candidate configurations and assign sequences among them [23, 41]. They can select larger degrees, but still rely on native Ulysses or ring communication and thus inherit the limits analyzed next. Effective balancing therefore requires both going beyond the memory-minimum degree and executing those larger degrees efficiently; the two native CP schemes fail in opposite ways.

Ulysses scales down both computation and communication but stops at the head count. Ulysses uses all-to-all transposes to partition attention heads across ranks. As the degree P grows, both per-rank attention computation

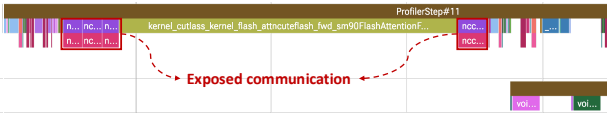


Figure 9. *Ulysses all-to-all (QKV and output transpose) is exposed on the attention critical path.*

and communication fall as $1/P$:

$$T^{\text{uly}}_{\text{comp}} \propto \frac{S^2 h}{P}, \qquad V^{\text{uly}}_{\text{comm}} \approx \frac{4Sh}{P}. \tag{1}$$

This joint reduction is what load balancing requires; otherwise, communication would replace computation as the bottleneck. First, head partitioning caps $P \leq H/\text{tp}$. Flagship models typically have at most about 64 attention heads [7, 19, 39, 40, 45], with even fewer full-attention heads in hybrid models [34]; at tp=2, Ulysses therefore stops at cp=32, short of what the bottleneck MB needs to reach the load target. Second, the QKV and output all-to-alls remain exposed on the attention critical path, accounting for 4% of end-to-end iteration time in production (Figure 9).

Ring reaches any degree, but its traffic never shrinks. Ring CP circulates K/V blocks around the CP ring using point-to-point send/recv. These blocks contain only the $h_{kv} \leq h$ key/value heads. Unlike Ulysses, ring has no head-count limit and pipelines communication with attention computation. Its per-rank computation decreases with P, but its communication volume does not:

$$T^{\text{ring}}_{\text{comp}} \propto \frac{S^2 h}{P}, \qquad V^{\text{ring}}_{\text{comm}} \approx 2Sh_{kv}, \qquad \frac{V^{\text{ring}}_{\text{comm}}}{V^{\text{uly}}_{\text{comm}}} = \frac{P\,h_{kv}}{2h}. \tag{2}$$

Ring traffic is independent of P. It undercuts Ulysses only below $P = 2h/h_{kv}$: cp=16 for h=64 and h_{kv}=8, but cp=2 under MLA, where $h_{kv} \approx h$. Both crossovers are far below the degrees needed to balance a 256K sequence. At those degrees, ring communication sets iteration time; under MLA, it already exceeds attention computation in production and cannot be hidden.

> ***Insight 4:*** *Balancing the heaviest MB requires increasing cp. Ulysses shrinks traffic with cp but stops at the head count and leaves its all-to-all exposed. Ring reaches any cp, but its traffic never shrinks. A CP scheme must scale beyond the head count without making communication the bottleneck.*

4 HyDra Overview

4.1 Design Principles

The four insights of §3 translate into four principles that guide the design of HyDra.

(1) Balance computation, not just tokens. Insight 1 shows that equal token allocations conceal orders-of-magnitude differences in attention workload, leaving MB

execution times 5× apart. Production DCP must therefore balance attention computation across MBs, not just their tokens, while preserving the memory feasibility it already provides.

(2) Balance within and across MBs. Insight 2 shows that per-sequence CP leaves CP groups within an MB with widely different computation loads; this skew stalls the MoE all-to-alls and accumulates across MBs into the DP bubble. Balancing must therefore cover CP groups within each MB and DP replicas across the iteration.

(3) Keep online planning lightweight. Insight 3 shows that building the plan stalls 8–9% of every production iteration, at a cost that grows with both the sequence count and the width of the rank pool. Planning must therefore avoid whole-pool work as cluster size enlarges both.

(4) Scale CP without creating a communication bottleneck. Insight 4 shows that balancing the heaviest MB drives the CP degree up rather than down, yet Ulysses and ring communication fail those degrees from opposite directions. The communication mechanism must therefore reach degrees beyond the attention-head count with traffic that shrinks as the degree grows and can be overlapped.

4.2 System Overview

As shown in Figure 10, HyDra realizes the four principles with two co-designed components, a load-driven scheduler and a balance-preserving CP communication engine, integrated into an existing training stack.

(1) Load-driven scheduling. At each iteration, the scheduler estimates attention workload from sequence lengths and chooses each sequence's CP degree and placement to balance MBs, their CP groups, and DP replicas against a single per-rank load target, while keeping planning inexpensive. This component realizes Principles 1–3.

(2) Balance-preserving CP communication. Reaching that target means raising a heavy sequence's CP degree, which helps only if communication does not become the new critical path. This component executes the selected degrees beyond native CP limits and hides the exposed transfers behind dependency-free computation, realizing Principle 4.

The end-to-end flow is simple: the data path collects sequence lengths, the scheduler produces a placement plan, the data are routed accordingly, and attention executes with the selected CP groups. Neither component stands alone: the target the scheduler sets is unreachable on native CP, and the degrees the engine executes pay off only when a balanced plan requests them. Section 5 presents both designs in detail.

5 HyDra Design

5.1 Load-Driven DCP Scheduling

HyDra sizes each sequence's CP degree by its quadratic attention load, not its memory footprint: splitting a length-s sequence over cp ranks yields per-rank load s^2/cp, and the

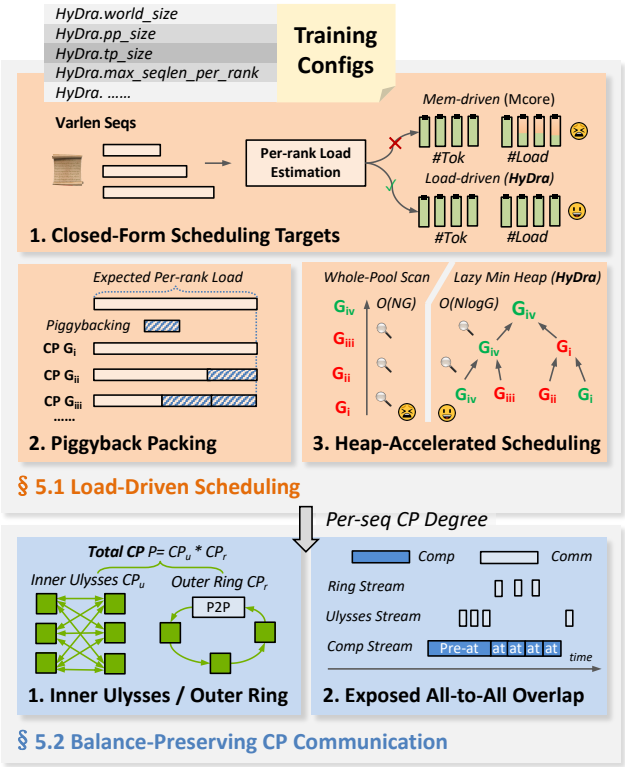


Figure 10. *HyDra overview.*

heaviest MB paces the iteration (§3.2). It then derives closed-form targets, realizes them through piggyback packing, and accelerates planning with heaps.

5.1.1 Closed-Form Scheduling Targets.

The targets couple per-rank load Λ (and thus MB count M) with a CP-degree cap C; we derive $\Lambda^\star$ and $M^\star$ for fixed C, then $C^\star$ and each sequence's $cp(s)$ (Table 1).

Balance-oriented load target. Fix the cap C for now. The target must reconcile two asymmetric limits imposed by long and short sequences. First, the longest sequence sets a hard floor on the target: even at the maximum degree C, the MB holding it carries a per-rank load of at least $s_{\max}^2/C$. Second, short-sequence MBs can exhaust the per-rank token budget B before reaching a large quadratic-load target. For a sequence i assigned to p_i ranks, let $x_i = s_i/p_i$ be its rank-local token count. For the sequences $\mathcal{P}_r$ packed on rank r, token occupancy t_r and quadratic attention load q_r satisfy

$$\begin{aligned} t_r &= \sum_{i \in \mathcal{P}_r} x_i \le B, \\ q_r &= \sum_{i \in \mathcal{P}_r} s_i x_i = t_r \bar{s}_r \le B \bar{s}_r. \end{aligned} \tag{3}$$

Here $\bar{s}_r = q_r/t_r$ is the token-weighted sequence length: it determines the quadratic work obtained from each packed token, and $t_r \le B$ caps the attainable load at $q_r \le B\bar{s}_r$. A rank filled with short sequences has a small $\bar{s}_r$, so it exhausts B

Table 1. *Notation for HyDra scheduling.*

Symbol	Description
Workload	
$s,\ s_i,\ s_{\max}$	sequence length; of sequence i; longest in the batch
N	# sequences in the batch
W	total quadratic attention work, $\sum_i s_i^2$
System configuration	
G	# ranks in the HDP pool (DP×CP)
B	per-rank token budget (memory limit)
pp	pipeline-parallel depth (# stages)
$h,\ h_{kv}$	# query / KV heads per rank after TP ($h_{kv} \ll h$)
Scheduler decisions	
$\Lambda,\ \Lambda^\star$	per-rank load target; balance-oriented target
$M,\ M^\star$	MB count; target count
C_{mem}	smallest cap fitting the longest sequence within B
$C,\ \widehat{C}^\star,\ C^\star$	CP-degree cap; optimum; power-of-two value
$cp(s)$	CP degree of a length-s sequence
Cap cost model	
$\theta,\ c$	time per unit per-rank load; fixed cost per MB
$\beta,\ \gamma$	bubble-arm / overhead-arm coefficient
κ	MBs per unit of cap, $W/(s_{\max}^2 G)$
CP execution	
P	CP degree of a sequence (= $cp(s)$)
$cp_u,\ cp_r$	inner Ulysses / outer ring factor of P

while q_r remains far below a large Λ. Increasing Λ beyond this attainable load cannot put more computation into such MBs; instead, it leaves them underloaded while allowing long sequences to use fewer CP ranks and approach the higher target, worsening inter-MB skew.

HyDra therefore sets the load target to this floor,

$$\Lambda^\star = \frac{s_{\max}^2}{C}. \tag{4}$$

A lower target is infeasible for the capped longest sequence, while a higher one is harder for token-limited short-sequence MBs to attain, so $\Lambda^\star$ is at once the smallest feasible target and the easiest to approach. It is selected primarily for MB balance, rather than as the unconstrained minimizer of an idealized latency function.

Target MB count. Let $\bar{q}$ be the realized average per-rank load. Work conservation gives $W = MG\bar{q}$. Under balanced packing, $\bar{q} \approx \Lambda^\star$, yielding the target

$$M^\star = \left\lceil \frac{W}{G\Lambda^\star} \right\rceil = \left\lceil \frac{CW}{Gs_{\max}^2} \right\rceil. \tag{5}$$

$M^\star$ is a planning target rather than the realized count. Token limits, power-of-two CP groups, and group alignment may push packing off it; PP-bubble and virtual-pipeline constraints then enforce feasibility.

Closed-form CP cap. The cap is the only value left to choose, and choosing it is a trade-off. Since $\Lambda^\star = s_{\max}^2/C$ and $M^\star \approx \kappa C$ with $\kappa = W/(s_{\max}^2 G)$, a larger cap makes every MB smaller and the pipeline bubble shorter, but it also creates more MBs, and each MB carries a fixed cost c,

$$\begin{aligned} T(C) &\approx \frac{\theta W}{G} + (pp-1)\theta\Lambda^\star + cM^\star \\ &\approx \underbrace{T_0}_{\text{compute}} + \underbrace{\beta/C}_{\text{bubble}} + \underbrace{\gamma C}_{\text{overhead}}. \end{aligned} \tag{6}$$

Without balance, iteration time is a graph of waits, each a maximum over ranks, and no closed form exists. Under a perfectly balanced schedule the equation is exact, since every MB takes the same time $\theta\Lambda^\star$, 1F1B takes $M+pp-1$ MB times, and work conservation turns the first M of them into $\theta W/G$. The targets above drive the schedule toward that balance, so we use it as an approximation. The coefficients in Eq. (6) are

$$T_0 = \theta W/G, \quad \beta = (pp-1)\,\theta\, s_{\max}^2, \quad \gamma = c\,\kappa. \tag{7}$$

T_0 is the total work divided over the ranks, which the cap does not change, so only the last two terms decide the best cap. The β/C term decreases as the cap grows while γC rises, so the curve is convex and has one lowest point,

$$\begin{aligned} T'(C) = 0 \quad &\Longleftrightarrow \quad C^2 = \frac{\beta}{\gamma}, \qquad C > 0, \\ \widehat{C}^\star &= \sqrt{\frac{\beta}{\gamma}} = s_{\max}^2 \sqrt{\frac{(pp-1)\theta G}{cW}} \end{aligned} \tag{8}$$

Only θ/c has to be measured, and a ratio of two costs is easier to measure reliably than either cost by itself. Memory and power-of-two groups then decide the cap we can run. Let $\lceil x \rceil_2$ denote the smallest value in $\{1, 2, 4, \ldots\}$ that is at least x; then

$$\begin{aligned} C_{\text{mem}} &= \left\lceil \frac{s_{\max}}{B} \right\rceil_2, \\ C^\star &= \min\left\{G, \left\lceil \max\{C_{\text{mem}}, \widehat{C}^\star\} \right\rceil_2 \right\}. \end{aligned} \tag{9}$$

We round up rather than down, because a larger cap only lowers the target, and MBs made of short sequences can always reach a lower target. Rounding to powers of two also means that a small error in θ/c usually gives the same cap.

Load-driven CP degree. A length-s sequence at degree cp also places s/cp tokens per rank, so meeting the computation target requires $cp \geq \lceil s^2/\Lambda^\star \rceil$, while fitting within the token budget requires $cp \geq \lceil s/B \rceil$. Megatron-Core enforces only the latter and thus leaves the computation skew measured in Section 3. HyDra uses the smallest power-of-two degree satisfying both:

$$\text{cp}(s) = \min\left\{C^\star, \left\lceil \max\left\{\frac{s^2}{\Lambda^\star}, \frac{s}{B}\right\} \right\rceil_2 \right\}. \tag{10}$$

Because $\Lambda^\star = s_{\max}^2/C^\star$ and $C^\star \geq C_{\text{mem}}$, both per-sequence requirements fit within the cap, so the outer minimum preserves them. An open group g is feasible for sequence s if adding it keeps both $\text{load}(g) + s^2/|g| \leq \Lambda^\star$ and $\text{tokens}(g) + s/|g| \leq B$.

Algorithm 1: HyDra DCP Scheduling

Input : documents $\{s_i\}$ and $s_{\max}$; rank pool G; cap $C = C^\star$ (Eq. (9)); budget B
Output : $\mathcal{G}[m][r]$: doc. lists for MB m and rank r

▹ *Phase 1: closed-form load targets*
$W \leftarrow \sum_i s_i^2$; $\Lambda^\star \leftarrow s_{\max}^2 / C$ *target per-rank load*
$M^\star \leftarrow \lceil C\,W / (s_{\max}^2 G) \rceil$ *Eq. (5)*

▹ *Phase 2: longest-first piggyback packing*
sort $\{s_i\}$ descending; $m \leftarrow 0$
while *documents remain* **do**
  open MB m over all G ranks
  while *documents remain* **do**
    $s \leftarrow$ next document; $k \leftarrow \mathsf{cp}(s)$ *Eq. (10)*
    if *free ranks* $\geq k$ **then**
      $g \leftarrow$ new k-aligned group on free ranks
    else if $\exists$ *feasible group* g *with* $|g| \geq k$ **then**
      $g \leftarrow$ smallest such group
      break ties by load *piggyback*
    else break
    pack s into g; update load and occupancy
    if k *has shrunk* **and** *MB balanced* **then break**
  backfill idle ranks; $m \leftarrow m + 1$

▹ *Phase 3: finalize*
broadcast m; **return** $\mathcal{G}$

5.1.2 Piggyback Packing. Algorithm 1 places sequences longest-first, one MB at a time across all G ranks. For a sequence needing degree k, it first opens a new k-aligned group if enough free ranks are available. Otherwise, it *piggybacks* on the smallest feasible open group of size at least k, breaking ties by load. Alignment keeps every group within a prebuilt power-of-two rank chunk and preserves ring ordering. This packs short sequences into the gaps left by wide long-sequence groups, cutting both idle ranks and the MB count. After closing each MB, HyDra backfills idle ranks; once all sequences are placed, it broadcasts the realized MB count so every pipeline stage uses the same count. Because every MB is packed across the whole HDP pool against one target $\Lambda^\star$, the balance extends past the pipeline to the CP and EP groups inside an MB and the DP replicas that synchronize after it (Insight 2).

5.1.3 Heap-Accelerated Scheduling. A direct implementation scans every compatible group per sequence and resweeps the rank pool on each backfill, recreating the whole-pool traversals of Insight 3. HyDra partitions open groups by CP degree and keeps, per class, a lazy min-heap keyed by group load and a companion max-heap keyed by remaining token capacity. Stale load-heap entries are discarded on access, and a candidate that violates the token budget is *demoted* to the capacity heap rather than rescanned. Readmission is cheap because a class fixes the divisor $|g|$ and sequences arrive longest-first, so the per-rank demand a class sees never grows: each group is touched $O(1)$ times per placement instead of once per search. If the load-heap minimum already reaches $\Lambda^\star$, the class is pruned outright, since loads only grow and no later sequence can enter it. A third min-heap backfills idle ranks by repeatedly doubling the smallest group, and the pool maximum read by the MB-balance test is maintained incrementally, as every rank in a CP group carries the same load. These structures reproduce the linear search's least-loaded placement and deterministic tie-breaking exactly, while replacing its $O(NG)$ group and rank sweeps with $O(N \log G)$. HyDra lowers this cost rather than hiding it behind training, since planning and kernel launch share one CPU.

5.2 Balance-Preserving CP Communication

Balancing raises a heavy sequence's CP degree P, reducing its per-rank load to s^2/P. Native CP cannot exploit this: Ulysses is head-count limited and exposed, while ring has a degree-independent traffic floor (§3.5). HyDra combines Ulysses–ring factorization with exposed all-to-all overlap.

5.2.1 Inner Ulysses × Outer Ring. The factorization runs each selected degree on an inner Ulysses dimension inside a shallow outer ring [10].

Beyond the head-count limit. We factor the scheduler-selected degree into an inner Ulysses group over heads and an outer ring over the sequence:

$$P = cp_u \cdot cp_r, \qquad cp_u = \min(P, h), \qquad cp_r = P / cp_u. \tag{11}$$

For $P \leq h$ this is pure Ulysses; beyond h the head dimension saturates and the outer ring supplies the residual factor, so the scheduler may request any P up to $C^\star$ while per-rank computation stays s^2/P.

Keeping the outer ring shallow. Per-rank traffic splits across the two dimensions:

$$V(P) \approx \underbrace{\Theta\big(s\,h/P\big)}_{\text{inner all-to-all}} + \underbrace{\Theta\Big(s\,h_{kv}\,\frac{cp_r - 1}{cp_r}\Big)}_{\text{outer ring}}. \tag{12}$$

The Ulysses term falls as $1/P$ while the ring term flattens as cp_r grows; the decomposition does not remove that floor but confines it to the residual factor $cp_r = P/h$ left after the Ulysses dimension saturates. At $P = 256$ and $h = 64$ the outer ring spans only four ranks, and its payload is KV-only:

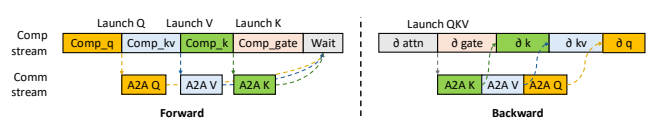
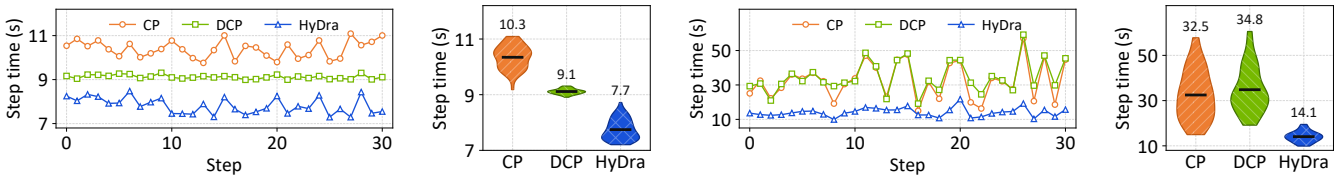
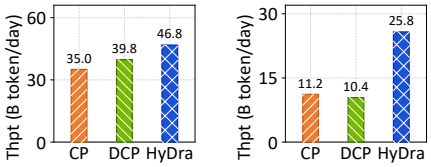


Figure 11. *Asynchronous all-to-all schedule in gated MLA, forward and backward.*



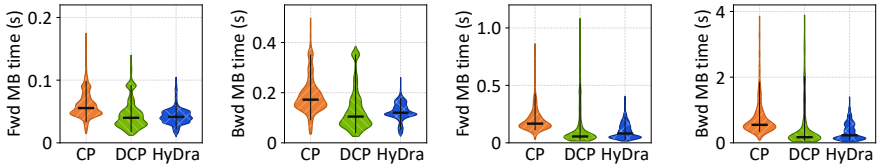
(a) *32K, per step* **(b)** *32K, range* **(c)** *256K, per step* **(d)** *256K, range*

Figure 12. *Step time on 512 H20 GPUs.*



(a) *32K* **(b)** *256K*

Figure 13. *Training throughput.*



(a) *Fwd, 32K* **(b)** *Bwd, 32K* **(c)** *Fwd, 256K* **(d)** *Bwd, 256K*

Figure 14. *Per-microbatch forward and backward time on 512 H20 GPUs.*

small under GQA, though under MLA the up-projected K/V restores a full per-head payload ($h_{kv} \approx h$).

5.2.2 Exposed All-to-All Overlap. The inner all-to-all still lies on the attention critical path, so instead of materializing the query, key, and value and then issuing three blocking collectives, HyDra dispatches each all-to-all on a separate communication stream as soon as its tensor reaches final form (Figure 11). The mutually independent streams of the gated MLA [33] projection fix the launch order: the query goes first, right after its RoPE, covered by the KV down- and up-projections; the value is final once the up-projection output is split, covered by the RoPE and concatenation the key still needs; the key goes last, covered by the gate projection, which needs no context-parallel communication at all. Deferring all three waits until after the gate maximizes every overlap window. Backward needs no extra scheduling: launch and wait are paired autograd functions that share one communication handle and swap roles under differentiation, so the adjoint of the compute hiding a collective lands exactly between the two nodes and hides the corresponding gradient collective. The shallow outer ring pipelines with attention as in standard ring CP. Together, the factorization and the overlap let a larger P deliver its s^2/P latency reduction instead of a new communication tail.

6 Evaluation

6.1 Experimental Setup

Clusters. We evaluate HyDra in three settings of increasing scale: a **512**-GPU NVIDIA H20 testbed (§6.2), a production job on **2,048** NVIDIA H800 GPUs (§6.3), and simulation at 8K–40K GPUs (§6.4). Both clusters place 8 GPUs per NVLink node and connect nodes over RoCE.

Model. Both hardware settings train the same internal MLA-based model[1].

Dataset. We use two long-context training datasets, one at a 32K context length and one at 256K, whose sequence lengths span orders of magnitude.

Parallelism. Both settings use $tp1, pp4, ep8$, with a 4K-token per-rank budget at 32K context and 8K at 256K.

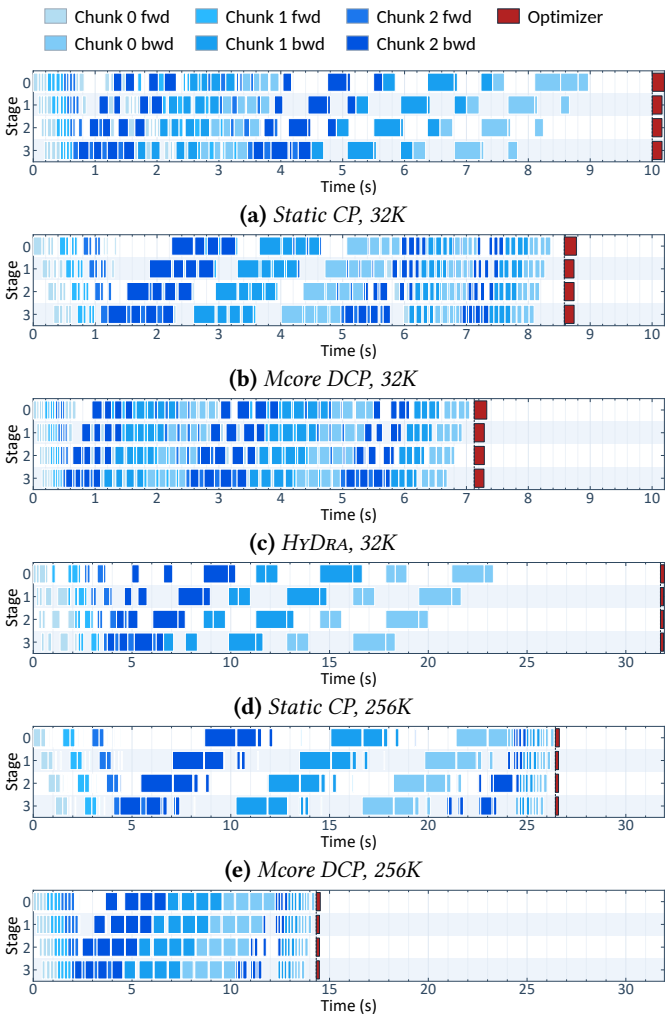
Baselines. On hardware we compare against two production baselines: static CP at the smallest degree that fits the longest sequence, cp=8 at 32K and cp=32 at 256K, and Mcore DCP [24], which sizes CP per sequence. In simulation we additionally compare against a broader set of baselines [2], including ByteScale [13], WLB-LLM [42], and FlexSP [41].

6.2 Testbed Performance

Throughput. On the 512-GPU testbed, HyDra is the fastest system at both context lengths, and its margin widens as the context grows. At 32K, it raises end-to-end training throughput by 1.19–1.52× (averaging 1.34×) over static CP and by 1.09–1.25× (averaging 1.18×) over Mcore DCP, sustaining 46.8 B tokens per day against 35.0 B and 39.8 B, and cutting the mean step time to 7.7 s from 10.3 s and 9.1 s (Figures 12b and 13a). At 256K, the margins grow to 1.19–3.00× (averaging 2.30×) over static CP and 1.42–3.08× (averaging 2.48×) over Mcore DCP: 25.8 B tokens per day against 11.2 B and 10.4 B, at a mean step time of 14.1 s against 32.5 s and 34.8 s (Figures 12d and 13b). Mcore DCP is the stronger baseline at 32K but falls behind static CP at 256K, where, as §3 shows, its memory-driven degrees leave the heavy MBs unbalanced. The per-step series make the gap visible: HyDra holds a nearly flat step time, while both baselines fluctuate from step to step, most severely at 256K (Figures 12a and 12c). The

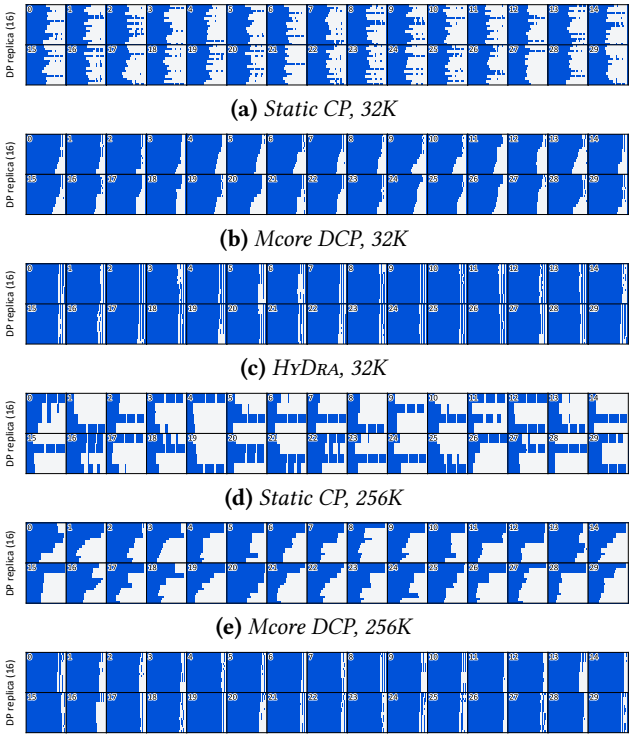
[1]Model details are withheld for confidentiality.

[2]These baselines are either closed-source or never validated at production scale, so we can only compare against them in simulation.



(a) *Static CP, 32K*
(b) *Mcore DCP, 32K*
(c) *HyDra, 32K*
(d) *Static CP, 256K*
(e) *Mcore DCP, 256K*
(f) *HyDra, 256K*

Figure 15. *One iteration timeline across the four PP stages on 512 H20 GPUs.*



(a) *Static CP, 32K*
(b) *Mcore DCP, 32K*
(c) *HyDra, 32K*
(d) *Static CP, 256K*
(e) *Mcore DCP, 256K*
(f) *HyDra, 256K*

Figure 16. *Gradient reduce-scatter (gray) of the 16 DP replicas, one panel per step, on 512 H20 GPUs.*

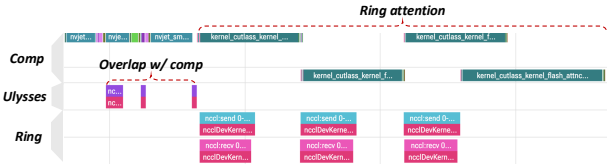


Figure 17. *Attention trace in HyDra.*

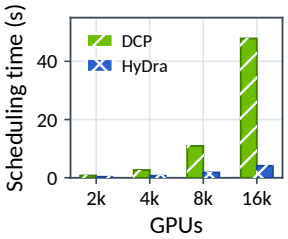


Figure 18. *Scheduling time.*

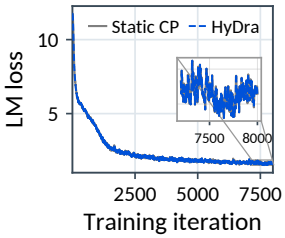
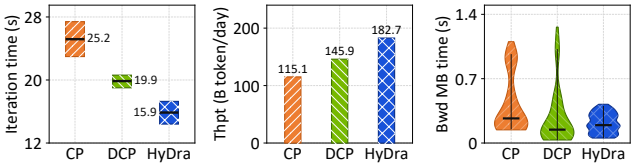


Figure 19. *Training loss.*

same imbalance appears one level down, on individual MBs: both baselines spread MB times widely and trail long tails of slow MBs, whereas HyDra compresses the distribution at both context lengths and in both directions (Figure 14).

PP bubble. Tracing where each iteration goes explains the gap. At 32K, HyDra leaves the smallest pipeline bubble, averaging 17.5% of the iteration and staying within 13.9–20.9%, against 24.7% (13.6–35.5%) under static CP and 26.9% (21.7–31.6%) under Mcore DCP. At 256K the baselines diverge sharply: static CP averages 36.5% (10.1–64.7%) and Mcore DCP 55.7% (42.0–65.5%), since a memory-driven degree keeps the heaviest MB heavy and every stage waits on it once per microbatch. HyDra averages 23.3% (18.0–34.4%), the only system whose pipeline idle stays both low and narrow as the context grows. Figure 15 shows this on the pipeline: under Mcore DCP the four stages are pocked with white gaps that recur through the step and widen at 256K (Figures 15b and 15e), whereas under HyDra the stages stay packed and the iteration ends far earlier (Figures 15c and 15f).

DP bubble. Static CP suffers the largest DP bubble, averaging 15.3% (0.5–33.3%) of the iteration at 32K and 24.5% (0.1–67.4%) at 256K, because a fixed degree never rebalances



(a) *Iteration time* **(b)** *Throughput* **(c)** *MB-time range*

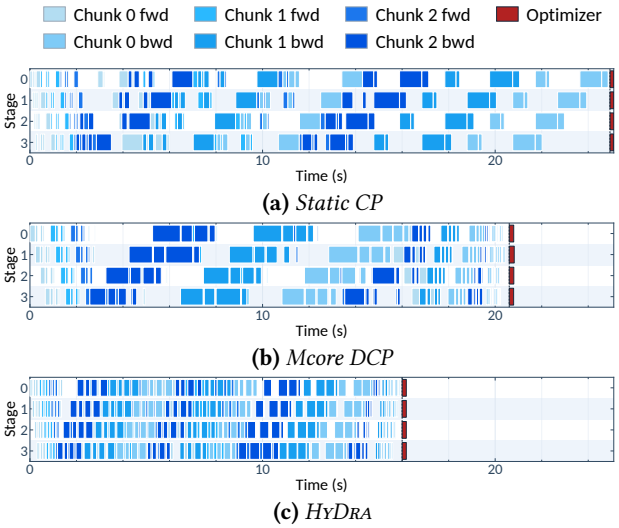
Figure 20. *End-to-end performance on 2,048 GPUs.*

computation across replicas. In Figures 15a and 15d the bubble is the wide empty span between the last chunk's backward kernels and the optimizer, where rank 0 has finished its share and waits for the slowest replica; under HyDra the optimizer follows the last backward almost immediately. Figure 16 shows the same effect across all 16 replicas: the gray bands are the reduce-scatter itself, and since a replica stays in the collective until the slowest one arrives, wider bands mean more DP idle. The collective takes 34.8% of the iteration on average under static CP at 32K and 39.3% at 256K, against 4.2% under Mcore DCP at both lengths and 5.3% and 4.3% under HyDra, whose bands stay thin and aligned. Sizing CP per sequence largely removes this, but not reliably: Mcore DCP averages 0.9% (0.6–2.4%) at 32K and 2.1% (0.1–20.9%) at 256K, where its worst steps still idle over a fifth of the iteration. HyDra is the only system that stays low at both context lengths, averaging 1.0% (0.7–2.9%) at 32K and 0.8% (0.2–2.7%) at 256K. The two effects compound: the compute stream is busy 81.5% of the iteration under HyDra at 32K, against 60.0% under static CP and 72.2% under Mcore DCP, and 76.0% at 256K, against 39.0% and 42.2%.

CP communication. The degrees the scheduler picks pay off only if the engine can execute them, so we profile one attention layer of an MB under HyDra (Figure 17). Both halves of Section 5.2 behave as intended: the query, key, and value all-to-alls, fully exposed under Mcore's Ulysses (Figure 9), now run behind dependency-free computation, and the outer ring's send/recv pipelines with the attention kernels instead of blocking them. HyDra thus runs any scheduler-selected degree, including those beyond the head count.

Scheduling overhead. Because scheduling runs entirely on the CPU, we can measure it at any rank count without the GPUs; Figure 18 compares the two schedulers from 2K to 16K GPUs. Mcore DCP's whole-pool scans reach 47.85 s at 16K GPUs, while HyDra stays at 4.12 s, an 11.6× speedup.

Model convergence. HyDra assigns sequences to different ranks and CP-group sizes than static CP, so its gradient and loss reductions accumulate in a different order; because floating-point addition is not associative, a small divergence is expected rather than a defect. Figure 19 bounds it on the aggregate language-modeling loss: over roughly 8,000 iterations, by which point the curve has flattened and the model is close to converged, the two curves stay within 10^{-3}. The inset over the final iterations shows them interleaved, and

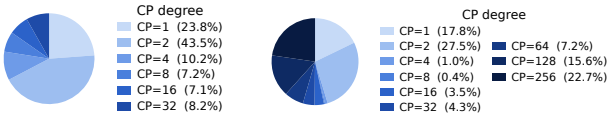


(a) *Static CP*

(b) *Mcore DCP*

(c) *HyDra*

Figure 21. *One iteration timeline across the four PP stages.*



(a) *Mcore DCP* **(b)** *HyDra*

Figure 22. *Fraction of GPUs at each CP degree.*

HyDra records the lower loss in around 50% of iterations, so the gap is unbiased noise rather than systematic drift.

6.3 Production Speedup

Throughput. On the 2,048-GPU production run, HyDra consistently outperforms both baselines, raising end-to-end training throughput by 1.33–1.90× (averaging 1.59×) over static CP and by 1.10–1.43× (averaging 1.25×) over Mcore DCP. On average, it sustains 182.7 B tokens per day and cuts the iteration time to 15.9 s, from 25.2 s under static CP and 19.9 s under Mcore DCP (Figures 20a and 20b). MB times are also far more even: their coefficient of variation is 49% under HyDra, against 73% under static CP and 108% under Mcore DCP, and the maximum MB time falls as well (Figure 20c), which directly shrinks the PP bubble examined next.

PP bubble. The pipeline bubble wastes 47.1% of each iteration under static CP and 36.4% under Mcore DCP, but only 13.6% under HyDra (Figure 21). The mechanism is the one measured on the testbed: HyDra spends context parallelism where it balances MBs, placing a far larger fraction of GPUs at high CP degrees than Mcore DCP (Figure 22).

DP bubble. The DP bubble idles 3.2–59.2% of the critical path under static CP and 0.4–10.1% under Mcore DCP, while HyDra holds it under 1%. Appendix A traces both bubbles per stream on rank 0: static CP ends in one long

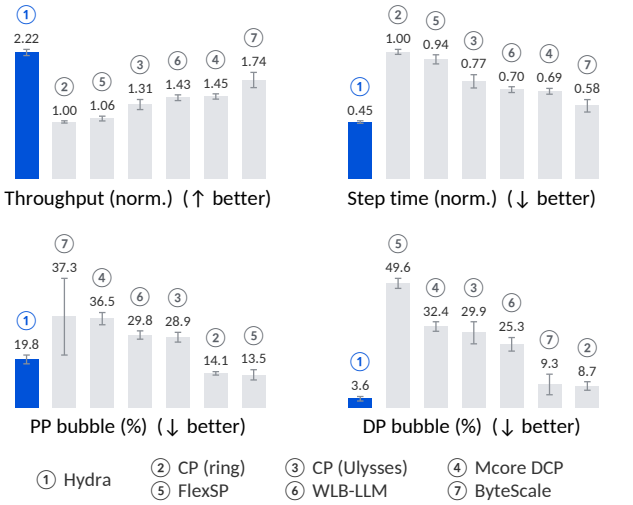
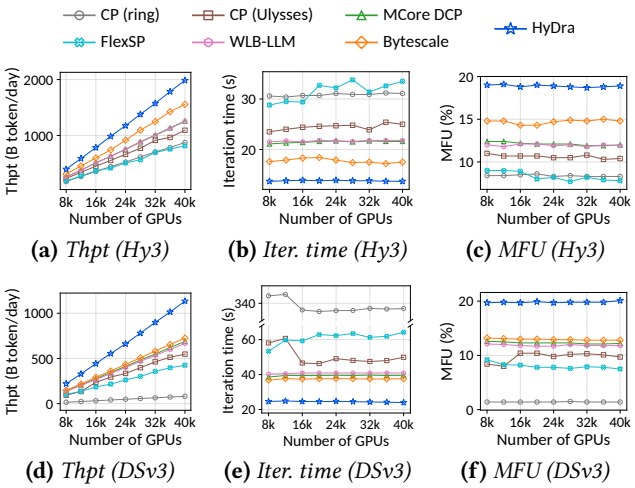


Figure 23. *Simulation results for Hy3 on 8K GPUs; throughput and step time are normalized to static CP (ring).*



(a) *Thpt (Hy3)* **(b)** *Iter. time (Hy3)* **(c)** *MFU (Hy3)*
(d) *Thpt (DSv3)* **(e)** *Iter. time (DSv3)* **(f)** *MFU (DSv3)*

Figure 24. *Weak-scaling performance.*

reduce-scatter, Mcore DCP replaces it with gaps recurring inside the step, and HyDra leaves neither.

Scheduling overhead. On the 2K-GPU production run, HyDra cuts online scheduling time from 0.78 s to 0.34 s, a 2.3× speedup over Mcore DCP.

Training cost. The throughput gap also translates into rental cost. A 2T-token budget[3] at the rates of Figure 20b takes 17.4 days under static CP, 13.7 under Mcore DCP, and 11.0 under HyDra. At $2 per GPU-hour [6], 2,048 GPUs cost $1.71M, $1.35M, and $1.08M, so HyDra saves $632K over static CP and $271K over Mcore DCP.

6.4 Scale-Out Simulation

Simulator fidelity. Larger scales use an in-house simulator that replays a batch through the full parallel schedule with operator costs measured on the same hardware. Validated against per-rank production traces at more than 11K GPUs, its steady-state iteration time lands inside the observed range in every configuration and within 10% of the measured mean.

Throughput and load balancing. Figure 23 compares the six systems on Hy3 [39] at 8K GPUs, where HyDra attains the highest throughput: it improves over the strongest baseline, ByteScale, by 1.28×, over Mcore DCP by 1.53×, over WLB-LLM by 1.56×, and over FlexSP and static CP by 1.7–2.2×. The win comes from balance rather than the best value on any single axis: static CP keeps both bubbles low but pays a heavy communication cost, leaving the longest step time; Mcore DCP cuts that cost yet runs high PP and DP bubbles (36.5% and 32.4%); ByteScale trims the DP bubble but leaves the largest PP bubble (37.3%); FlexSP trims the PP bubble but inflates the DP bubble to nearly half the iteration (49.6%). HyDra avoids static CP's communication cost while holding the PP bubble at 19.8% and the DP bubble at 3.6%, so it records the highest throughput.

Weak scaling. Figure 24 extends both the GQA model (Hy3) and an MLA model (DeepSeek-V3 [6]) to 40K GPUs, where HyDra stays the fastest at every scale. On Hy3 the 8K-GPU margins hold across the sweep: averaged over it, HyDra improves throughput by 1.28× over ByteScale, 1.53–1.56× over Mcore DCP and WLB-LLM, and 1.7–2.2× over FlexSP and static CP. Its lead is wider on DeepSeek-V3: 1.53× over ByteScale, 1.61× over Mcore DCP, 1.7–2.5× over the remaining schedulers, and up to 13.8× over ring static CP, whose flat per-rank floor must circulate MLA's full per-head K/V, many times the payload of the eight KV heads Hy3 shares. HyDra therefore leads on both attention architectures at every scale in the sweep.

7 Conclusion

We presented HyDra, a scalable load-driven DCP system for long-context training. Production DCP sizes each CP degree to fit memory, and our study on more than 11K GPUs shows the cost. Ranks with similar token counts differ by 5× in microbatch time, leaving a 46% pipeline bubble and a 13% data-parallel bubble. HyDra instead pulls every rank toward one load target, computed in closed form, and places sequences with lazy heaps. That target asks for larger CP degrees than memory requires, so HyDra nests a Ulysses group in a shallow ring, lowering computation and communication together. On a **512**-GPU testbed it raises throughput over Mcore DCP by 1.18× on average at 32K context and 2.48× at 256K, and on a **2,048**-GPU production job it cuts the pipeline bubble from 36% to 14% and raises throughput by 1.25× on average over Mcore DCP and 1.59× over static CP. We believe the principles behind HyDra can guide academia and industry in building efficient DCP systems for large-scale long-context training.

[3]The 2T-token budget is illustrative and does not reflect any production token count.

References

[1] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv:2305.13245 [cs.CL] https://arxiv.org/abs/2305.13245

[2] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. *arXiv preprint arXiv:2004.05150* (2020). https://doi.org/10.48550/arXiv.2004.05150

[3] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2978–2988. https://doi.org/10.18653/v1/P19-1285

[4] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations*.

[5] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems*, Vol. 35. 16344–16359.

[6] DeepSeek-AI. 2024. DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024).

[7] DeepSeek-AI. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. *arXiv preprint arXiv:2606.19348* (2026). https://arxiv.org/abs/2606.19348

[8] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, T. Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Liu, Xin Xie, Xingkai Yu, Xinnan Song, Xinyi Zhou, Xinyu Yang, Xuan Lu, Xuecheng Su, Y. Wu, Y. K. Li, Y. X. Wei, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Zheng, Yichao Zhang, Yiliang Xiong, Yilong Zhao, Ying He, Ying Tang, Yishi Piao, Yixin Dong, Yixuan Tan, Yiyuan Liu, Yongji Wang, Yongqiang Guo, Yuchen Zhu, Yuduan Wang, Yuheng Zou, Yukun Zha, Yunxian Ma, Yuting Yan, Yuxiang You, Yuxuan Liu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhewen Hao, Zhihong Shao, Zhiniu Wen, Zhipeng Xu, Zhongyu Zhang, Zhuoshu Li, Zihan Wang, Zihui Gu, Zilin Li, and Ziwei Xie. 2024. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[9] Hantian Ding, Zijian Wang, Giovanni Paolini, Varun Kumar, Anoop Deoras, Dan Roth, and Stefano Soatto. 2024. Fewer Truncations Improve Language Modeling. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

[10] Jiarui Fang and Shangchun Zhao. 2024. USP: A Unified Sequence Parallelism Approach for Long Context Generative AI. *arXiv preprint arXiv:2405.07719* (2024).

[11] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research (JMLR)* 23, 120 (2022), 1–39.

[12] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. In *Proceedings of Machine Learning and Systems*, Vol. 4. 721–747.

[13] Hao Ge, Junda Feng, Qi Huang, Fangcheng Fu, Xiaonan Nie, Lei Zuo, Haibin Lin, Bin Cui, and Xin Liu. 2025. ByteScale: Communication-Efficient Scaling of LLM Training with a 2048K Context Length on 16384 GPUs. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 963–978.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[15] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. 2023. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *arXiv preprint arXiv:2309.14509* (2023).

[16] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems*, Vol. 1. 1–13.

[17] Chenyu Jiang, Zhenkun Cai, Ye Tian, Zhen Jia, Yida Wang, and Chuan Wu. 2025. DCP: Addressing Input Dynamism in Long-Context Training via Dynamic Context Parallelism. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 221–236. https://doi.org/10.1145/3731569.3764849

[18] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[19] Kimi Team. 2026. Kimi K2.5. https://huggingface.co/moonshotai/Kimi-K2.5.

[20] Mario Michael Krell, Matej Kosec, Sergio P Perez, and Andrew Fitzgibbon. 2021. Efficient Sequence Packing without Cross-contamination: Accelerating Large Language Models without Impacting Performance. *arXiv preprint arXiv:2107.02027* (2021).

[21] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2021. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. In *International Conference on Learning Representations (ICLR)*.

[22] Dacheng Li, Rulin Shao, Anze Xie, Eric P. Xing, Xuezhe Ma, Ion Stoica, Joseph E. Gonzalez, and Hao Zhang. 2024. DISTFLASHATTN: Distributed Memory-efficient Attention for Long-context LLMs Training. In *First Conference on Language Modeling*. https://openreview.net/forum?id=pUEDkZyPDl

[23] Haoyang Li, Fangcheng Fu, Sheng Lin, Hao Ge, Xuanyu Wang, Jiawen Niu, Jinbao Xue, Yangyu Tao, Di Wang, Jie Jiang, and Bin Cui. 2025. Hydraulis: Balancing Large Transformer Model Training via Co-designing Parallel Strategies and Data Assignment. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–30. https://doi.org/10.1145/3769802

[24] Kunlun Li. 2026. Speeding Up Variable-Length Training with Dynamic Context Parallelism and NVIDIA Megatron Core. NVIDIA

Technical Blog. https://developer.nvidia.com/blog/speeding-up-variable-length-training-with-dynamic-context-parallelism-and-nvidia-megatron-core/

[25] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. 2023. Sequence Parallelism: Long Sequence Training from System Perspective. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. 2391–2404. https://doi.org/10.18653/v1/2023.acl-long.134

[26] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proceedings of the VLDB Endowment (VLDB)* 13, 12 (2020).

[27] Hao Liu and Pieter Abbeel. 2023. Blockwise Parallel Transformers for Large Context Models. In *Advances in Neural Information Processing Systems*, Vol. 36.

[28] Hao Liu, Matei Zaharia, and Pieter Abbeel. 2024. RingAttention with Blockwise Transformers for Near-Infinite Context. In *International Conference on Learning Representations (ICLR)*.

[29] Llama Team, AI @ Meta. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).

[30] Cheng Luo, Jiawei Zhao, Zhuoming Chen, Beidi Chen, and Anima Anandkumar. 2024. Mini-Sequence Transformers: Optimizing Intermediate Memory for Long Sequences Training. In *Advances in Neural Information Processing Systems*, Vol. 37. https://doi.org/10.52202/079017-3086

[31] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[33] Zihan Qiu, Zekun Wang, Bo Zheng, Zeyu Huang, Kaiyue Wen, Songlin Yang, Rui Men, Le Yu, Fei Huang, Suozhi Huang, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025. Gated Attention for Large Language Models: Non-linearity, Sparsity, and Attention-Sink-Free. arXiv:2505.06708 [cs.CL] https://arxiv.org/abs/2505.06708

[34] Qwen Team. 2026. Qwen3.6. https://huggingface.co/Qwen/Qwen3.6-27B.

[35] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[36] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems*, Vol. 31. 10435–10444.

[37] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[38] Konrad Staniszewski, Szymon Tworkowski, Sebastian Jaszczur, Yu Zhao, Henryk Michalewski, Łukasz Kuciński, and Piotr Miłoś. 2025. Structured Packing in LLM Training Improves Long Context Utilization. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 24 (2025), 25201–25209. https://doi.org/10.1609/aaai.v39i24.34706

[39] Tencent Hunyuan Team. 2026. Hy3: A Leading Reasoning and Agent Model with Great Cost Efficiency. https://github.com/Tencent-Hunyuan/Hy3. GitHub repository, accessed Jul. 25, 2026.

[40] Tencent Hunyuan Team. 2026. Hy4-preview. https://github.com/Tencent-Hunyuan/Hy4-preview. GitHub repository, accessed Sep. 10, 2026.

[41] Yujie Wang, Shiju Wang, Shenhan Zhu, Fangcheng Fu, Xinyi Liu, Xuefeng Xiao, Huixia Li, Jiashi Li, Faming Wu, and Bin Cui. 2025. FlexSP: Accelerating Large Language Model Training via Flexible Sequence Parallelism. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[42] Zheng Wang, Anna Cai, Xinfeng Xie, Zaifeng Pan, Yue Guan, Weiwei Chu, Jie Wang, Shikai Li, Jianyu Huang, Chris Cai, Yuchen Hao, and Yufei Ding. 2025. WLB-LLM: Workload-Balanced 4D Parallelism for Large Language Model Training. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[43] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. 2020. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems*, Vol. 33. 17283–17297.

[44] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, Joseph E Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[45] Zhipu AI. 2026. GLM-5. https://huggingface.co/zai-org/GLM-5.

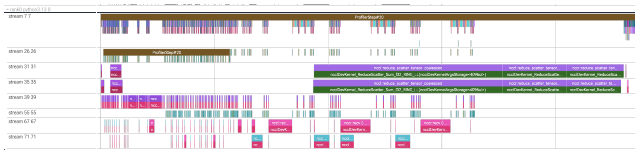
[46] Jiasheng Zhou, Longbin Zeng, Clavis Chen, Ruiming Lu, Qinwei Yang, Leyi Ye, Ray Ying, and Key Zhang. 2026. ARGUS: Production-Scale Tracing and Performance Diagnosis for over 10,000-GPU Clusters. *arXiv preprint arXiv:2606.20374* (2026).

A Rank-0 Execution Traces

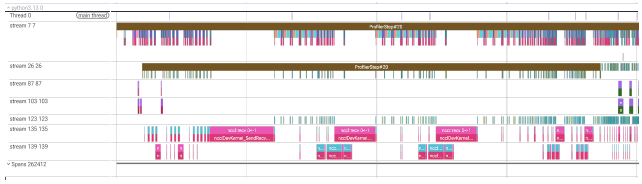
Figure 25 shows the rank-0 profiler trace of one steady-state iteration of the 2,048-GPU run of Section 6, under static CP, Mcore DCP, and HyDra. Each row is a CUDA stream: the compute stream carries the forward and backward kernels, and the remaining streams carry the NCCL collectives. Stalls are therefore visible as white space on the compute stream, and the trace localizes them to the barrier that causes them.

Static CP stalls at the gradient reduce-scatter. The iteration ends in one long collective: the `ReduceScatter` kernels span roughly the second half of the trace, by which time the compute stream has already gone quiet. A fixed degree never rebalances computation across sequences, so rank 0 finishes early and idles at the collective until the slowest replica arrives, the DP bubble of Section 3.3 at its most severe.

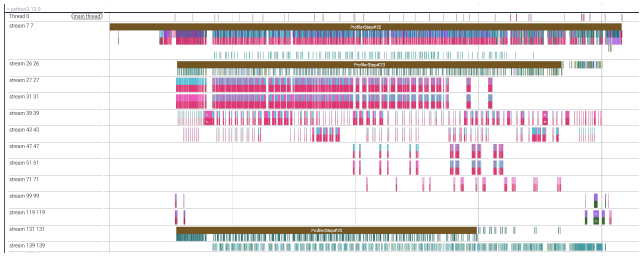
Mcore DCP trades it for pipeline idle. The closing collective shrinks, but white space appears *inside* the step, where the compute stream alternates between dense bursts of kernels and gaps of comparable width. A memory-driven degree leaves the heaviest MB heavy and every stage waits on that MB once per microbatch, so these PP bubbles recur through the iteration.



(a) *Static CP*



(b) *Mcore DCP*



(c) *HyDra*

Figure 25. *Rank-0 profiler trace of one steady-state iteration under each system.*

HyDra leaves no significant bubble on the compute stream. The bursts merge and the wide gaps disappear, leaving only the short intervals between consecutive kernels. Balancing MBs by attention computation removes the recurring pipeline gaps, and balancing inside each MB keeps the closing reduce-scatter short.