DualPipe

Deep Dive into DeepSeek — Chapter 29

Yin Yang

Foundation Books Project

The one rule of this chapter

A pipeline schedule may move *where* idle time appears.
It may never move *when* a dependency is allowed to be ready.

- Pipeline parallelism divides a model into ordered stages so no single device group holds the whole thing — that solves ownership.

The one rule of this chapter

A pipeline schedule may move *where* idle time appears.
It may never move *when* a dependency is allowed to be ready.

- Pipeline parallelism divides a model into ordered stages so no single device group holds the whole thing — that solves ownership.
- It creates a scheduling problem: early, middle, and late stages must stay busy despite forward and backward dependencies.

The one rule of this chapter

A pipeline schedule may move *where* idle time appears.
It may never move *when* a dependency is allowed to be ready.

- Pipeline parallelism divides a model into ordered stages so no single device group holds the whole thing — that solves ownership.
- It creates a scheduling problem: early, middle, and late stages must stay busy despite forward and backward dependencies.
- DualPipe changes the schedule's shape — bidirectional injection, overlap, deferred weight gradients — without relaxing a single dependency edge.

The one rule of this chapter

A pipeline schedule may move *where* idle time appears.
It may never move *when* a dependency is allowed to be ready.

- Pipeline parallelism divides a model into ordered stages so no single device group holds the whole thing — that solves ownership.
- It creates a scheduling problem: early, middle, and late stages must stay busy despite forward and backward dependencies.
- DualPipe changes the schedule's shape — bidirectional injection, overlap, deferred weight gradients — without relaxing a single dependency edge.

Goal: follow one four-stage, eight-microbatch training step through bubble accounting, the public repository surface, bidirectional scheduling, overlap, MoE interactions, memory tradeoffs, and measured evidence — and keep dependency legality, evidence tier, and measurement convention visible at every step.

Roadmap

Scheduling as ownership and throughput

Bubbles, the repository surface, and bidirectional scheduling

Overlap, MoE interactions, and memory tradeoffs

Comparing schedules, checking arithmetic, and from concept to code

Scheduling as ownership and throughput

Two problems, one abstraction

- **Ownership:** no single device group should hold and execute the whole model — pipeline parallelism divides it into ordered pieces.

Two problems, one abstraction

- **Ownership:** no single device group should hold and execute the whole model — pipeline parallelism divides it into ordered pieces.
- **Throughput:** early, middle, and late pieces must stay busy despite data dependencies — that is the scheduling problem.

Two problems, one abstraction

- **Ownership:** no single device group should hold and execute the whole model — pipeline parallelism divides it into ordered pieces.
- **Throughput:** early, middle, and late pieces must stay busy despite data dependencies — that is the scheduling problem.

DualPipe writes the ordered stage sequence as function composition:

$$f = f_{D_{\mathrm{pp}}-1} \circ f_{D_{\mathrm{pp}}-2} \circ \cdots \circ f_0,$$

where stage s owns subfunction f_s, receives activations from stage $s-1$ forward, and returns input gradients to stage $s-1$ backward.

Two problems, one abstraction

- **Ownership:** no single device group should hold and execute the whole model — pipeline parallelism divides it into ordered pieces.
- **Throughput:** early, middle, and late pieces must stay busy despite data dependencies — that is the scheduling problem.

DualPipe writes the ordered stage sequence as function composition:

$$f = f_{D_{\mathrm{pp}}-1} \circ f_{D_{\mathrm{pp}}-2} \circ \cdots \circ f_0,$$

where stage s owns subfunction f_s, receives activations from stage $s-1$ forward, and returns input gradients to stage $s-1$ backward.

A **pipeline rank** is the execution owner (process or device group); a **stage** is a model partition. DualPipe's own example pairs a module from one end of the pipeline with a module from the opposite end on each rank — so a rank need not equal one logical stage.

The running example, fixed for the whole chapter

Quantity	Value in the running example
D_{pp}	4 logical stages, S_0, S_1, S_2, S_3
Microbatches	8, named $m_0, \ldots, m_7$

- A naive serial run of one microbatch across four stages needs four forward and four backward stage operations.

The running example, fixed for the whole chapter

Quantity	Value in the running example
D_{pp}	4 logical stages, S_0, S_1, S_2, S_3
Microbatches	8, named $m_0, \ldots, m_7$
$F(s, i)$, $B(s, i)$	forward / backward work item, stage s, microbatch i
W	weight-gradient (backward-for-weights) work item

- A naive serial run of one microbatch across four stages needs four forward and four backward stage operations.

The running example, fixed for the whole chapter

Quantity	Value in the running example
D_{pp}	4 logical stages, S_0, S_1, S_2, S_3
Microbatches	8, named $m_0, \ldots, m_7$
$F(s, i)$, $B(s, i)$	forward / backward work item, stage s, microbatch i
W	weight-gradient (backward-for-weights) work item

- A naive serial run of one microbatch across four stages needs four forward and four backward stage operations.
- Interleaving eight microbatches keeps more stages busy at once — but the first and last few slots cannot fill or keep every stage busy. Those idle positions are the pipeline bubbles this chapter studies.

Bubbles, the repository surface, and bidirectional scheduling

Bubble slot: a narrow definition, on purpose

Definition (Bubble slot)
An exposed stage-time position in which a stage or rank has no executable scheduled work because a dependency, communication, or local resource constraint prevents useful pipeline progress.

Bubble slot: a narrow definition, on purpose

Definition (Bubble slot)
An exposed stage-time position in which a stage or rank has no executable scheduled work because a dependency, communication, or local resource constraint prevents useful pipeline progress.

For D_{pp} stages and step length T_{step}:

$$T_{\mathrm{bubble}} = \sum_{s=0}^{D_{\mathrm{pp}}-1} \mathrm{idle_slots}(s), \qquad \text{bubble fraction} = \frac{T_{\mathrm{bubble}}}{D_{\mathrm{pp}}\, T_{\mathrm{step}}}.$$

Bubble slot: a narrow definition, on purpose

Definition (Bubble slot)
An exposed stage-time position in which a stage or rank has no executable scheduled work because a dependency, communication, or local resource constraint prevents useful pipeline progress.

For D_{pp} stages and step length T_{step}:

$$T_{\mathrm{bubble}} = \sum_{s=0}^{D_{\mathrm{pp}}-1} \mathrm{idle_slots}(s), \qquad \text{bubble fraction} = \frac{T_{\mathrm{bubble}}}{D_{\mathrm{pp}}\, T_{\mathrm{step}}}.$$

Worked toy example: a 4-stage, 14-slot timeline with 12 idle stage-slots. Total budget $4 \cdot 14 = 56$; bubble fraction $12/56 = 3/14 \approx 0.214$. [DERIVED: book's own toy timeline, Section 3]

README bubble and memory comparison

Method	Bubble	Param	Activation	Devices
1F1B	$(P_{\mathrm{P}}-1)(F+B)$	$1\times$	P_{P}	P_{P}
ZB1P	$(P_{\mathrm{P}}-1)(F+B-2W)$	$1\times$	P_{P}	P_{P}
DualPipe	$(P_{\mathrm{P}}/2-1)(F\&B+B-3W)$	$2\times$	$P_{\mathrm{P}}+1$	P_{P}

README bubble and memory comparison

Method	Bubble	Param	Activation	Devices
1F1B	$(P_{\mathrm{P}}-1)(F+B)$	$1\times$	P_{P}	P_{P}
ZB1P	$(P_{\mathrm{P}}-1)(F+B-2W)$	$1\times$	P_{P}	P_{P}
DualPipe	$(P_{\mathrm{P}}/2-1)(F\&B+B-3W)$	$2\times$	$P_{\mathrm{P}}+1$	P_{P}
DualPipeV	$(P_{\mathrm{P}}/2-1)(F\&B+B-3W)$	$2\times$	$P_{\mathrm{P}}+1$	$P_{\mathrm{P}}/2$

README bubble and memory comparison

Method	Bubble	Param	Activation	Devices
1F1B	$(P_{\mathrm{P}}-1)(F+B)$	$1\times$	P_{P}	P_{P}
ZB1P	$(P_{\mathrm{P}}-1)(F+B-2W)$	$1\times$	P_{P}	P_{P}
DualPipe	$(P_{\mathrm{P}}/2-1)(F\&B+B-3W)$	$2\times$	$P_{\mathrm{P}}+1$	P_{P}
DualPipeV	$(P_{\mathrm{P}}/2-1)(F\&B+B-3W)$	$2\times$	$P_{\mathrm{P}}+1$	$P_{\mathrm{P}}/2$

[INSPECTED: DualPipe README] These are the README's own comparison formulas under its stated same-stage-count convention — not measurements from this book.

README bubble and memory comparison

Method	Bubble	Param	Activation	Devices
1F1B	$(P_{\mathrm{P}}-1)(F+B)$	$1\times$	P_{P}	P_{P}
ZB1P	$(P_{\mathrm{P}}-1)(F+B-2W)$	$1\times$	P_{P}	P_{P}
DualPipe	$(P_{\mathrm{P}}/2-1)(F\&B+B-3W)$	$2\times$	$P_{\mathrm{P}}+1$	P_{P}
DualPipeV	$(P_{\mathrm{P}}/2-1)(F\&B+B-3W)$	$2\times$	$P_{\mathrm{P}}+1$	$P_{\mathrm{P}}/2$

[INSPECTED: DualPipe README] These are the README's own comparison formulas under its stated same-stage-count convention — not measurements from this book.

DualPipe and DualPipeV share the listed bubble expression; they differ in device count. A lower symbolic bubble term is not a speed claim.

Five surfaces, five different technical objects

Surface	What it establishes
V3 technical report	Reported system behavior, no released engine.
DualPipe README	Reference schedule arithmetic, no measured execution.
Python repository	Executable schedule + correctness examples, with scaffold limitations.
`profile-data/train.json`	A captured overlap window under declared omissions.
Production implementation	Not released; callbacks and topology unspecified.

Five surfaces, five different technical objects

Surface	What it establishes
V3 technical report	Reported system behavior, no released engine.
DualPipe README	Reference schedule arithmetic, no measured execution.
Python repository	Executable schedule + correctness examples, with scaffold limitations.
`profile-data/train.json`	A captured overlap window under declared omissions.
Production implementation	Not released; callbacks and topology unspecified.

`DualPipe.step` asserts even ranks, even positive chunks, and `num_chunks` $\geq 2 \cdot$ `num_ranks`. `DualPipeV.step` keeps the chunk bound but drops the parity assertions. [INSPECTED: DualPipe `dualpipe.py`, `dualpipev.py`]

Injecting work from both ends

Definition (Bidirectional pipeline schedule)
A schedule in which different streams of microbatch work traverse the ordered rank line from opposite ends, while each stream still preserves its forward, backward, and parameter-gradient dependencies.



Injecting work from both ends

Definition (Bidirectional pipeline schedule)
A schedule in which different streams of microbatch work traverse the ordered rank line from opposite ends, while each stream still preserves its forward, backward, and parameter-gradient dependencies.

one direction



two phases



Injecting work from both ends

Definition (Bidirectional pipeline schedule)
A schedule in which different streams of microbatch work traverse the ordered rank line from opposite ends, while each stream still preserves its forward, backward, and parameter-gradient dependencies.



Reading slot 1 across all four ranks: R_0 advances $F_0(0,1)$; R_1 consumes the previous phase-0 activation as $F_0(1,0)$; on the mirrored half, R_2 consumes the symmetric phase-1 activation as $F_1(2,4)$; R_3 advances $F_1(3,5)$.

Injecting work from both ends

Definition (Bidirectional pipeline schedule)
A schedule in which different streams of microbatch work traverse the ordered rank line from opposite ends, while each stream still preserves its forward, backward, and parameter-gradient dependencies.



Reading slot 1 across all four ranks: R_0 advances $F_0(0,1)$; R_1 consumes the previous phase-0 activation as $F_0(1,0)$; on the mirrored half, R_2 consumes the symmetric phase-1 activation as $F_1(2,4)$; R_3 advances $F_1(3,5)$.

Same four-slot prefix that had six idle cells in a one-direction fill has only two here. Rank r still names the execution owner; phase ϕ is the directional stream; neither relabels logical stage S_r.

[DERIVED: book's toy fill comparison, Section 5]

Overlap, MoE interactions, and memory tradeoffs

An $F\&B$ cell is an implementation assumption

Definition (Overlapped forward/backward chunk)
A schedule position where a forward chunk and a backward chunk are intended to make concurrent progress on a rank or local module pair.

- Legal only if both incoming tensors (activation, gradient) are ready *and* the local modules implement `overlapped_forward_backward`.

An *F&B* cell is an implementation assumption

Definition (Overlapped forward/backward chunk)
A schedule position where a forward chunk and a backward chunk are intended to make concurrent progress on a rank or local module pair.

- Legal only if both incoming tensors (activation, gradient) are ready *and* the local modules implement `overlapped_forward_backward`.
- `WeightGradStore` queues weight-gradient closures in a FIFO so backward-input work can unblock an earlier stage before the local parameter-gradient contribution is flushed.

An *F&B* cell is an implementation assumption

Definition (Overlapped forward/backward chunk)
A schedule position where a forward chunk and a backward chunk are intended to make concurrent progress on a rank or local module pair.

- Legal only if both incoming tensors (activation, gradient) are ready *and* the local modules implement `overlapped_forward_backward`.
- `WeightGradStore` queues weight-gradient closures in a FIFO so backward-input work can unblock an earlier stage before the local parameter-gradient contribution is flushed.

At the pinned revision, the released example callback runs forward then backward *sequentially* — it validates correctness, not concurrent execution. [INSPECTED: DualPipe `dualpipe.py`, `utils.py`]

A slow MoE chunk becomes pipeline-visible

- DeepEP controls dispatch/combine; EPLB controls physical expert placement and redundancy; DualPipe schedules the resulting stage work.

S_2 slow MoE

A slow MoE chunk becomes pipeline-visible

- DeepEP controls dispatch/combine; EPLB controls physical expert placement and redundancy; DualPipe schedules the resulting stage work.
- If S_2's expert dispatch is imbalanced, S_3 waits on a late activation and S_1 waits on a late gradient — *one* slow chunk creates idle time in *two* directions.

S_2 slow MoE

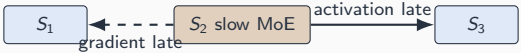
A slow MoE chunk becomes pipeline-visible

- DeepEP controls dispatch/combine; EPLB controls physical expert placement and redundancy; DualPipe schedules the resulting stage work.
- If S_2's expert dispatch is imbalanced, S_3 waits on a late activation and S_1 waits on a late gradient — *one* slow chunk creates idle time in *two* directions.



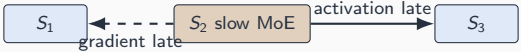
A slow MoE chunk becomes pipeline-visible

- DeepEP controls dispatch/combine; EPLB controls physical expert placement and redundancy; DualPipe schedules the resulting stage work.
- If S_2's expert dispatch is imbalanced, S_3 waits on a late activation and S_1 waits on a late gradient — *one* slow chunk creates idle time in *two* directions.



A slow MoE chunk becomes pipeline-visible

- DeepEP controls dispatch/combine; EPLB controls physical expert placement and redundancy; DualPipe schedules the resulting stage work.
- If S_2's expert dispatch is imbalanced, S_3 waits on a late activation and S_1 waits on a late gradient — *one* slow chunk creates idle time in *two* directions.



The schedule does not cause the imbalance; it exposes it as stage-time loss.

Bubble reduction is not free

Activation residency

- Live activation records at stage S_1: 4 before any backward returns, 2 once two backward chunks return.
- Recomputation trades memory for extra compute.

Bubble reduction is not free

Activation residency

- Live activation records at stage S_1: 4 before any backward returns, 2 once two backward chunks return.
- Recomputation trades memory for extra compute.

Parameter ownership

- README: $1\times$ per device for 1F1B, $2\times$ for DualPipe/DualPipeV under matched stage count.
- Reducing idle slots is compared against a larger local parameter footprint.

Bubble reduction is not free

Activation residency

- Live activation records at stage S_1: 4 before any backward returns, 2 once two backward chunks return.
- Recomputation trades memory for extra compute.

Parameter ownership

- README: 1× per device for 1F1B, 2× for DualPipe/DualPipeV under matched stage count.
- Reducing idle slots is compared against a larger local parameter footprint.

A bubble table without a memory column is incomplete. [INSPECTED: DualPipe README]

Comparing schedules, checking arithmetic, and from concept to code

1F1B, DualPipe, DualPipeV — matched conventions only

Field	1F1B-style	DualPipe-style	DualPipeV-style
Geometry	one-direction stream	bidirectional	V-shaped
Devices (README)	P_{P}	P_{P}	$P_{\mathrm{P}}/2$

1F1B, DualPipe, DualPipeV — matched conventions only

Field	1F1B-style	DualPipe-style	DualPipeV-style
Geometry	one-direction stream	bidirectional	V-shaped
Devices (README)	P_{P}	P_{P}	$P_{\mathrm{P}}/2$
Step condition	none (toy legality)	even ranks/chunks, $M \geq 2D_{\mathrm{pp}}$	$M \geq 2D_{\mathrm{pp}}$, no parity

1F1B, DualPipe, DualPipeV — matched conventions only

Field	1F1B-style	DualPipe-style	DualPipeV-style
Geometry	one-direction stream	bidirectional	V-shaped
Devices (README)	P_{P}	P_{P}	$P_{\mathrm{P}}/2$
Step condition	none (toy legality)	even ranks/chunks, $M \geq 2D_{\mathrm{pp}}$	$M \geq 2D_{\mathrm{pp}}$, no parity

Symbolic formulas, repository step conditions, and measured behavior answer three different questions. A three-way row is only comparable when device count, memory accounting, and measurement convention are matched across it. [DERIVED: book's comparison table, Section 9]

From toy legality to measured evidence

Quantity	Baseline	DualPipe-style	Convention
Idle slots (32-cell window)	6	2	unit F/B, folded comm.

From toy legality to measured evidence

Quantity	Baseline	DualPipe-style	Convention
Idle slots (32-cell window)	6	2	unit F/B, folded comm.
Bubble fraction	0.1875	0.0625	deterministic simulator

[DERIVED: book's deterministic simulator, Section 10]

From toy legality to measured evidence

Quantity	Baseline	DualPipe-style	Convention
Idle slots (32-cell window)	6	2	unit F/B, folded comm.
Bubble fraction	0.1875	0.0625	deterministic simulator

[DERIVED: book's deterministic simulator, Section 10]

A retained two-node, four-H200 run: exact loss/output match, max gradient difference 3.64×10^{-12}; temporal concurrency present in every trace, yet concurrent medians ran 1.95–3.80% *slower* than serialized. [OBSERVED: Volume II Panther H200 campaigns, Section 10]

From toy legality to measured evidence

Quantity	Baseline	DualPipe-style	Convention
Idle slots (32-cell window)	6	2	unit F/B, folded comm.
Bubble fraction	0.1875	0.0625	deterministic simulator

[DERIVED: book's deterministic simulator, Section 10]

A retained two-node, four-H200 run: exact loss/output match, max gradient difference 3.64×10^{-12}; temporal concurrency present in every trace, yet concurrent medians ran 1.95–3.80% *slower* than serialized. [OBSERVED: Volume II Panther H200 campaigns, Section 10]

Correctness and concurrency are not the same finding as speedup.

From concept to code: the executable step conditions

The chapter's integer constraints are literal asserts in the reference scheduler, not paraphrased prose.

from `DualPipe/dualpipe/dualpipe.py`

```
num_ranks = self.num_ranks
assert num_ranks % 2 == 0
assert num_chunks > 0 and num_chunks % 2 == 0 \
    and num_chunks >= num_ranks * 2, f"{num_chunks=}, {num_ranks=}"
```

From concept to code: the executable step conditions

The chapter's integer constraints are literal asserts in the reference scheduler, not paraphrased prose.

from **DualPipe/dualpipe/dualpipe.py**

```
num_ranks = self.num_ranks
assert num_ranks % 2 == 0
assert num_chunks > 0 and num_chunks % 2 == 0 \
    and num_chunks >= num_ranks * 2, f"{num_chunks=}, {num_ranks=}"
```

With four ranks, the smallest legal chunk count is eight — exactly the running example's microbatch count. `DualPipeV.step` keeps the `num_chunks` $\geq 2 \cdot$ `num_ranks` bound but drops both parity asserts, so odd ranks or chunks are legal there. [INSPECTED: DualPipe `dualpipe.py`, `dualpipev.py`]

Concept-to-code map for the chapter

- **rank halves, phase labels, integer step conditions** → `DualPipe/dualpipe/dualpipe.py`

Concept-to-code map for the chapter

- **rank halves, phase labels, integer step conditions** → `DualPipe/dualpipe/dualpipe.py`
- **V-shaped phase order, relaxed step condition** → `DualPipe/dualpipe/dualpipev.py`

Concept-to-code map for the chapter

- **rank halves, phase labels, integer step conditions** → `DualPipe/dualpipe/dualpipe.py`
- **V-shaped phase order, relaxed step condition** → `DualPipe/dualpipe/dualpipev.py`
- **asynchronous neighbor transfers** → `DualPipe/dualpipe/comm.py`
- **deferred weight-gradient FIFO** → `DualPipe/dualpipe/utils.py`

Concept-to-code map for the chapter

- **rank halves, phase labels, integer step conditions** → `DualPipe/dualpipe/dualpipe.py`
- **V-shaped phase order, relaxed step condition** → `DualPipe/dualpipe/dualpipev.py`
- **asynchronous neighbor transfers** → `DualPipe/dualpipe/comm.py`
- **deferred weight-gradient FIFO** → `DualPipe/dualpipe/utils.py`
- **schedule dependency checker** → `code/chapter_dualpipe/dualpipe_schedule_reference.py`

Concept-to-code map for the chapter

- **rank halves, phase labels, integer step conditions** → `DualPipe/dualpipe/dualpipe.py`
- **V-shaped phase order, relaxed step condition** → `DualPipe/dualpipe/dualpipev.py`
- **asynchronous neighbor transfers** → `DualPipe/dualpipe/comm.py`
- **deferred weight-gradient FIFO** → `DualPipe/dualpipe/utils.py`
- **schedule dependency checker** → `code/chapter_dualpipe/dualpipe_schedule_reference.py`

Fix the dependency contract first; then decide where the idle time goes.

Next: EPLB determines which rank actually runs scheduled expert work, and 3FS turns retained pipeline state into checkpoint payload. The inference-cost chapter's measurement protocol, and SGLang's serving-time schedule, both stay independent of this training pipeline.