LangGraph (Parallel Workflows) — Intuitive English Notes & Key Points

1) What this video covers

- **Recap:** Earlier videos = concepts + sequential (linear) flows in LangGraph.
- **Today's goal:** Build **parallel** workflows in LangGraph with **two examples**:
  1. **Cricket metrics** (non-LLM) — three calculations in parallel, then summarize.
  2. **UPSC essay evaluator** (LLM) — three parallel LLM judgments + a final reducer node with structured outputs.

---

2) Core ideas you must carry

- **State-first design:** Define a `TypedDict` describing all fields your flow reads/writes.
- **Nodes are Python functions:** Signature is `state_in -> (partial) state_out`.
- **Parallel needs partial updates:** In parallel branches, **return only the keys you changed**, not the whole state → avoids merge conflicts ("invalid update" error).
- **Reducers for merges:** When multiple parallel nodes update the **same key**, define a **reducer function** (e.g., list concatenation) so LangGraph knows **how** to merge.
- **Structured output for LLMs:** Use LangChain's structured outputs (e.g., Pydantic schemas `+ with_structured_output`) so every LLM reply has the exact fields you expect.

---

3) Example A — Cricket Metrics (Parallel, non-LLM)

Objective

Given an innings (`runs`, `balls`, `fours`, `sixes`), compute **in parallel**:

- **Strike rate** = `100 * runs / balls`
- **Boundary %** = `100 * (4*fours + 6*sixes) / runs`
- **Balls per boundary (BPB)** = `balls / (fours + sixes)`
  Then assemble a **summary** string.

State (TypedDict)

```
runs: int
balls: int
fours: int
sixes: int
strike_rate: float
boundary_percent: float
```

```
  bpb: float
  summary: str
```

Graph (ASCII)

```
START
  ├─▶ calculate_strike_rate ─┐
  ├─▶ calculate_boundary_% ──┤
  └─▶ calculate_bpb ─────────┤
                             ▼
                          summary
                             ▼
                            END
```

Pitfall & Fix

- **Error:** "Invalid update... key X can receive only one value per step."
  - Happens if each parallel node returns the **entire state**, which looks like competing writes.
- **Fix: Partial updates** only. Each node returns `{ "strike_rate": ... }` or `{ "bpb": ... }` etc. The `summary` node then reads these computed fields.

---

4) Example B — UPSC Essay Evaluator (Parallel, LLM + reducers)

Objective

Take an `essay` and evaluate on **three aspects in parallel** using LLMs:

1. **Language quality**
2. **Depth of analysis**
3. **Clarity of thought**

Each aspect returns:

- `feedback: str` (text comments)
- `score: int` (0–10)

Then a **final_evaluation** node:

- **Summarizes** the three feedbacks into one overall paragraph (LLM).
- **Averages** the three scores into a single `average_score` .

State (TypedDict)

```
essay_text: str
language_feedback: str
analysis_feedback: str
clarity_feedback: str
overall_feedback: str
individual_scores: list[int]   # merged via reducer (append/concat)
average_score: float
```

Using structured outputs (LangChain)

- Define Pydantic model:

  ```
  class EvaluationSchema(BaseModel):
      feedback: str = Field(description="Detailed feedback for the essay.")
      score: int = Field(ge=0, le=10, description="Score out of 10.")
  ```

- Get model that **forces** this schema:

  ```
  structured_model = chat.with_structured_output(EvaluationSchema)
  ```

- Each aspect node prompts and returns **only**:

  ```
  {
    "language_feedback": output.feedback,  # or analysis/clarity
    "individual_scores": [output.score]     # list for reducer merge
  }
  ```

Reducer (for merging parallel scores)

- Need all three scores collected into `individual_scores`.
- Declare field with a **reducer** that concatenates lists (conceptually `operator.add`):
  - Each node emits `{"individual_scores": [score]}`
  - Reducer merges to `[s1] + [s2] + [s3] -> [s1, s2, s3]`.

Graph (ASCII)

```
START
  ├─▶ evaluate_language ─┐
  ├─▶ evaluate_analysis ─┼─▶ final_evaluation ─▶ END
  └─▶ evaluate_clarity ──┘
```

Final node behavior

- Prompt LLM to **summarize**:

```
"Based on these feedbacks, write a concise overall evaluation:
 - Language: {language_feedback}
 - Analysis: {analysis_feedback}
 - Clarity: {clarity_feedback}"
```

- Compute:

```
average_score = sum(individual_scores) / len(individual_scores)
```

- Return partial:

```
{"overall_feedback": ..., "average_score": ...}
```

---

5) Implementation Patterns (copy-ready mental templates)

A. Node function pattern (parallel-safe)

```
def node_name(state: MyState) -> dict: # read what you need # compute result return
{"only_key_you_changed": value} # partial update
```

B. Safer default rule

- **Use partial returns everywhere** (sequential **and** parallel).
  Works in both settings and prevents accidental conflicts.

C. When to add reducers

- Multiple parallel branches update **the same key**
  → define a **merge strategy** (e.g., list concat, max, min, sum).

Examples:

- `individual_scores` : list concat
- `citations` : set union
- `tokens_used` : numeric sum

---

6) Quick Prompts (you can reuse)

Aspect prompts (structured model):

- *Language:*
  "Evaluate the **language quality** of the following essay. Provide **feedback** and a **score (0–10)**."
- *Depth of Analysis:*
  "Evaluate the **depth of analysis**… Provide **feedback** and a **score (0–10)**."
- *Clarity of Thought:*
  "Evaluate the **clarity of thought**… Provide **feedback** and a **score (0–10)**."

Final summary prompt (plain chat model):

- "Based on these three feedbacks (Language/Analysis/Clarity), write a **concise overall evaluation** for the essay."

---

7) Mini-Checklist (design & debugging)

- Did you **enumerate the state** fields up front?
- Do parallel nodes **return partial dicts**?
- Do shared keys across branches have a **reducer**?
- Are LLM outputs **structured** (schema) when you need reliability?
- After `graph.add_node()` and `add_edge()`, did you `compile()` and test with a small **initial_state**?

---

8) Why this matters

- **Parallelism** = better throughput & latency when steps are independent.
- **State + reducers** = determinism & debuggability in complex agent flows.
- **Structured outputs** = robust LLM pipelines that don't break on formatting.