LangGraph Execution Semantics

Concurrent branch specification — no parallel branch execution

Christoph Bussler 11 min read · Just now




LangGraph supports the definition and the execution of graphs composed of nodes that are connected by directional dependencies. It is a type-instance system where a graph type is defined and compiled and the compiled graph (type) can be executed subsequently as often as required (instances).

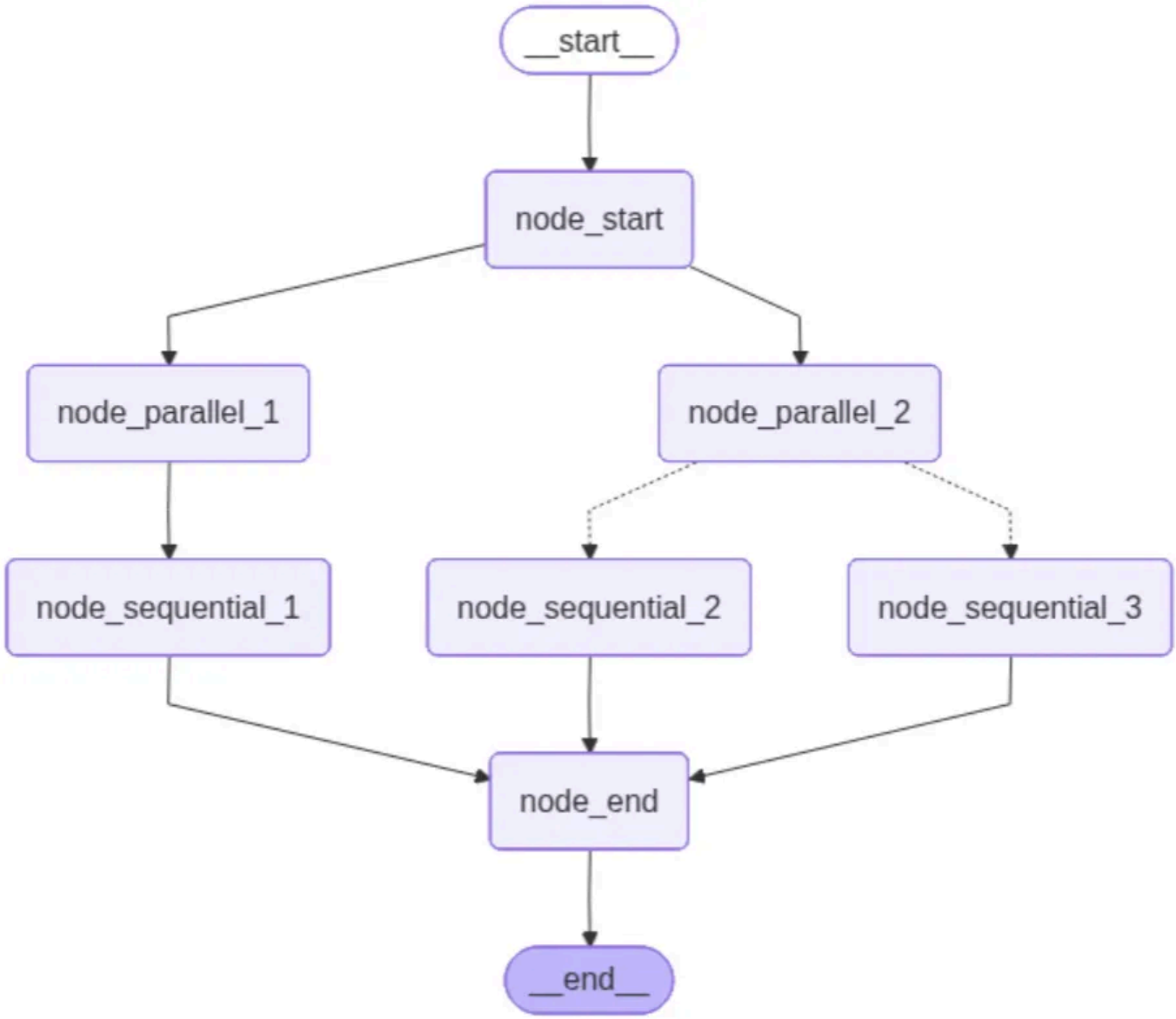
LangGraph is used for workflow definitions in context of AI use cases, but it is not limited to AI in practice; LangGraph can be deployed in non-AI use cases as well.

Two nodes `A`, `B` with a dependency from node `A` to node `B` means that node `A` is executed first, then node `B` — node `B` depends on the successful execution of node `A`. Conditional dependencies between nodes and splits into concurrent branches are supported as well.

Graphs have a graph scoped data state that is read and write accessible to the nodes in a graph so that data can be managed by nodes. The state is shared between the nodes, and this supports (indirect) data communicated between nodes. Each graph instance has its own data state instance scoped to the graph instance. The data and state aspect is not further discussed in the following.

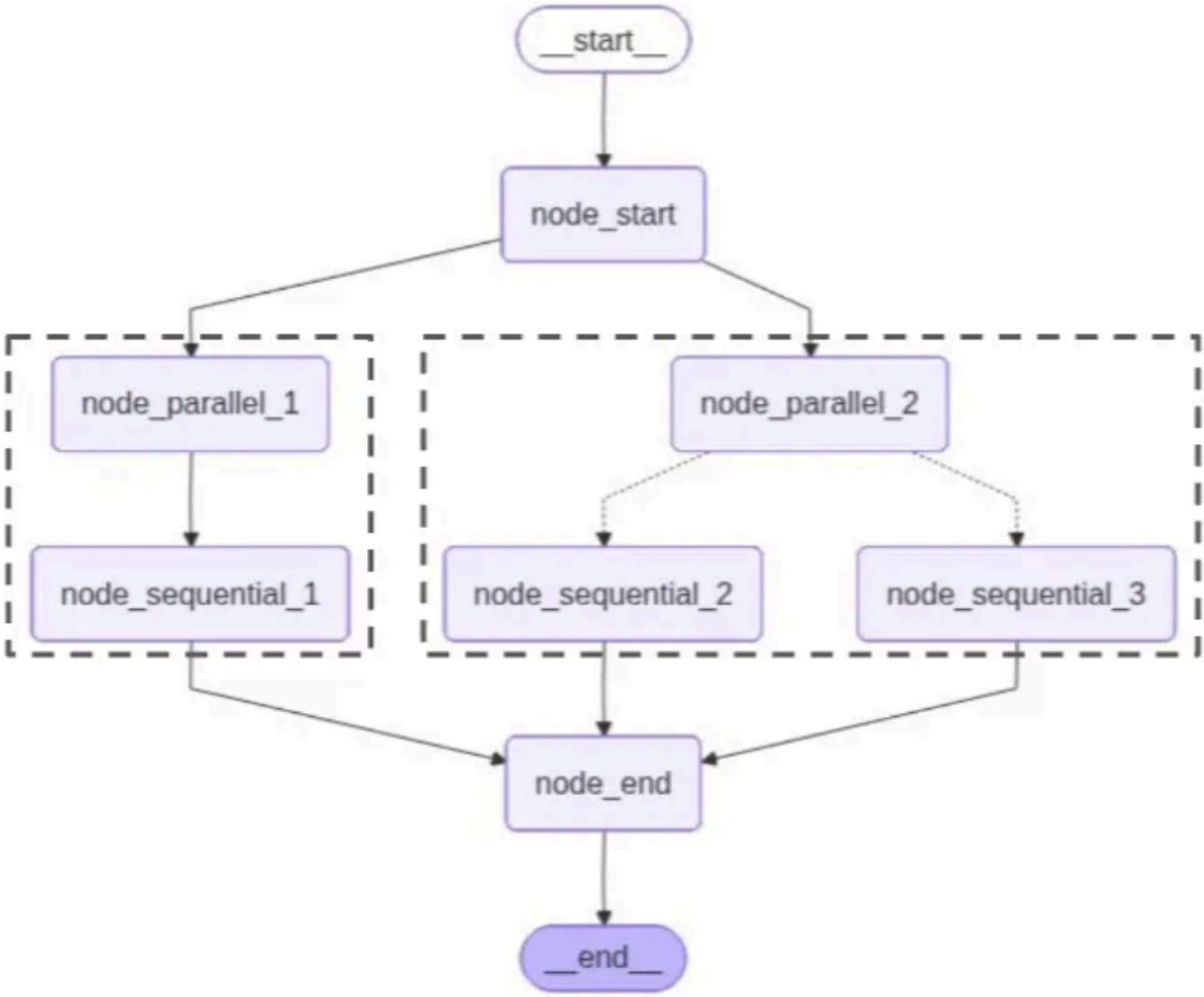
Concurrent branches in a graph

LangGraphs can have concurrent branches. An example graph specification is as follows:



In this graph there are two concurrent branches, `node_parallel_1` → `node_sequential_1` and `node_parallel_2` → `node_sequential_2` or `node_sequential_3`. The latter two are conditional: a condition decides which of `node_sequential_2` and `node_sequential_3` is executed. Graphically a conditional branch is depicted as dotted dependencies.

The following diagram marks the two concurrent branches.



The two branches are concurrent to each other as no dependencies exist from the nodes of one branch to the nodes of the other branch. The node `node_start` has two outgoing dependencies that implement a concurrent split; the node `node_end` has three incoming dependencies indicating that node `node_end` is only executed after nodes `node_sequential_1` and `node_sequential_2` or `node_sequential_3` are completed (`node_end` it is a join node).

I thought initially that the two concurrent branches in the graph type specification will be processed in parallel during graph instance execution. For example, as soon as node `node_parallel_2` is processed my expectation was that node `node_sequential_2` or `node_sequential_3` starts executing immediately, even if node `node_parallel_1` in the other concurrent branch is still executing and not completed yet.

However, this is not the case: parallel execution of concurrent branches does not take place in LangGraph.

Execution semantics

During experimentation I realized that the graph is executed based on sets of activated nodes following the idea behind Pregel (see later in the blog for references). An activated node is a node where all incoming dependencies are fulfilled (https://langchain-ai.github.io/langgraph/concepts/pregel/); this is automatically true for the start node.

The algorithm of node execution is as follows (for all nodes in a single graph instance): once all activated nodes are determined (first set of activated nodes), those are executed. Their successful execution might activate further nodes (second set of activated nodes), until no further nodes are activated; then execution is completed.

For example, if in the case above node `node_start` is a first set of activated nodes. When it completes, the second set of activated nodes is `node_parallel_1` and `node_parallel_2`. This is determined by the set of outgoing edges from `node_start`. Once `node_start` completes, its outgoing edges activate the nodes to which the edges are pointing to. If `node_parallel_1` is completed, it fulfills the dependency of node `node_sequential_1`;

however, `node_sequential_1` is not immediately executed until all currently activated nodes are completed.

All nodes in the first set of nodes are completed first. Once all nodes in the first set are completed, the second set of activated nodes is determined: for all nodes in the graph, which nodes have fulfilled incoming dependencies. Once all activated nodes are determined, the execution starts with the second set of activated nodes, and all nodes in that second set are executed. This in turn will create a third set of activated nodes (which might be empty if all nodes of a graph are completed, or no more nodes are activated).

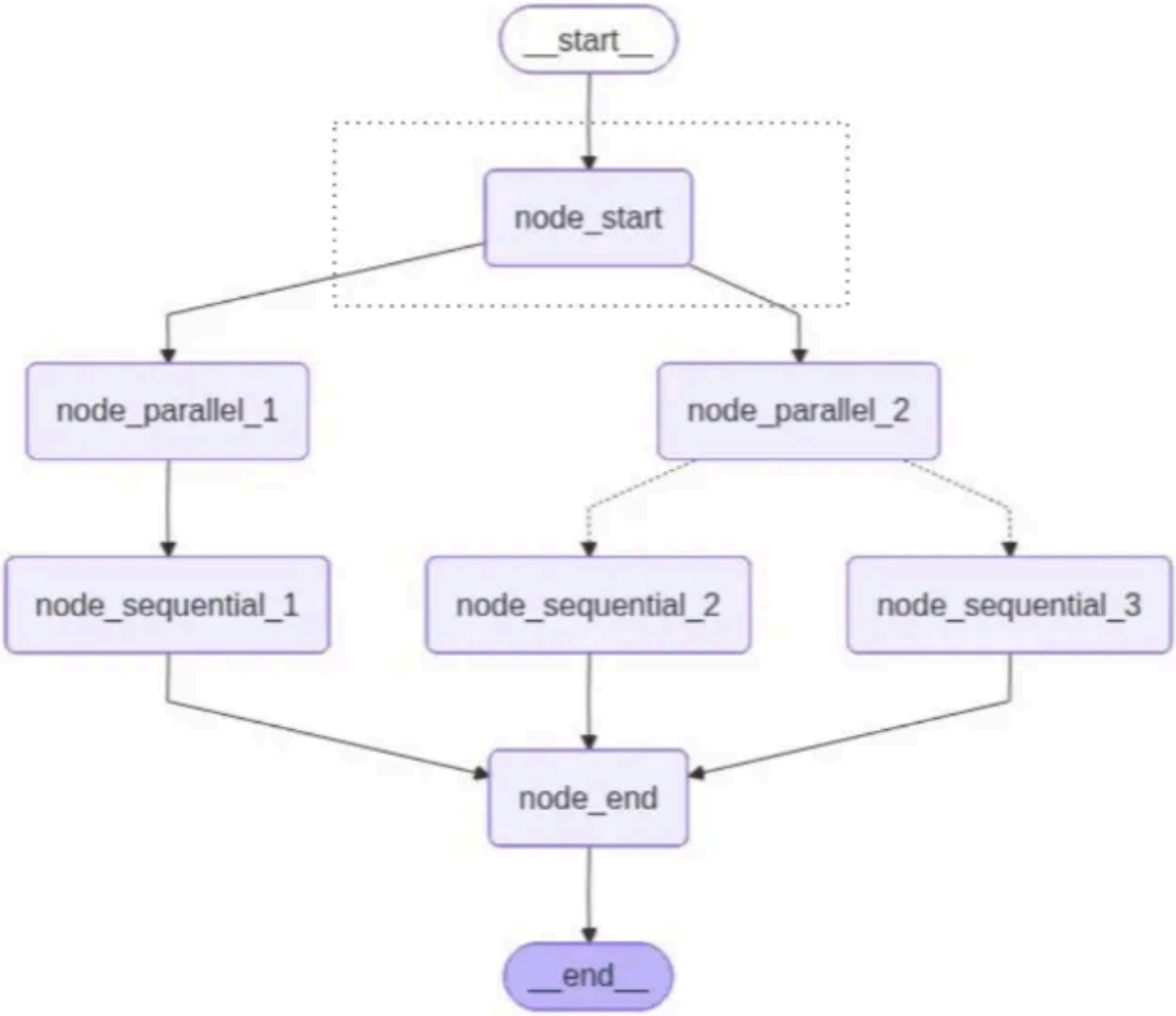
In the above example, the second set of activated nodes is `node_parallel_1` and `node_parallel_2`. Only once both are executed, the next set is determined and executed. Even if the dependencies of `node_sequential_1` are fulfilled (only one in the above case), it will not be executed immediately.

If conditional branching is present following a node like `node_parallel_2`, it is evaluated after the execution of the node set that `node_parallel_2` is part of. The outcome of the conditional branch is one or more activated nodes that will belong to the next set of activated nodes.

Execution of a sample graph

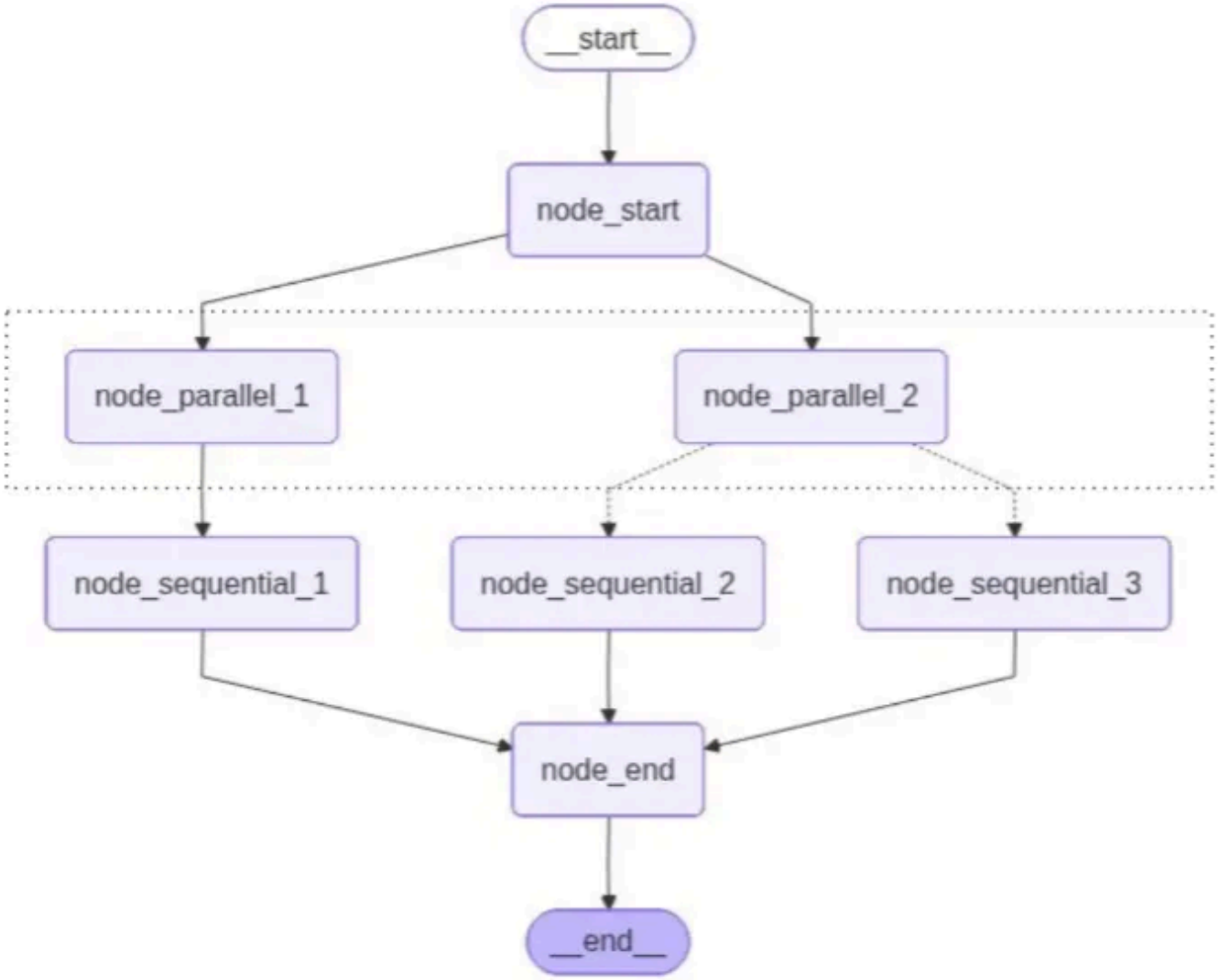
In the above example, node `node_start` is the start node and the first set of activated nodes (only one node is in the initial set — in principle more than one start node can exist in a graph).

In the following figures the dotted rectangle denotes the activated nodes.

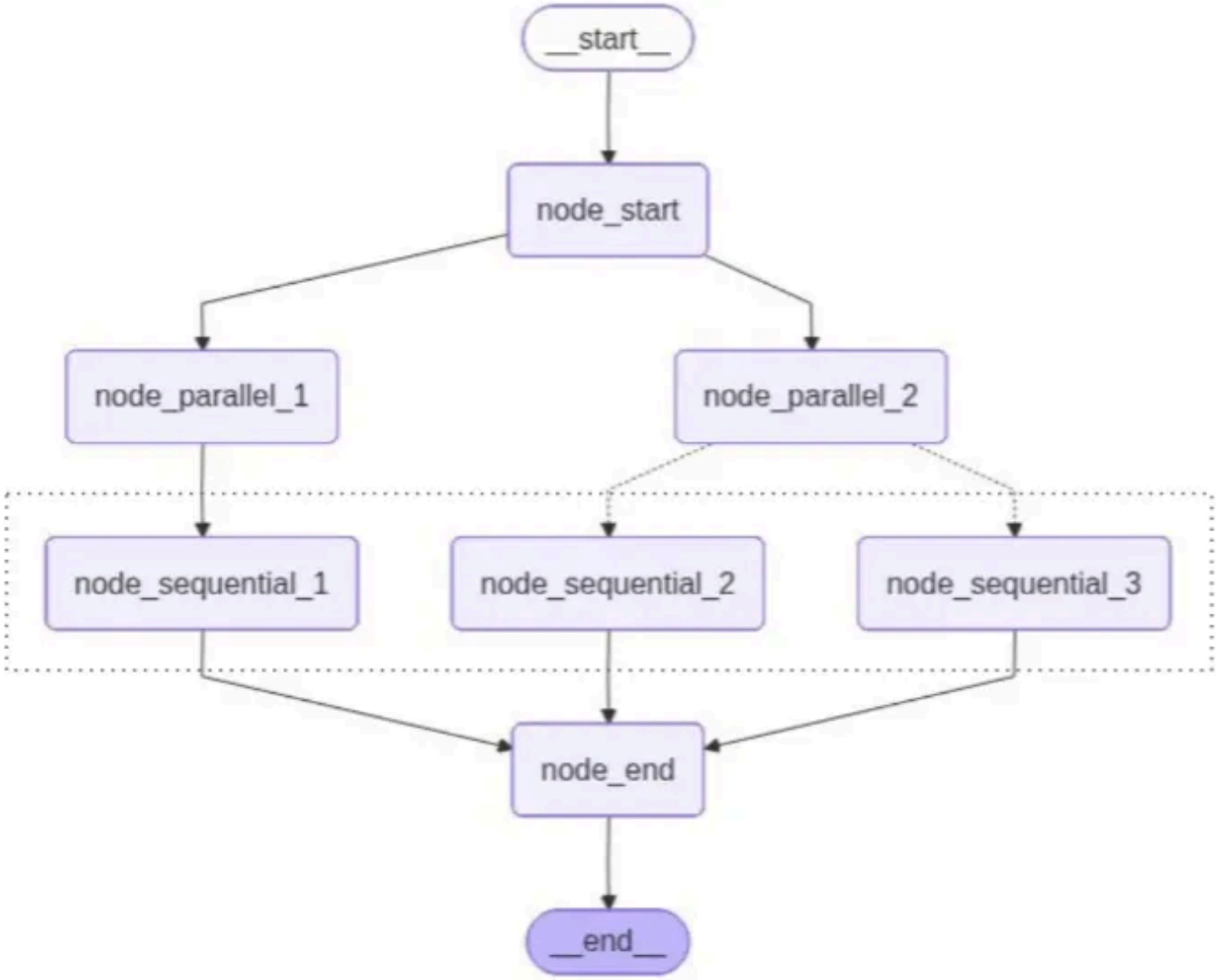


Once node `node_start` is completed, the next set of activated nodes is determined: this is node `node_parallel_1` and node `node_parallel_2`.

Node `node_parallel_1` and node `node_parallel_2` form the second set of activated nodes, and therefore node `node_parallel_1` and node `node_parallel_2` are executed.

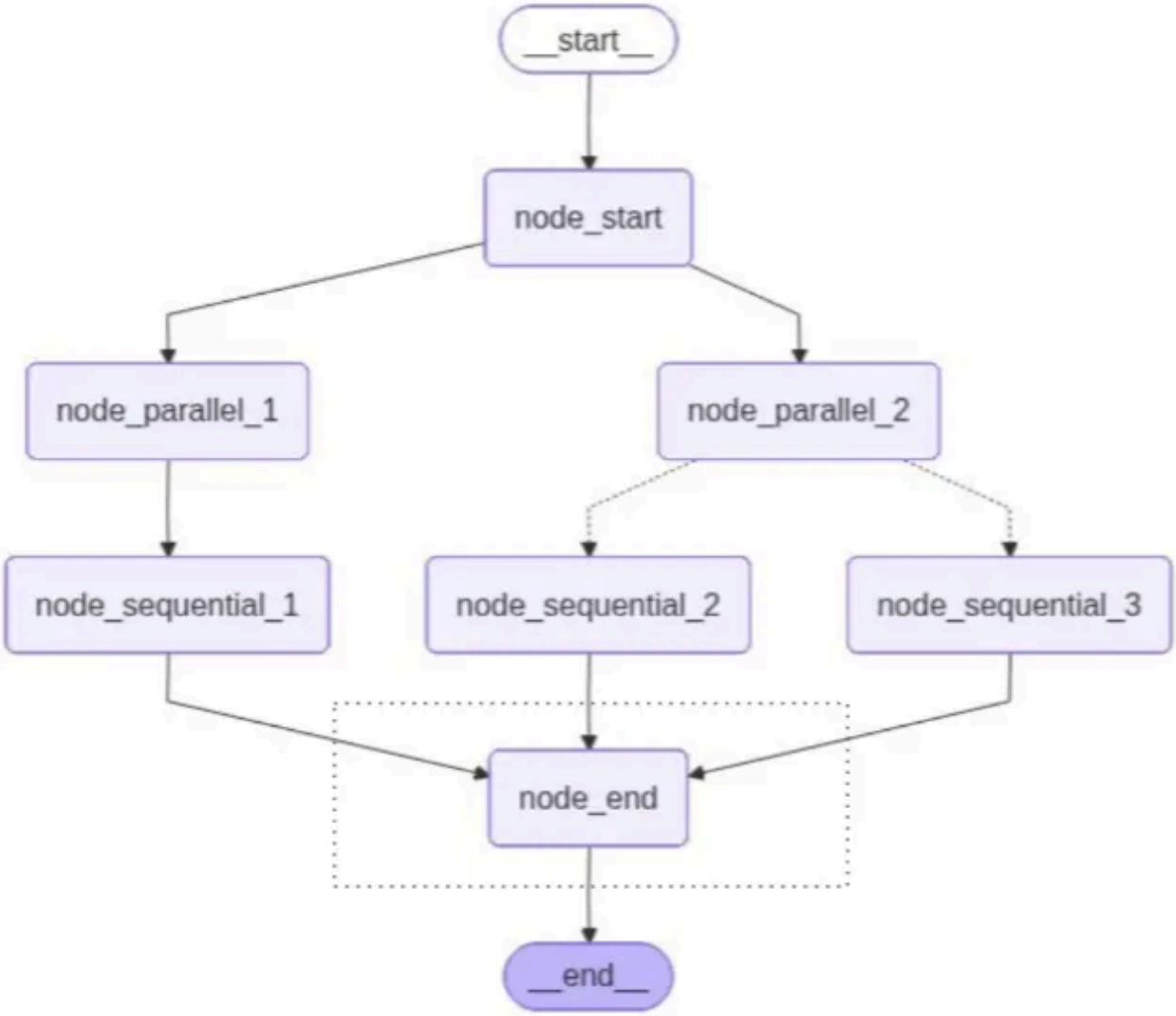


Only after node `node_parallel_1` and node `node_parallel_2` are completed, the next, third, set of activated nodes is determined: node `node_sequential_1` and either node `node_sequential_2` or `node_sequential_3`, depending on the outcome of the branching condition.



Once the third set is completed (aka, node `node_sequential_1` and either `node_sequential_2` or `node_sequential_3`), the last (fourth) set of activated nodes is determined and executed (only containing the node `node_end`).

Node `node_end` is the final node and after node `node_end`'s execution the execution of the entire graph is completed.

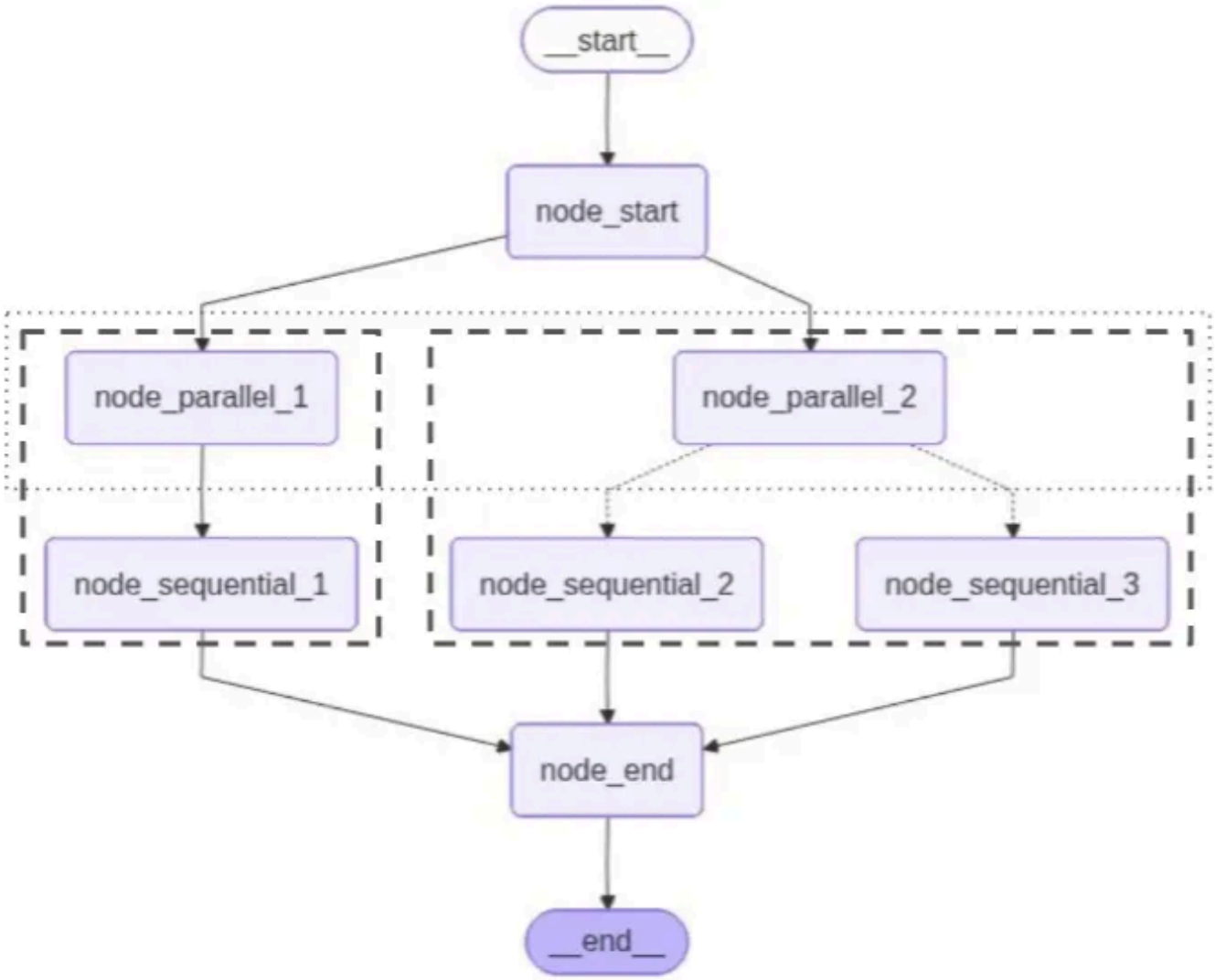


Observation: no branch parallelism during execution

Once the execution of node `node_parallel_1` is completed, node `node_sequential_1` is not immediately executed next as it is not in the same set of activated nodes as node `node_parallel_1`. Instead, the execution of node `node_sequential_1` only starts once the previous set of activated nodes that contains node `node_parallel_1` (its predecessor) has been completed.

Using the graphical representation from above, the dotted rectangle represents a set of activated nodes, and the dashed rectangles represent concurrent branches. All nodes within a dotted rectangle (set of activated nodes) are completely executed

before the next set of activated nodes starts executing even though nodes might be in concurrent branches.



As is visible graphically, even though in case of the two concurrent branches the nodes `node_sequential_1` and `node_sequential_2` or `node_sequential_3` could immediately start execution based on their dependencies alone, the LangGraph execution does not implement its execution semantics this way.

Formal LangGraph execution semantics based on Pregel

LangGraph execution (https://langchain-ai.github.io/langgraph/concepts/pregel/) is implemented based on the Pregel algorithm (https://research.google/pubs/pregel-a-system-for-large-scale-graph-processing/).

The Python classes functions implementing the LangGraph semantics can be found here: https://langchain-ai.github.io/langgraph/reference/pregel/.

The implemented semantics is straight forward and clear based on the Pregel algorithm. It does not restrict the expressiveness of graph structures (graph types). While not discussed in this blog, this also simplifies the data state coordination that would be more complex in parallel branch execution as different combinations of access would have to be guarded and to be coordinated in order to implement state correctness.

An article describing the semantics in more detail is this: https://medium.com/@maksymilian.pilzys/langgraph-transactions-pregel-message-passing-and-super-steps-0e101e620f10.

Parallelism of concurrent branches

Is it possible to have parallel execution of concurrent branches?

Nodes in a graph instance can call any function. A function could be calling another independent graph instance of a different graph type. Since this is independent of the node's graph, that independent graph is being executed separately. Two concurrent nodes can each call an independent graph, and that provides a higher degree of parallelism as each of the independent graphs is executed independently, aka, in parallel.

However, the complexity of state synchronization increases significantly if that has to take place across independent graph instances. In that case the execution is parallel as the execution of the node implementation creates an independent instance. This changes the model to a federation where each of the graph instances has its own state and no consistency across the various involved graphs by the LangGraph execution semantics. Any data coordination and consistency must then be implemented independent of the graph instances, and independent of the assurances of the LangGraph implementation.

A trade off discussion between the benefit of parallel execution of independent graph instances and the added complexity of state management has to carefully take place — at least I would suggest that strongly.

Summary

While LangGraph supports graphs with concurrent branches of nodes, the execution of nodes in concurrent branches is not independent of each other, aka, parallel. Instead, LangGraph's execution semantics is based on the graph-global execution of activated node sets following the Pregel semantics, one node set after the next node set, which does not take concurrent branches into consideration.

Appendix: code documentation

The following shows the code documentation from the class Pregel in this GitHub location: https://github.com/langchain-ai/langgraph/blob/main/libs/langgraph/langgraph/pregel/main.py.

```
Appendix: code documentation
The following shows the code documentation from the class Pregel in this GitHub


class Pregel(
```

```
    PregelProtocol[StateT, ContextT, InputT, OutputT],
    Generic[StateT, ContextT, InputT, OutputT],
):
    """
    Pregel manages the runtime behavior for LangGraph applications.


    ## Overview


    Pregel combines [**actors**](https://en.wikipedia.org/wiki/Actor_model)
    and **channels** into a single application.
    **Actors** read data from channels and write data to channels.
    Pregel organizes the execution of the application into multiple steps,
    following the **Pregel Algorithm**/**Bulk Synchronous Parallel** model.


    Each step consists of three phases:


    - **Plan**: Determine which **actors** to execute in this step. For example,
        in the first step, select the **actors** that subscribe to the special
        **input** channels; in subsequent steps,
        select the **actors** that subscribe to channels updated in the previous
    - **Execution**: Execute all selected **actors** in parallel,
        until all complete, or one fails, or a timeout is reached. During this
        phase, channel updates are invisible to actors until the next step.
    - **Update**: Update the channels with the values written by the **actors**
        in this step.


    Repeat until no **actors** are selected for execution, or a maximum number of
    steps is reached.


    ## Actors


    An **actor** is a `PregelNode`.
    It subscribes to channels, reads data from them, and writes data to them.
    It can be thought of as an **actor** in the Pregel algorithm.
    `PregelNodes` implement LangChain's
    Runnable interface.


    ## Channels


    Channels are used to communicate between actors (`PregelNodes`).
    Each channel has a value type, an update type, and an update function – which
```

```
takes a sequence of updates and
modifies the stored value. Channels can be used to send data from one chain t
another, or to send data from a chain to itself in a future step. LangGraph
provides a number of built-in channels:


### Basic channels: LastValue and Topic


- `LastValue`: The default channel, stores the last value sent to the channe
   useful for input and output values, or for sending data from one step to t
- `Topic`: A configurable PubSub Topic, useful for sending multiple values
   between *actors*, or for accumulating output. Can be configured to dedupli
   values, and/or to accumulate values over the course of multiple steps.


### Advanced channels: Context and BinaryOperatorAggregate


- `Context`: exposes the value of a context manager, managing its lifecycle.
  Useful for accessing external resources that require setup and/or teardown.
  `client = Context(httpx.Client)`
- `BinaryOperatorAggregate`: stores a persistent value, updated by applying
   a binary operator to the current value and each update
   sent to the channel, useful for computing aggregates over multiple steps.
  `total = BinaryOperatorAggregate(int, operator.add)`


## Examples


Most users will interact with Pregel via a
[StateGraph (Graph API)][langgraph.graph.StateGraph] or via an
[entrypoint (Functional API)][langgraph.func.entrypoint].


However, for **advanced** use cases, Pregel can be used directly. If you're
not sure whether you need to use Pregel directly, then the answer is probably
– you should use the Graph API or Functional API instead. These are higher-le
interfaces that will compile down to Pregel under the hood.


Here are some examples to give you a sense of how it works:


Example: Single node application
    ```python
    from langgraph.channels import EphemeralValue
    from langgraph.pregel import Pregel, NodeBuilder
```
```

```
    node1 = (
        NodeBuilder().subscribe_only("a")
        .do(lambda x: x + x)
        .write_to("b")
    )


    app = Pregel(
        nodes={"node1": node1},
        channels={
            "a": EphemeralValue(str),
            "b": EphemeralValue(str),
        },
        input_channels=["a"],
        output_channels=["b"],
    )


    app.invoke({"a": "foo"})
```

```
    ```con
    {'b': 'foofoo'}
```

Example: Using multiple nodes and multiple output channels

```
    ```python
    from langgraph.channels import LastValue, EphemeralValue
    from langgraph.pregel import Pregel, NodeBuilder


    node1 = (
        NodeBuilder().subscribe_only("a")
        .do(lambda x: x + x)
        .write_to("b")
    )


    node2 = (
        NodeBuilder().subscribe_to("b")
        .do(lambda x: x["b"] + x["b"])
        .write_to("c")
    )
```

```
app = Pregel(
    nodes={"node1": node1, "node2": node2},
    channels={
        "a": EphemeralValue(str),
        "b": LastValue(str),
        "c": EphemeralValue(str),
    },
    input_channels=["a"],
    output_channels=["b", "c"],
)


app.invoke({"a": "foo"})
```


```con
{'b': 'foofoo', 'c': 'foofoofoofoo'}
```


Example: Using a Topic channel

```python
from langgraph.channels import LastValue, EphemeralValue, Topic
from langgraph.pregel import Pregel, NodeBuilder


node1 = (
    NodeBuilder().subscribe_only("a")
    .do(lambda x: x + x)
    .write_to("b", "c")
)


node2 = (
    NodeBuilder().subscribe_only("b")
    .do(lambda x: x + x)
    .write_to("c")
)




app = Pregel(
    nodes={"node1": node1, "node2": node2},
    channels={
        "a": EphemeralValue(str),
        "b": EphemeralValue(str),
        "c": Topic(str, accumulate=True),
    },
```

````
        input_channels=["a"],
        output_channels=["c"],
    )


    app.invoke({"a": "foo"})
    ```


    ```pycon
    {'c': ['foofoo', 'foofoofoofoo']}
    ```


Example: Using a BinaryOperatorAggregate channel
    ```python
    from langgraph.channels import EphemeralValue, BinaryOperatorAggregate
    from langgraph.pregel import Pregel, NodeBuilder




    node1 = (
        NodeBuilder().subscribe_only("a")
        .do(lambda x: x + x)
        .write_to("b", "c")
    )


    node2 = (
        NodeBuilder().subscribe_only("b")
        .do(lambda x: x + x)
        .write_to("c")
    )




    def reducer(current, update):
        if current:
            return current + " | " + update
        else:
            return update


    app = Pregel(
        nodes={"node1": node1, "node2": node2},
        channels={
            "a": EphemeralValue(str),
            "b": EphemeralValue(str),
````

```
            "c": BinaryOperatorAggregate(str, operator=reducer),
        },
        input_channels=["a"],
        output_channels=["c"]
    )


    app.invoke({"a": "foo"})
    ```


    ```con
    {'c': 'foofoo | foofoofoofoo'}
    ```


Example: Introducing a cycle
    This example demonstrates how to introduce a cycle in the graph, by havir
    a chain write to a channel it subscribes to. Execution will continue
    until a None value is written to the channel.


    ```python
    from langgraph.channels import EphemeralValue
    from langgraph.pregel import Pregel, NodeBuilder, ChannelWriteEntry


    example_node = (
        NodeBuilder().subscribe_only("value")
        .do(lambda x: x + x if len(x) < 10 else None)
        .write_to(ChannelWriteEntry(channel="value", skip_none=True))
    )


    app = Pregel(
        nodes={"example_node": example_node},
        channels={
            "value": EphemeralValue(str),
        },
        input_channels=["value"],
        output_channels=["value"]
    )


    app.invoke({"value": "a"})
    ```


    ```con
    {'value': 'aaaaaaaaaaaaaaaa'}
```

```
    """
```

Langgraph | Execution Semantics | Parallelism | Concurrency | Process

Written by Christoph Bussler

260 followers · 36 following

www.real-programmer.com

Edit profile