How to run the project :

On Linux :
sudo apt install graphviz
cargo run

I couldn’t test on Windows, but graphviz has been ported for Windows, so it should work if installed.

main.rs contains an example of both way to define an `*ExecutableGraph*`
lib.rs contains the implementation of the thread pool
executable_graph.rs contains the implementation of the `*ExecutableGraph*`

First a `*ThreadPool*` instance is necessary. We get one with the `*ThreadPool::new*` method, specifying how many threads we want.
Then there are two ways to generate an `*ExecutableGraph*`, as shown in *main.rs*

- using `*ExecutableGraph::new*` method, passing it an instance of `*ThreadPool*`
  - then calling `*add_initial_node*` and `*add_node*` methods to iteratively construct the graph
- using `*ExecutableGraph::generate_random*`, passing it
  - the number of nodes it will generate
  - the max number of parents a node can have
  - a list of arguments, which defines how many initial nodes are created and what to feed them with
  - a list of `*Operation*`, containing a closure and a name
  - an optional probability distribution for the operations
  - an instance of `*ThreadPool*`

Then using the `*start*` method to run the graph, and `*show_graph*` to print a representation of it.
If too many nodes are used, it’s probably wise not to use the `*show_graph*` method.

Report

I am a beginner with Rust, but I had read the online book, and it was the occasion to practise.

Comprehension of the problematic

- at first, I thought of the parallelizability only in term of graph dependencies
  - every node that have the same parents are parallelizable
  - I thought I would create bundles of parallelizable nodes as clusters, and work with clusters instead of nodes and treat clusters sequentially

- then I realized the time factor matters A LOT
  - if one node inside a cluster took too much time, everything would wait for it

- in the end, I found a simpler way to tell which nodes to execute
  - those whose parents are done with their task
    - no need to test every nodes, just keeping track of which are possible candidates
  - once a candidate is done, track all its children instead
    - also update the edges with the result

- therefore, I needed a way to tell which nodes were finished
  - I used messages for the graph and the threads to communicate
    - each thread handles one node operation, and sends a message with the result and the identity of the node back to the graph

Implementation

- since Rust doesn't support functions with arbitrary number of arguments, I used a vector

- I used to implement the Iterator trait to tell when there is no more node to execute, but traits are public, and it hadn't the behavior the user would expect
  - created `` `*is_done*` `` method instead

- tried to implement the graph itself with `` `*Node*` `` structure that hold references to other `` `*Node*` `` instances
  - big fight with the borrow checker
  - looked the implementation of petgraph to have an idea how they do it. They use indices.
  - ended up using petgraph to represent the graph

- `` `*current_nodes*` `` and `` `*finished_nodes*` `` used to be vectors
  - had to find elements inside it efficiently
  - ended up turning them into Hashmaps, as they can become really big, especially `` `*finished_nodes*` ``, who end up containing every nodes of the graph. Looking into the vector is O(n), and O(1) with the hashmap
  - used to filter a lot the vectors to avoid adding nodes that were already there, used a hashset instead of hashmap
- Petgraph can generate .dot format files from the graph
  - but only if the types of the nodes and edges implement the `` `*Display*` `` trait, and the type `` `*Option*` `` doesn't
  - reconstruct an equivalent graph with String type for nodes and edges
    I guess I could have reimplemented my own `` `*Option*` `` type and implement the Display type on it instead

Graph generation and execution

- the graph can only compute functions that have the same output type than the input
  $f: X \times X \ldots \times X \rightarrow X$ No verification is done on it. It can panic, for instance if the user defines a function that overflows an integer, or divide by zero.

- handle the acyclic constraint of the graph
  - first thought about trying to find circuits inside the graph everytime it's changed
  - then I realized that if instead of letting the user set the children of the nodes, we let him set the (already existing) parents, we can't end up with a cyclic graph

- how to tell which nodes are being executed in parallel ?
  - I know when I request the thread pool, but not exactly when the node is going to be executed
  - I could be notified when the thread starts its operation, via messaging
  - so I can know at an instant **t** which ones are being executed in parallel, by maintaining a list, but when do I print this list ? Every second ? I might be missing some if the operation is too fast

    - in the end I simply print it everytime the main thread receive a message

- looked into openCL to try to execute on the GPU
    - but how do I pass an arbitrary rust function to the GPU ?
    - Apparently would require langage translation
      how does TensorFlow does it ? Are all possible functions already predefined ?

Definition of initial nodes

- initially thought I would look for every nodes which don't have parents

- since every node is an operation, they all have to have inputs, but inputs are defined by edges, then how do I define nodes that have no parent, like the entry points ?

- Thought they could be special type of nodes that would apply the identity function, but it's a waste to treat those as nodes to compute
    - ended up definying a special type of node called `initial_node`, that would require arguments and that would be executed with those instead of the data defined in edges
    - other nodes require parents instead, they can't be orphean
        - therefore, the number of possible connected subgraphs is equal to the number of `initial nodes`