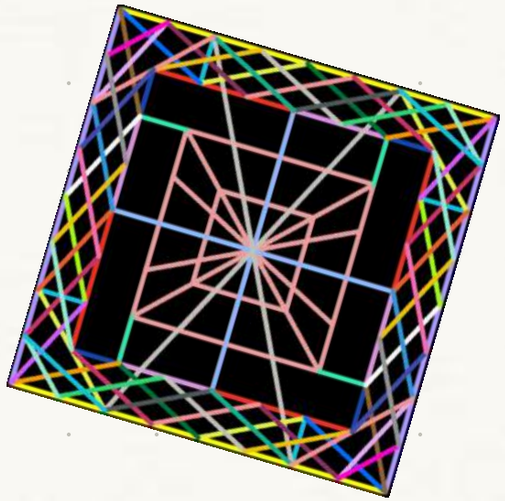
# Understanding NetworkX's API Dispatching with a parallel backend

-By Aditi Juneja
(Mostly goes by
Schefflera Arboricola)



# Understanding NetworkX's API Dispatching with a parallel backend

# NetworkX Backend Dispatching

- Plugin Arch.
- entry_points
- Automatic backend testing

can_run
should_run
on_start_tests
caching
logging
backend priority

config

## NetworkX

*A pure Python library for network analysis*

backends

Backend Interface

backend_info

get_info

## nx-parallel

*A parallel backend for NetworkX, utilizing Joblib to implement graph algorithms on multiple CPU cores.*

too many parallel processes and no GIL

To chunk or not to chunk?

## Usage:

```
>>> import nx_parallel as nxp
>>> nx.betweenness_centrality(G,
backend="parallel")
>>> H = nxp.ParallelGraph(G)
>>> nx.betweenness_centrality(H)

$ export NETWORKX_AUTOMATIC_BACKEND=parallel
  && python nx_code.py
```

## Speed-ups

local_efficiency-**8.8x**
betweenness_centrality-**6.4x**
square_clustering-**4.1x**
bipartite.node_redundancy-**4.1x**

*for 300-node random graphs (edge probability = 0.6)

For testing:

```
NETWORKX_TEST_BACKEND=parallel
NETWORKX_FALLBACK_TO_NX=True
pytest --pyargs networkx "$@"
```

feel free to contribute and nurture!

TO-DO

- Designing the pipeline nicely
- Benchmarking
- Adding distributed graph algorithms

joblib

loky

threading

multiprocessing

dask, ray...

9th July, 2024

Aditi Juneja (@Schefflera-Arboricola)

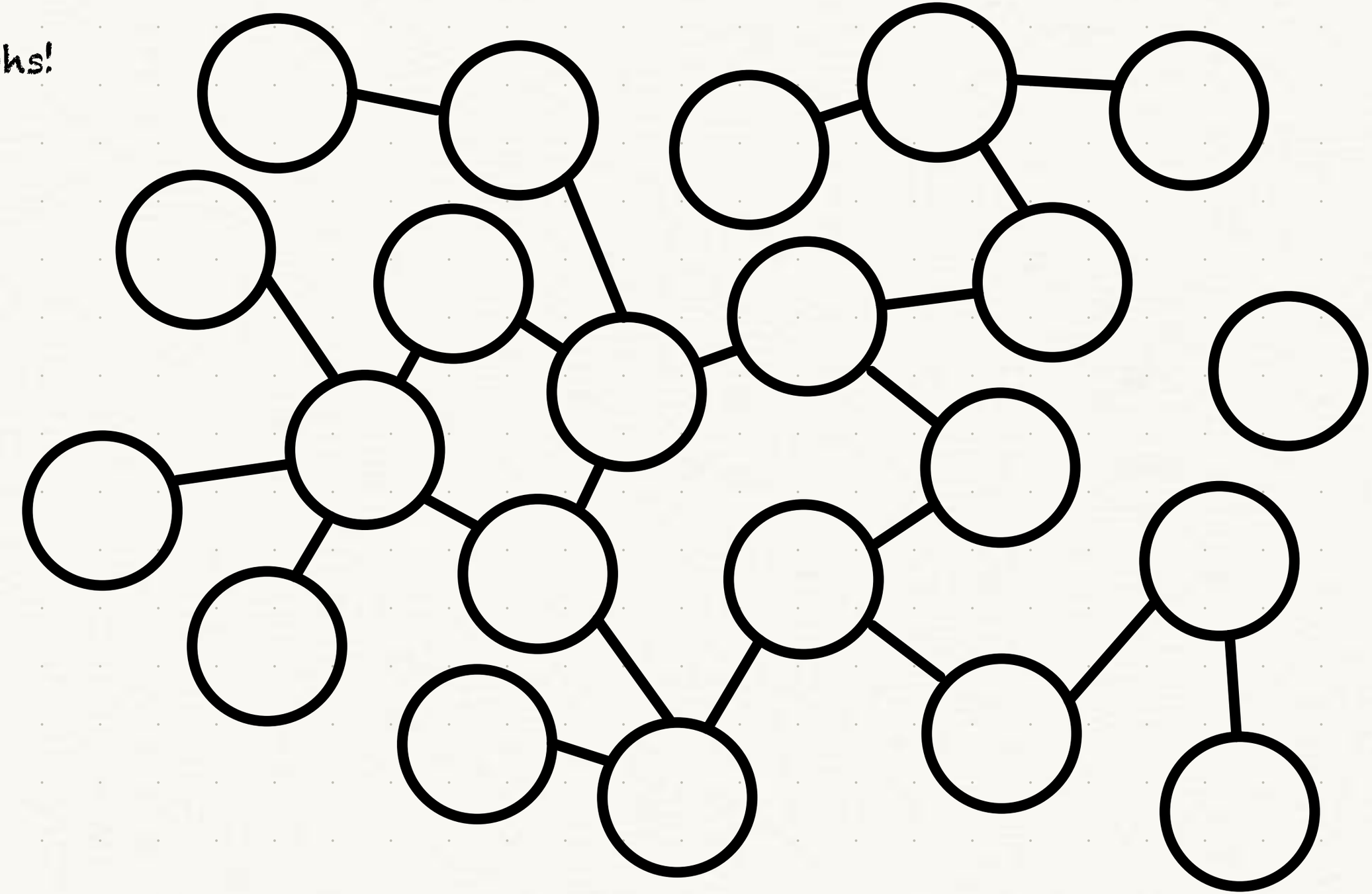
# Understanding NetworkX's API Dispatching with a parallel backend

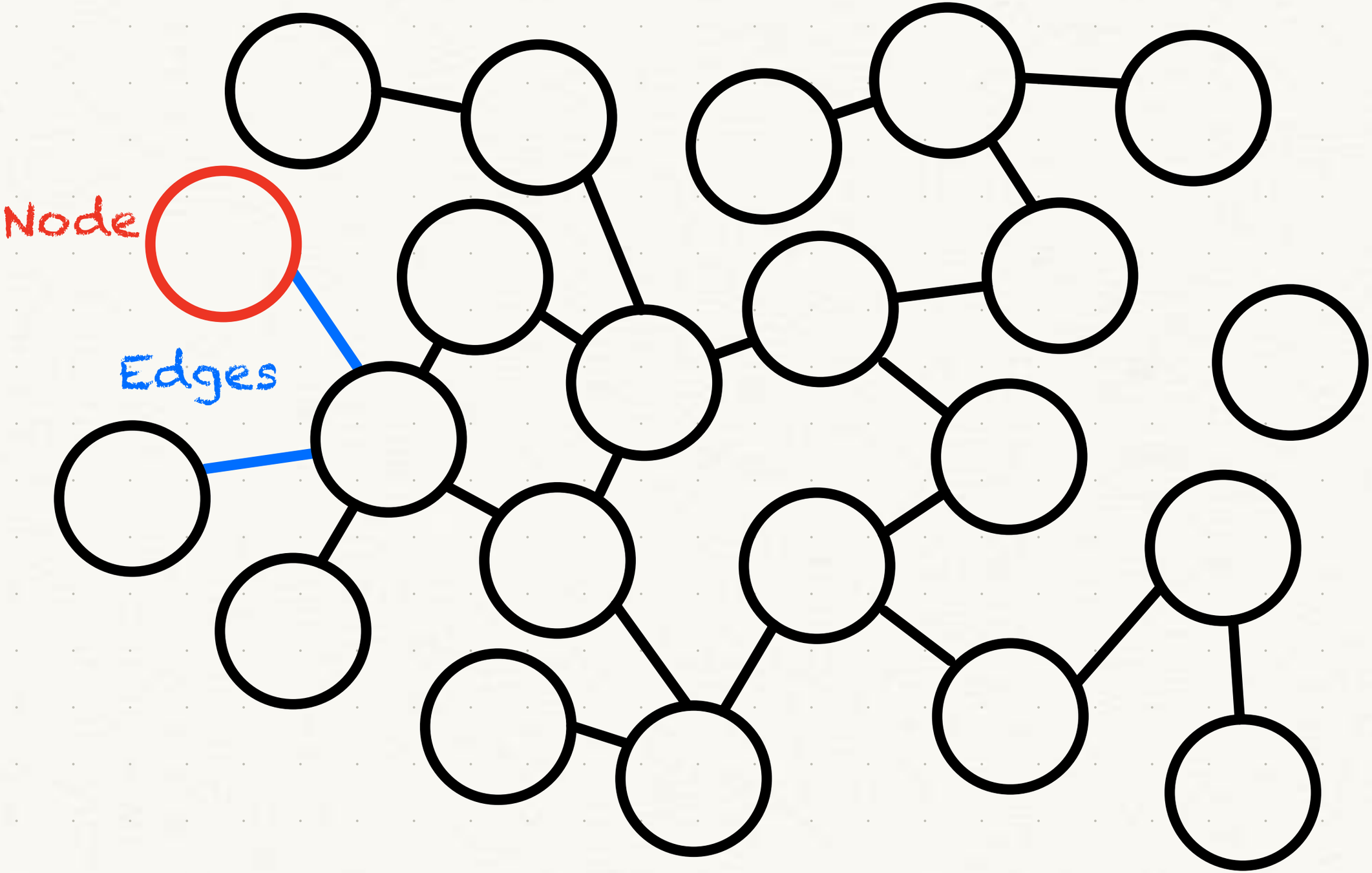
# Understanding NetworkX's API Dispatching with a parallel backend

A graph (aka network) analysis Python library

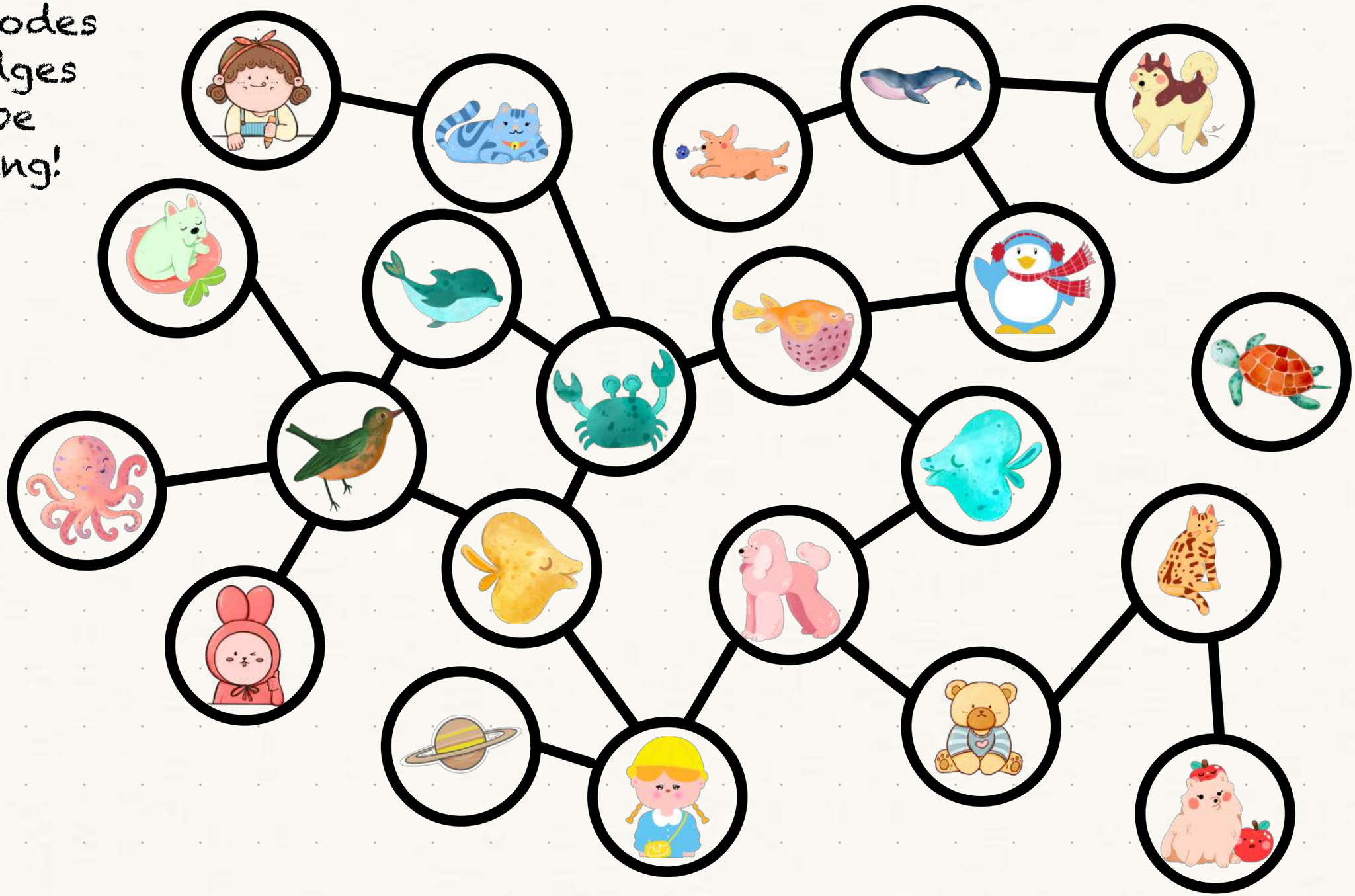
**NOT** these kind of graphs

# Graphs!



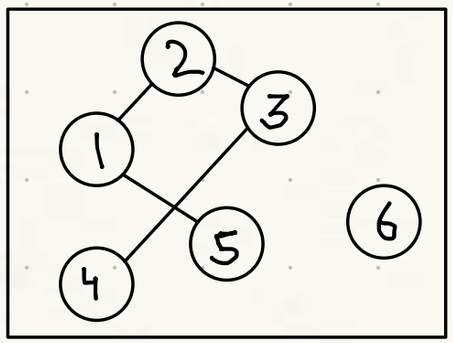

Node
Edges

These nodes and edges can be anything!



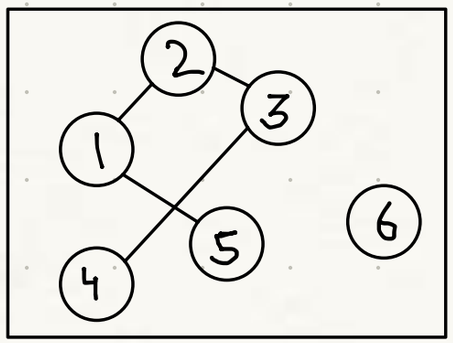
# NetworkX API

```
>>> import networkx as nx

>>> G = nx.Graph()

>>> G.add_edges_from([(1,2), (3,4), (2,3),
(5,1)])

>>> G.add_node(6)

>>> nx.betweenness_centrality(G)
```



# NetworkX API

```
>>> import networkx as nx

>>> G = nx.Graph()

>>> G.add_edges_from([(1,2), (3,4), (2,3),
(5,1)])

>>> G.add_node(6)

>>> nx.betweenness_centrality(G)

{1: 0.30000000000000004, 2: 0.4, 3:
0.30000000000000004, 4: 0.0, 5: 0.0, 6: 0.0}
```



# NetworkX API

```
>>> import networkx as nx

>>> G = nx.Graph()

>>> G.add_edges_from([(1,2), (3,4), (2,3),
(5,1)])

>>> G.add_node(6)

>>> nx.betweenness_centrality(G)

>>> G = nx.fast_gnp_random_graph(1000000, 0.5)

>>> nx.betweenness_centrality(G)
```

Now, lets try this...

# NetworkX API

```
>>> import networkx as nx

>>> G = nx.Graph()

>>> G.add_edges_from([(1,2), (3,4), (2,3),
(5,1)])

>>> G.add_node(6)

>>> nx.betweenness_centrality(G)

>>> G = nx.fast_gnp_random_graph(1000000, 0.5)

>>> nx.betweenness_centrality(G)

...takes forever 😫😫😫😭😭
```

# NetworkX API

```
>>> import networkx as nx

>>> G = nx.Graph()

>>> G.add_edges_from([(1,2), (3,4), (2,3),
(5,1)])

>>> G.add_node(6)

>>> nx.betweenness_centrality(G)

>>> G = nx.fast_gnp_random_graph(1000000, 0.5)

>>> nx.betweenness_centrality(G)
```

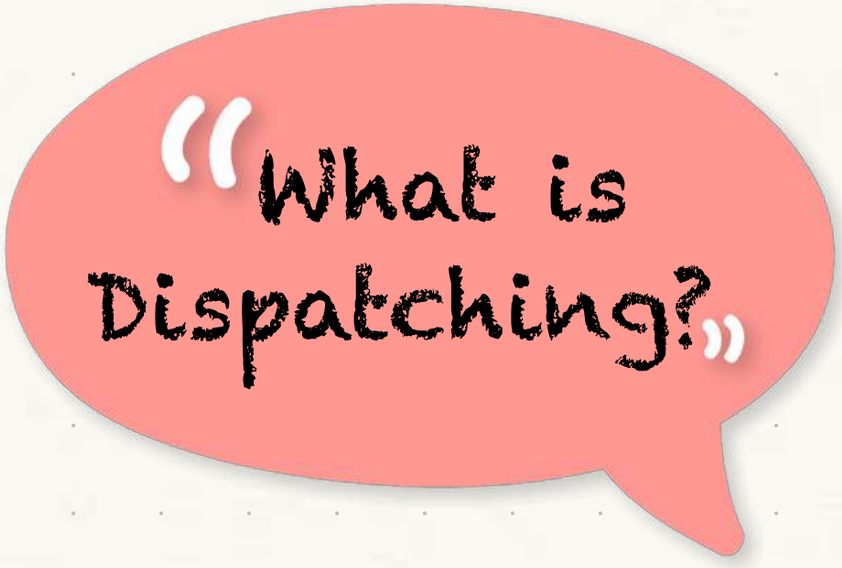
...takes forever 😫😫😫😭😭

What to do now?

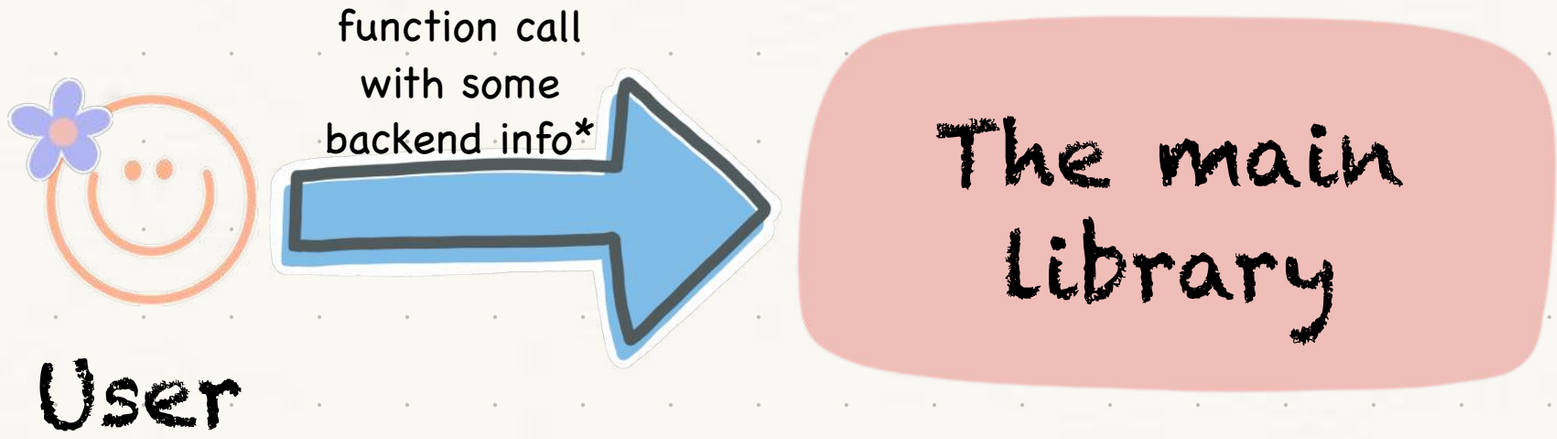
# Understanding NetworkX's API Dispatching with a parallel backend

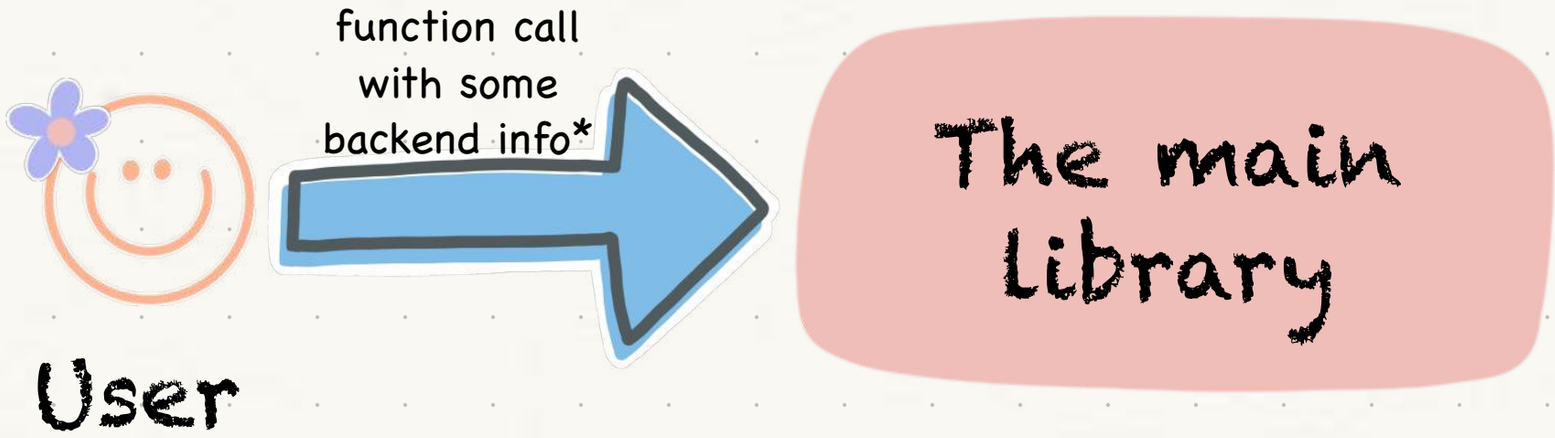
# Understanding NetworkX's API Dispatching with a parallel backend

# Understanding NetworkX's API Dispatching with a parallel backend



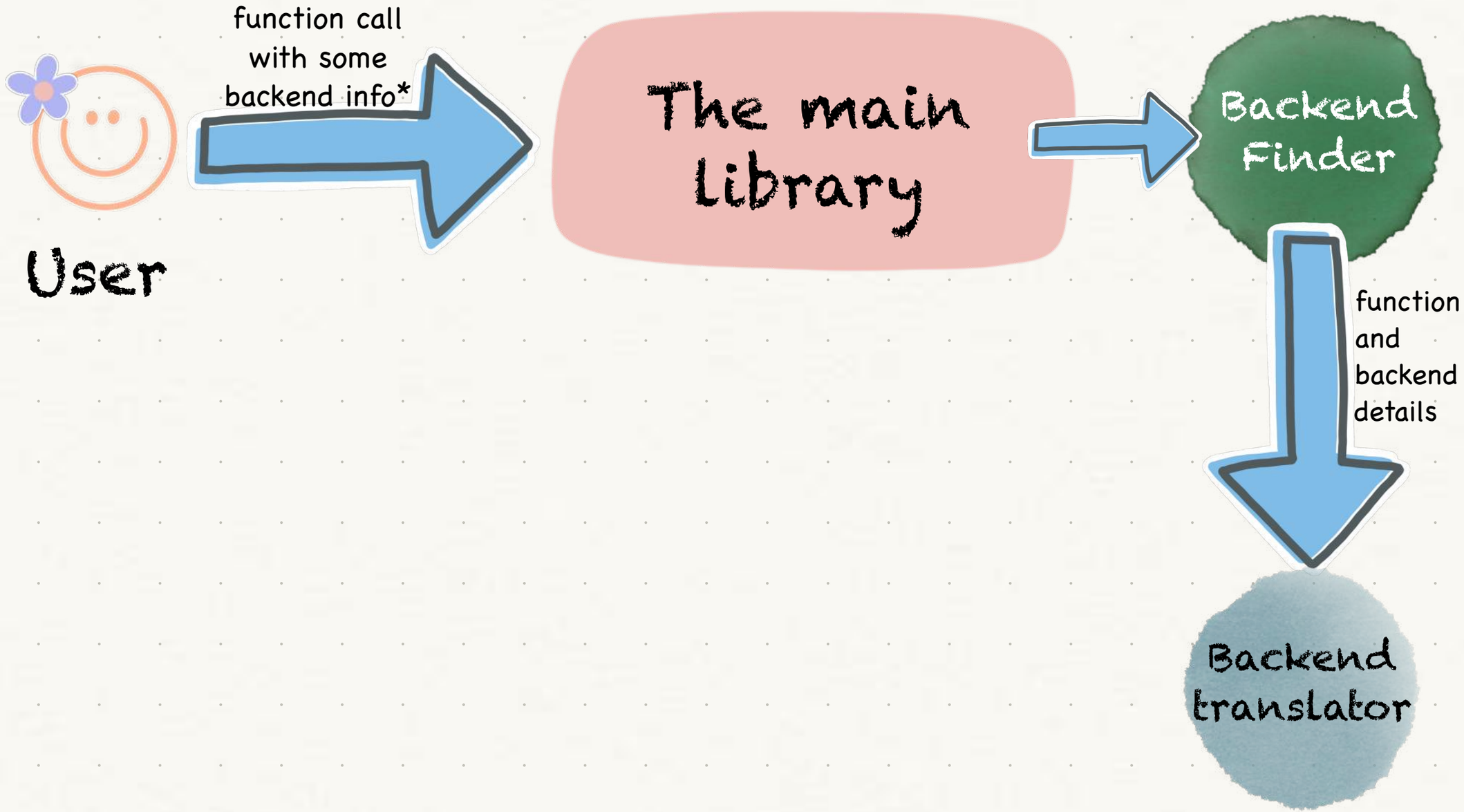
User

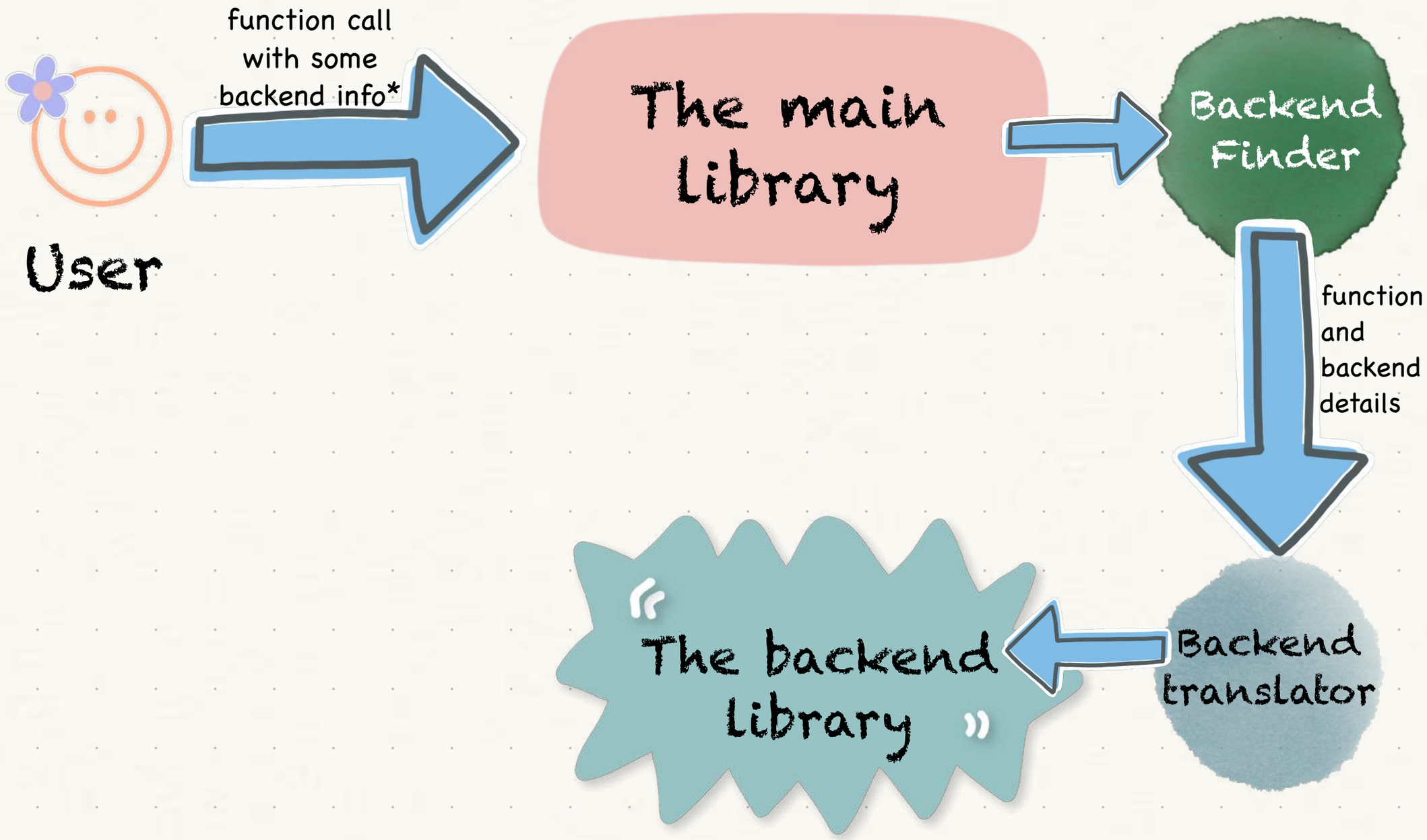

function call
with some
backend info*
The main
library
User

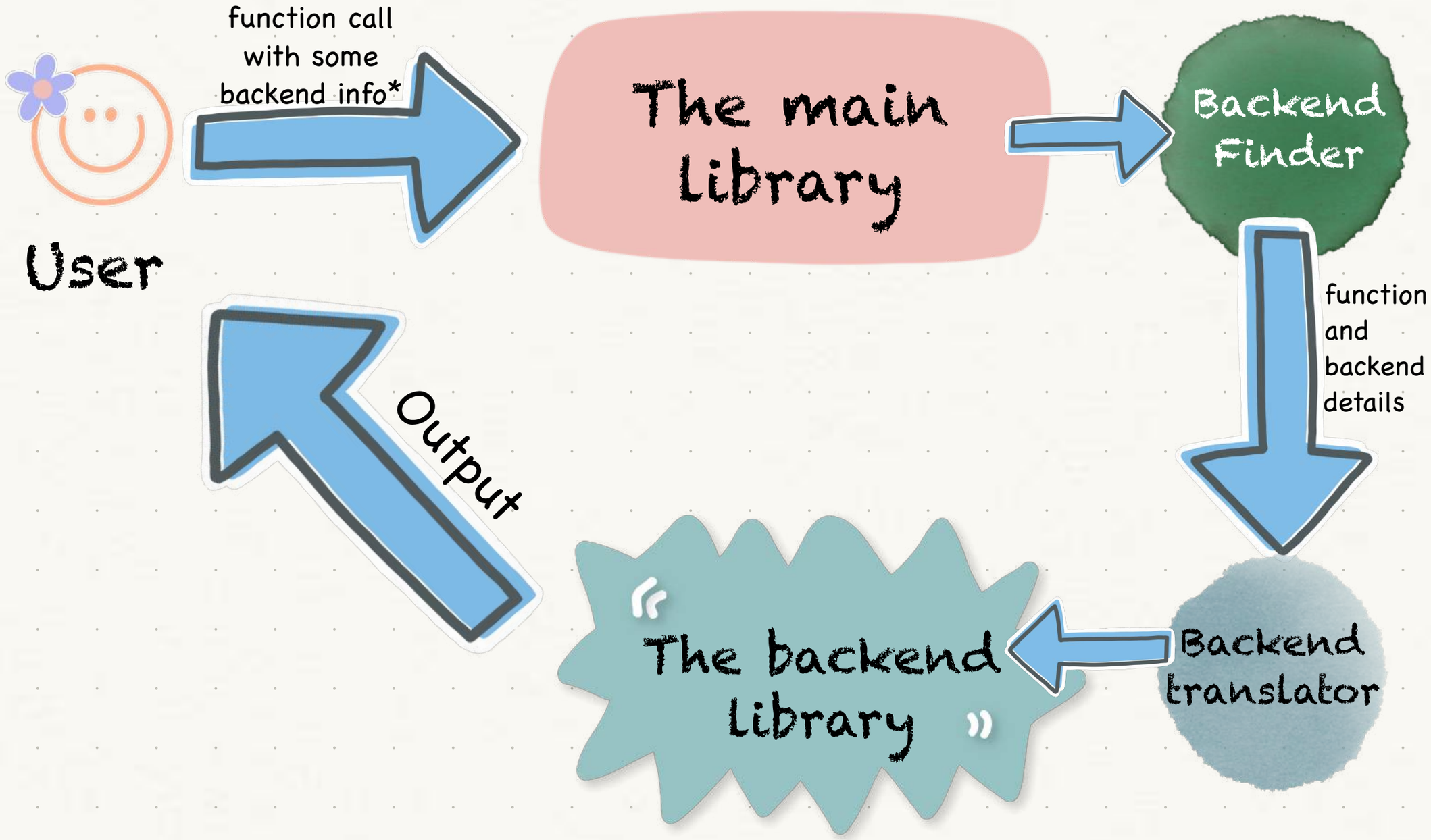


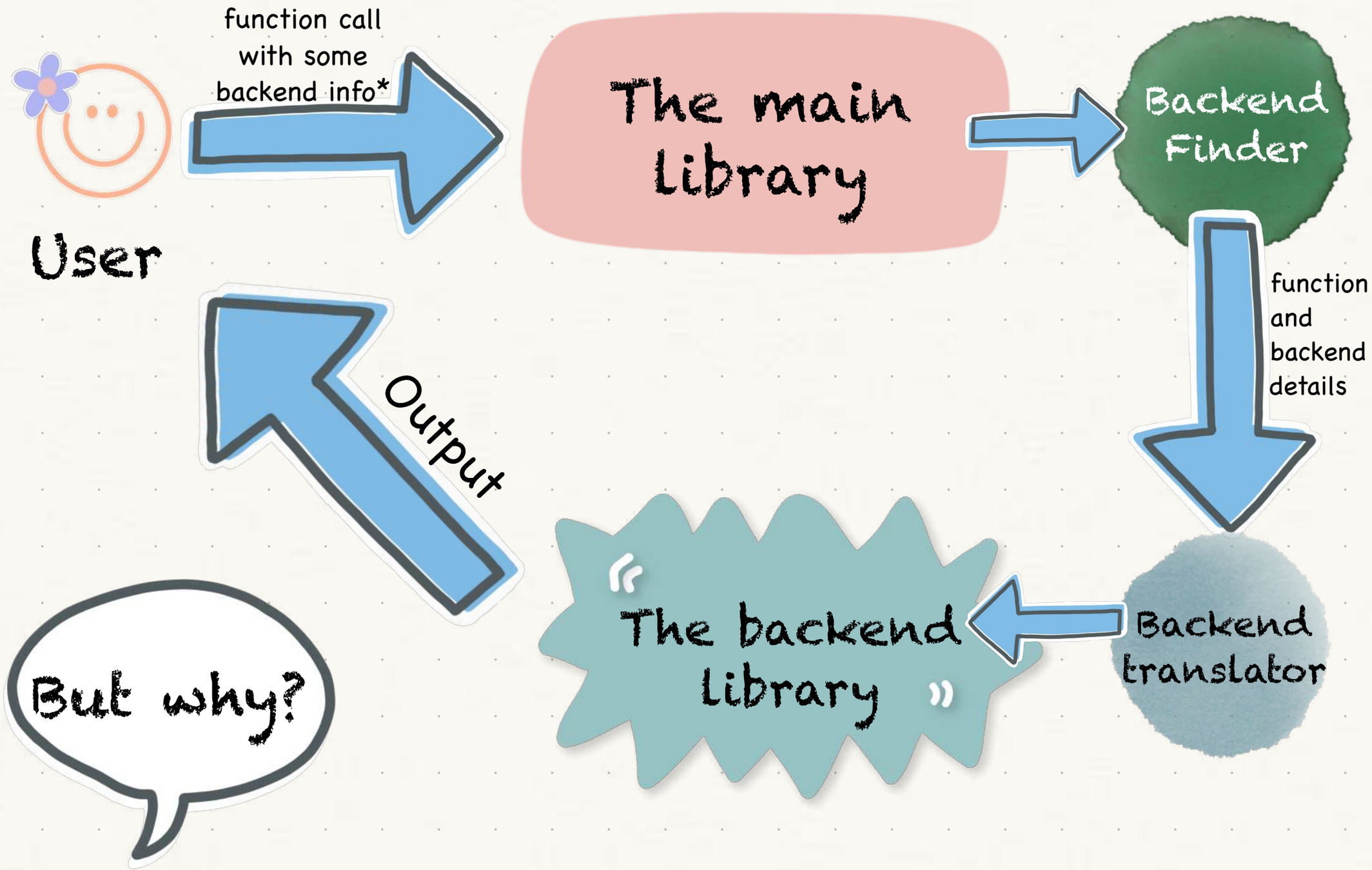

*This backend information might not be entered in the function call itself but can also be globally stored, like as an environment variable.


User
function call
with some
backend info*
The main
Library
Backend
Finder


User
function call
with some
backend info*
The main
library
Backend
Finder
function
and
backend
details
Backend
translator


User
function call
with some
backend info*
The main
library
Backend
Finder
function
and
backend
details
Backend
translator
« The backend
library »


User
function call
with some
backend info*
The main
library
Backend
Finder
function
and
backend
details
Backend
translator
« The backend
library »
Output

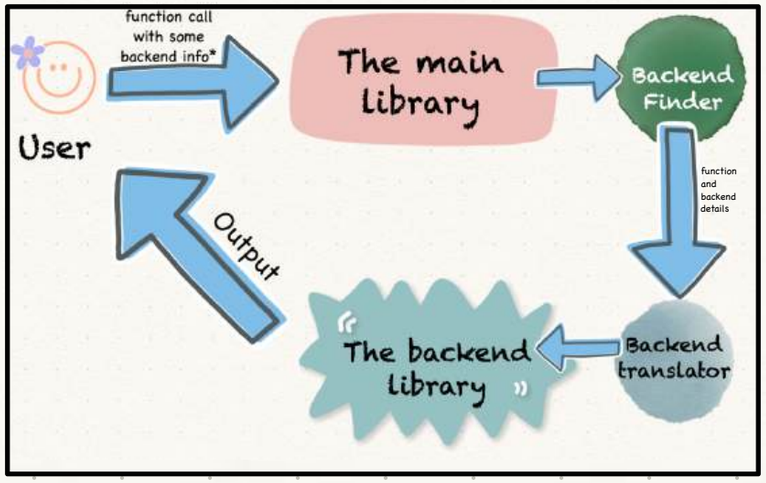

User
function call with some backend info*
The main library
Backend Finder
function and backend details
Backend translator
« The backend library »
Output
But why?

# But why dispatching?

- Consistency!!
- NetworkX is a big and old project.
- The API remains similar for the end user, except they just have to pass some additional backend information.
- Is API dispatching the best thing for your package? - It depends...

Lets see API dispatching in NetworkX

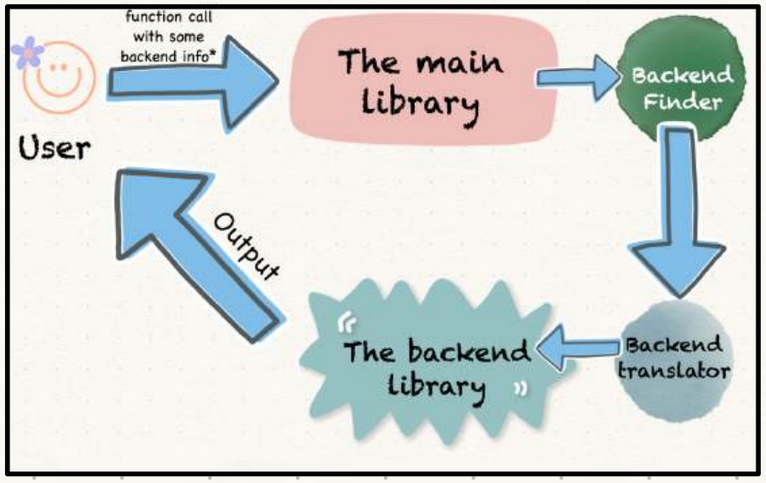
# Function call with "some backend information" in NetworkX




1. `backend` kwarg in the function call:

```
nx.betweenness_centrality(G, backend="parallel")
```

# Function call with "some backend information" in NetworkX



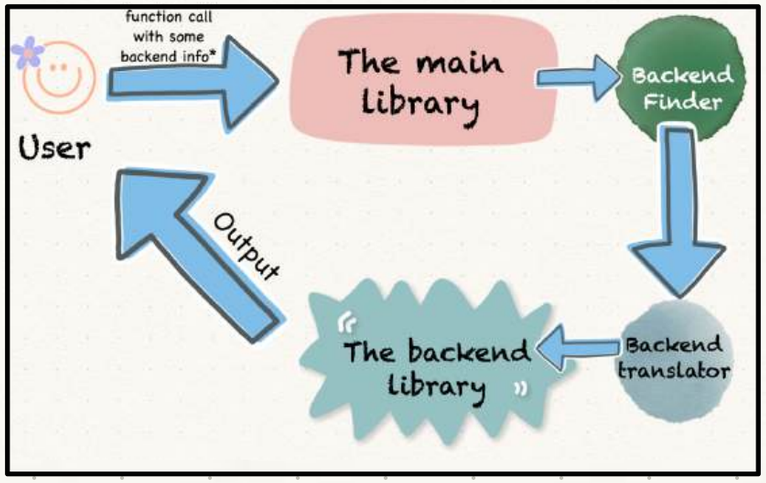

1. `backend` kwarg in the function call:

```
nx.betweenness_centrality(G, backend="parallel")
```

2. Type-based dispatching

```
import nx_parallel as nxp
H = nxp.ParallelGraph(nx.complete_graph(3))
nx.betweenness_centrality(H)
```

# Function call with "some backend information" in NetworkX




1. `backend` kwarg in the function call:

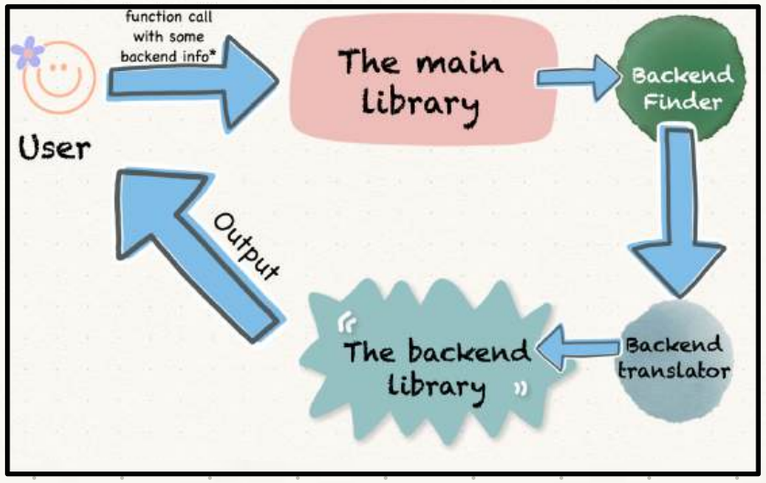
```
nx.betweenness_centrality(G, backend="parallel")
```

2. Type-based dispatching

```
import nx_parallel as nxp
H = nxp.ParallelGraph(nx.complete_graph(3))
nx.betweenness_centrality(H)
```

```
class ParallelGraph:
    __networkx_backend__ = "parallel"

    def __init__(self, ...):
        ....
```

# Function call with "some backend information" in NetworkX




1. `backend` kwarg in the function call:

```
nx.betweenness_centrality(G, backend="parallel")
```

2. Type-based dispatching

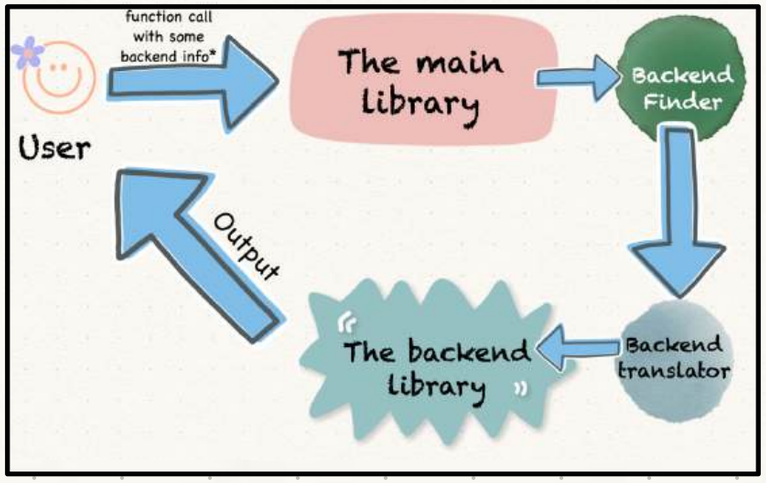
```
nx.betweenness_centrality(H)
```

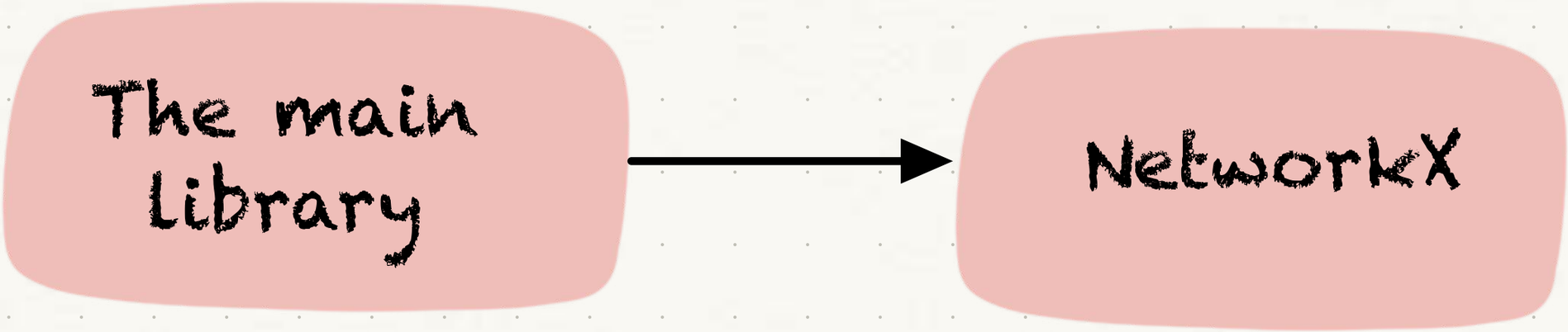
3. Using networkx's configurations

```
with nx.config(backend_priority=['parallel',]):
    nx.square_clustering(G)
```

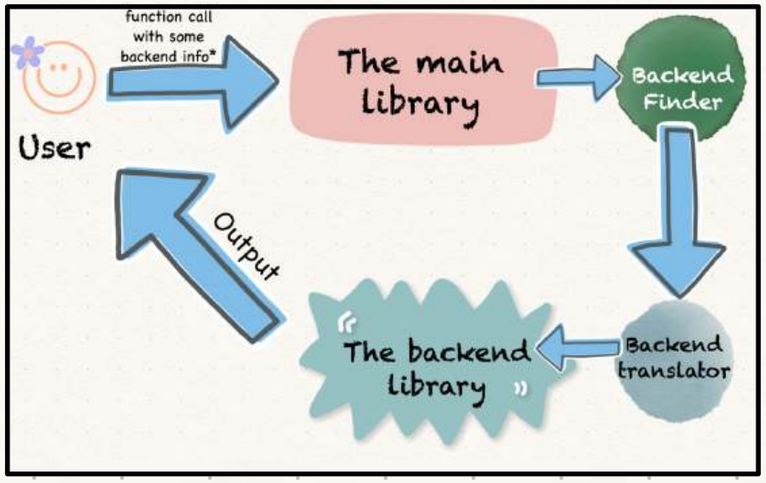
`@nx._dispatchable` decorator's job:

To identify "some backend info" and accordingly redirect the call to the specified backend's implementation.


function call
with some
backend info*
User
The main
library
Backend
Finder
Output
The backend
library
Backend
translator

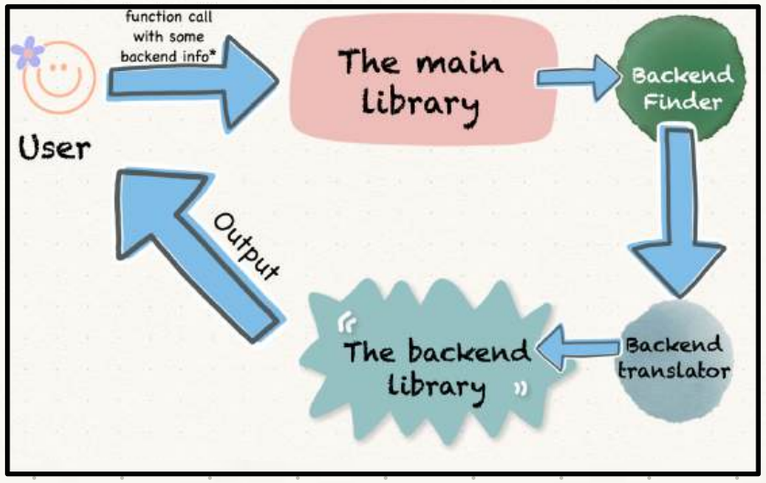

The main
library
NetworkX

# Backend Finder in NetworkX

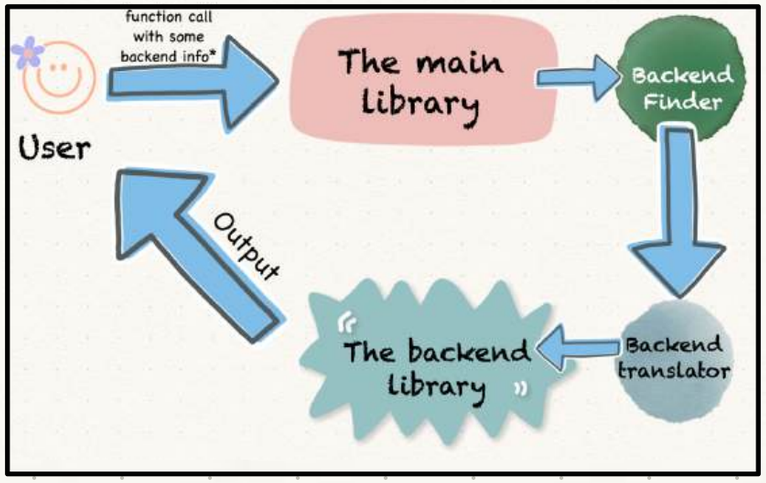


# Backend Finder in NetworkX

*NetworkX discovers backend packages by loading the* `networkx.backends` *Python entry_point*



# Backend Finder in NetworkX




*NetworkX discovers backend packages by loading the* `networkx.backends` *Python entry_point*

*What is* `networkx.backends`*?*

*what is an entry_point?*

# NetworkX's entry_points

```
[project.entry-points."networkx.backends"]
parallel = "nx_parallel.interface:BackendInterface"
```

```
[project.entry-points."networkx.backend_info"]
parallel = "_nx_parallel:get_info"
```

```
[project.entry-points."networkx.backend_info"]
parallel = "_nx_parallel:get_info"
```

ntation/stable/reference/algorithms/generated/networkx.algorithms.sh...

ll Tutorial Reference Gallery Developer Releases Guides Search

```
1 - 4: 3
>>> length[3][2]
1
>>> length[2][2]
0
```

**Additional backends implement this function**

cugraph : ***GPU-accelerated backend.***

Negative cycles are not yet supported. `NotImplementedError` will be raised if there are negative edge weights. We plan to support negative edge weights soon. Also, callable `weight` argument is not supported.

**Additional parameters:**

**dtype** : ***dtype or None, optional***

The data type (np.float32, np.float64, or None) to use for the edge weights in the algorithm. If None, then dtype is determined by the edge values.

graphblas : ***OpenMP-enabled sparse linear algebra backend.***

**Additional parameters:**

**chunksize** : ***int or str, optional***

Split the computation into chunks; may specify size as string or number of rows. Default "10 MiB"

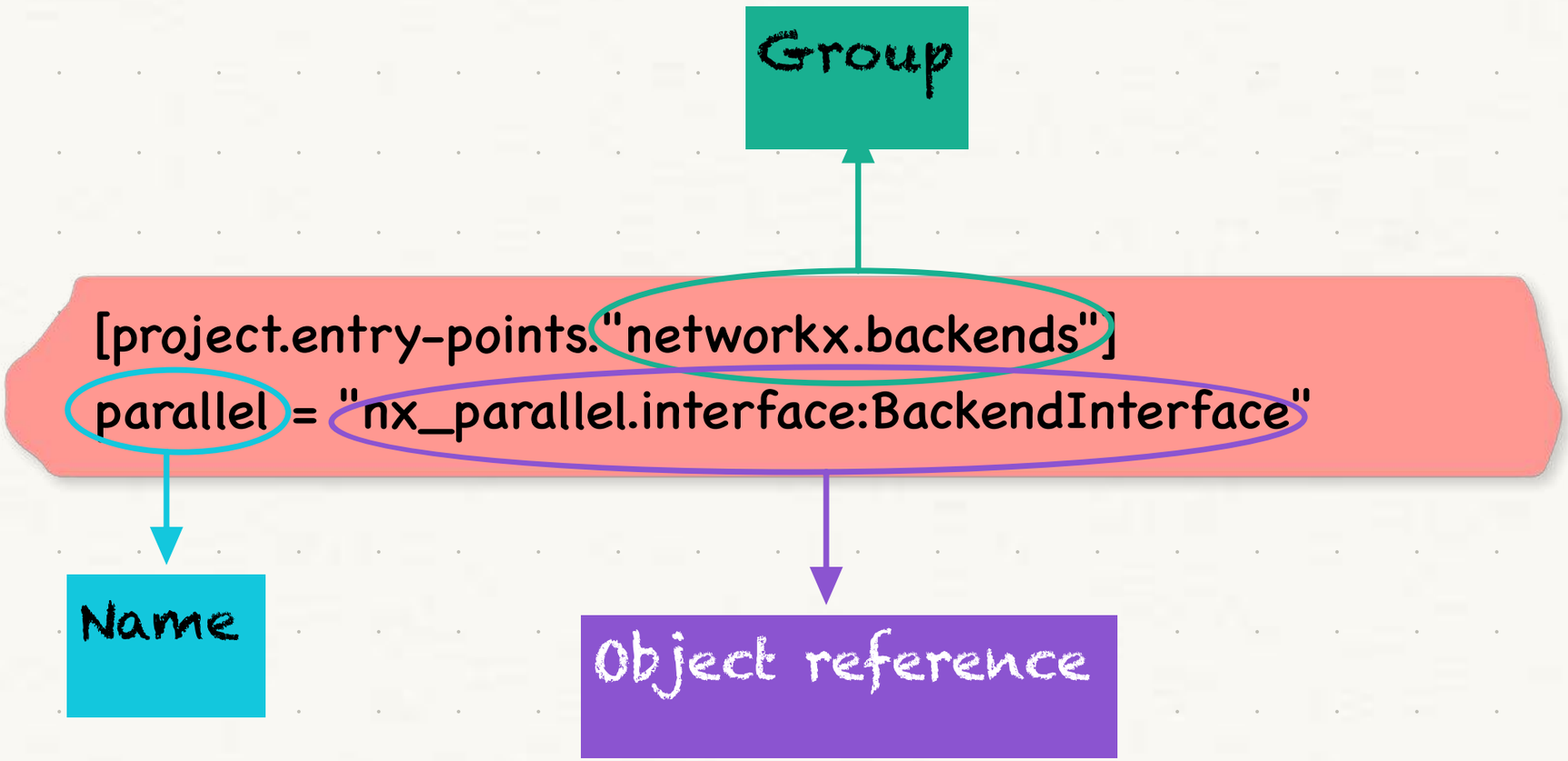
parallel : ***Parallel backend for NetworkX algorithms***

The parallel implementation first divides the nodes into chunks and then creates a generator to lazily compute shortest paths lengths for each node in `node_chunk`, and then employs joblib's `Parallel` function to execute these computations in parallel across all available CPU cores.

**Additional parameters:**

**get_chunks** : ***str, function (default = "chunks")***

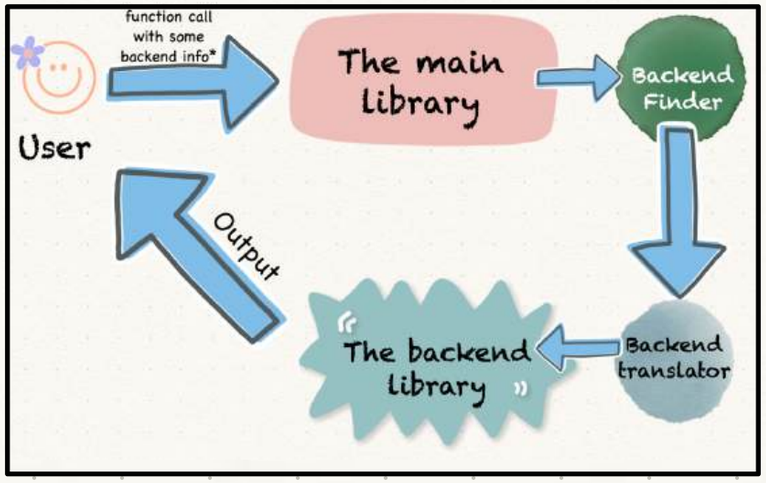
# Three required properties to define a Python entry_point:




ref. https://packaging.python.org/en/latest/specifications/entry-points/

# Backend Finder in NetworkX

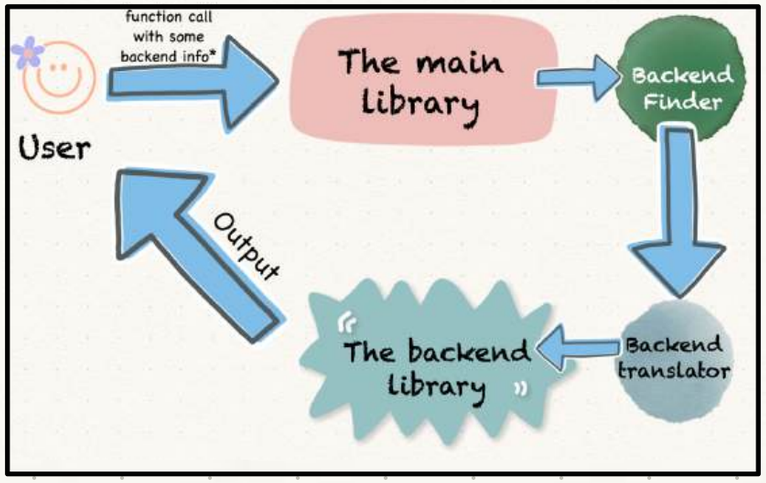
*NetworkX discovers backend packages by loading the `networkx.backends` Python entry_point*




Loading? -

```
>>> from importlib.metadata import entry_points
>>> entry_points(group="networkx.backends")
(EntryPoint(name='parallel',
value='nx_parallel.interface:BackendInterface',
group='networkx.backends'),
EntryPoint(name='nx_loopback',
value='networkx.classes.tests.dispatch_interface:bac
kend_interface', group='networkx.backends'))
```

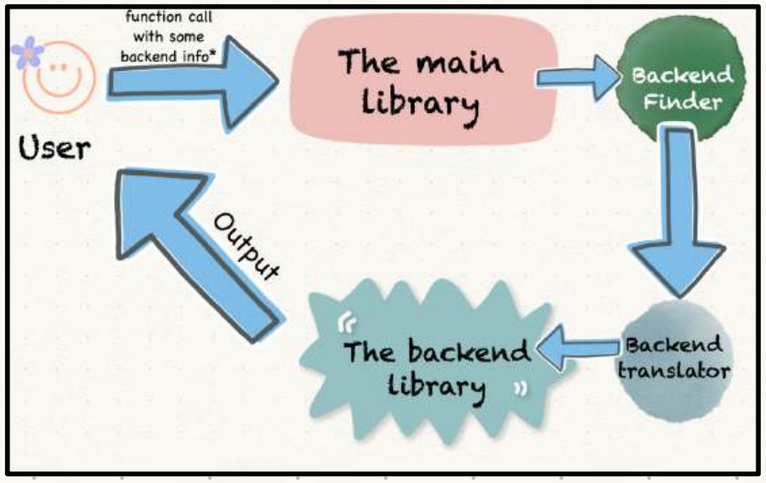
# Backend translator in NetworkX




*To convert args in the function call into something a backend can understand we use* `convert_from_nx` *and* `convert_to_nx` *(which are the attributes of the object referenced in the "networkx.backends" entry_point by the backend)*

`convert_from_nx` *and* `convert_to_nx` are just for args that are Graph objects.

# « The backend library » in NetworkX




What makes a networkx backend?

`networkx.backends` entry_point referring to a `BackendInterface` object with attributes - `convert_from_nx`, `convert_to_nx`, and all the algorithms implemented in the backend. And a backend graph object with `__networkx_backend__` attribute.

# So, to summarise...

```
nx.betweenness_centrality(G, backend="parallel")
```

```
nx.betweenness_centrality(cug)
```

```
with nx.config(backend_priority=[graphblas,]):
    nx.betweenness_centrality(G)
```

NetworkX

```
@_dispatchable
def betweenness_centrality(G, ...):
    ....
```

nx-parallel

```
def betweenness_centrality(G, ...):
    ....
```

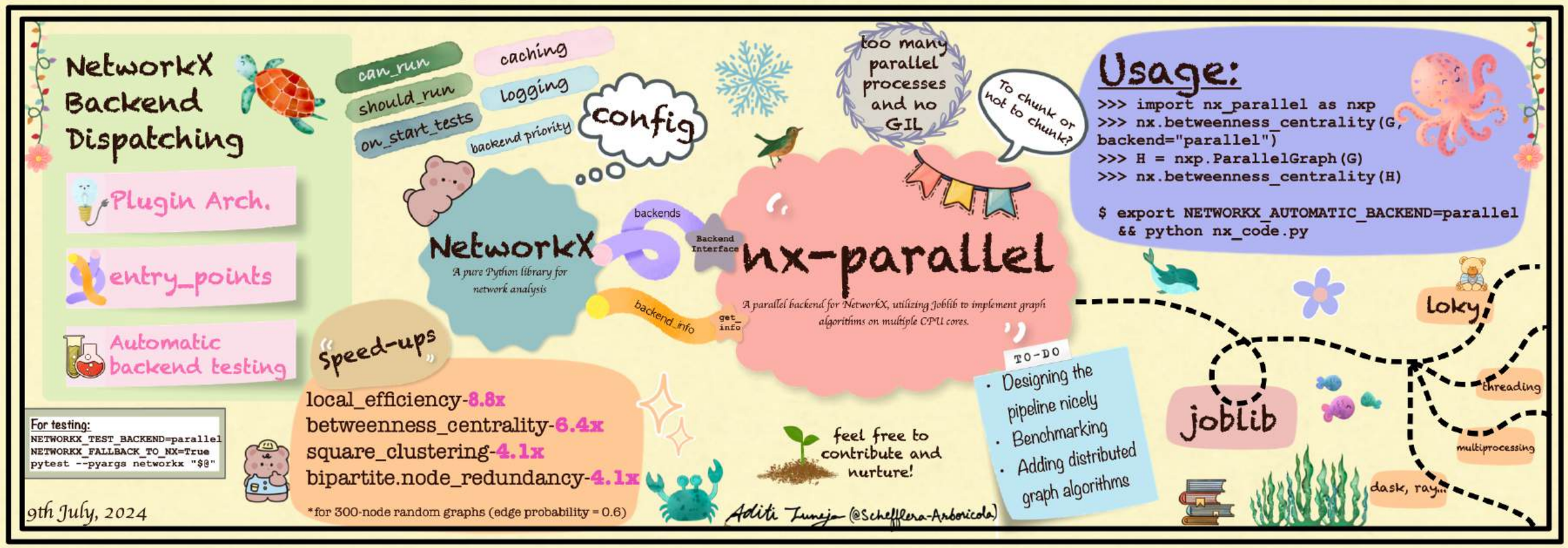
nx-cugraph

```
def betweenness_centrality(G, ...):
    ....
```

graphblas

```
def betweenness_centrality(G, ...):
    ....
```

# Going back to the poster...



# Understanding NetworkX's API Dispatching with a parallel backend

# Configs in NetworkX

```
>>> import networkx as nx
>>> nx.config
NetworkXConfig(backend_priority=[], backends=Config(parallel=ParallelConfig(active=False,
backend='loky', n_jobs=None, verbose=0, temp_folder=None, max_nbytes='1M',
mmap_mode='r', prefer=None, require=None, inner_max_num_threads=None,
backend_params={})), cache_converted_graphs=True)
```

# Configs in NetworkX

```
>>> import networkx as nx
>>> nx.config
NetworkXConfig(backend_priority=[], backends=Config(parallel=ParallelConfig(active=Fa
backend='loky', n_jobs=None, verbose=0, temp_folder=None, max_nbytes='1M',
mmap_mode='r', prefer=None, require=None, inner_max_num_threads=None,
backend_params={})), cache_converted_graphs=True)
```

Default configs are stored in the object referenced by the `networkx.backend_info` entry_point

# A standard nx-parallel algorithm

(a few days ago)

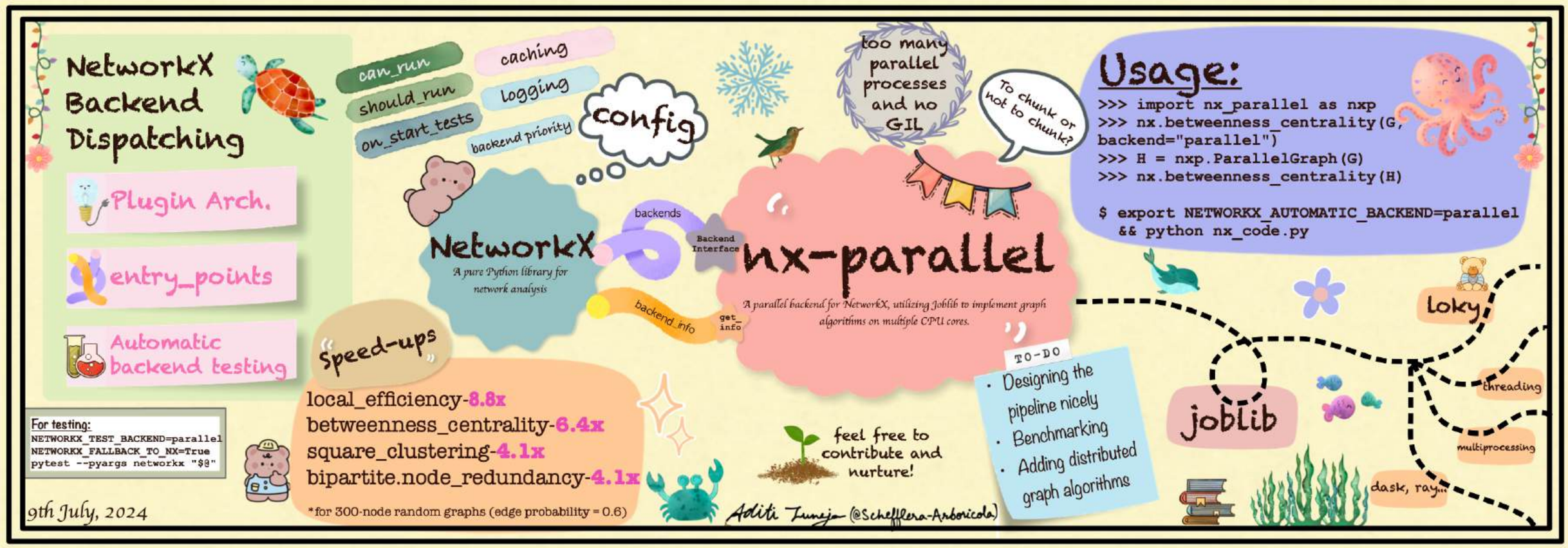
```
def xyz(G,..., get_chunks="chunks"):
    chunks = chunk(G)
    results = joblib.Parallel(n_jobs=-1)(
                joblib.delayed(chunks)
        )
    result = combine(results)
    return result
```

# A standard nx-parallel algorithm

(a few days ago)

allowing users to modify these parameters

```
def xyz(G,..., get_chunks="chunks"):
    chunks = chunk(G)
    results = joblib.Parallel(n_jobs=-1)(
                joblib.delayed(chunks)
        )
    result = combine(results)
    return result
```




Dashed line --> Solid line

# Configuring nx-parallel

## Joblib

```
joblib.parallel_config(n_jobs = 6, verbose=50)

G = nx.complete_graph(5)

# n_jobs = 6, verbose = 50
nx.square_clustering(G, backend="parallel")


with joblib.parallel_config(n_jobs=8):
    # n_jobs = 8, verbose = 50
    nx.square_clustering(G, backend="parallel")
```

## NetworkX

```
nx.config.backends.parallel.active = True
nx.config.backends.parallel.n_jobs = 6
nx.config.backends.parallel.verbose=50

G = nx.complete_graph(5)

# n_jobs = 6, verbose = 50
nx.square_clustering(G, backend="parallel")


with nx.config.backends.parallel(n_jobs=8):
    # n_jobs = 8, verbose = 50
    nx.square_clustering(G, backend="parallel")
```

# Syncing the two?

```
nx.config.backends.parallel.n_jobs = 6
joblib.parallel_config(verbose=50)

G = nx.complete_graph(5)

# n_jobs = 6, verbose = 50
nx.square_clustering(G, backend="parallel")


with nx.config.backends.parallel(n_jobs=8):
    # n_jobs = 8, verbose = 50
    nx.square_clustering(G, backend="parallel")
    with joblib.parallel_config(verbose=10):
        # n_jobs = 8, verbose = 10
        nx.square_clustering(G, backend="parallel")
```

ref. https://github.com/networkx/nx-parallel/issues/76 for more.

“Thank you :)”