Effortless Distributed Computing in Python

FOSDEM – Feb. 2025

Raphael J.
https://raphaelj.be

Menu

Three new Python libraries for distributed computing ! 🎉

- **Scaler**
  A **light-weight and resilient distributed scheduler**
- **Parfun**
  A hassle-free **map-reduce decorator**
- **Pargraph**
  A declarative **distributed graph engine**

Scaler

A **light-weight and resilient distributed scheduler**

Scaler

A distributed replacement for Python's built-in concurrent.futures **parallel executors**

Computes these functions in other processes, same computer

```
from concurrent.futures import Future, ProcessPoolExecutor

with ProcessPoolExecutor(max_workers=4) as executor:
    a: Future[float] = executor.submit(math.sqrt, 9)
    b: Future[float] = executor.submit(math.sqrt, 16)

    print(a.result() + b.result())  # prints "7.0"
```

Blocks until the result is available

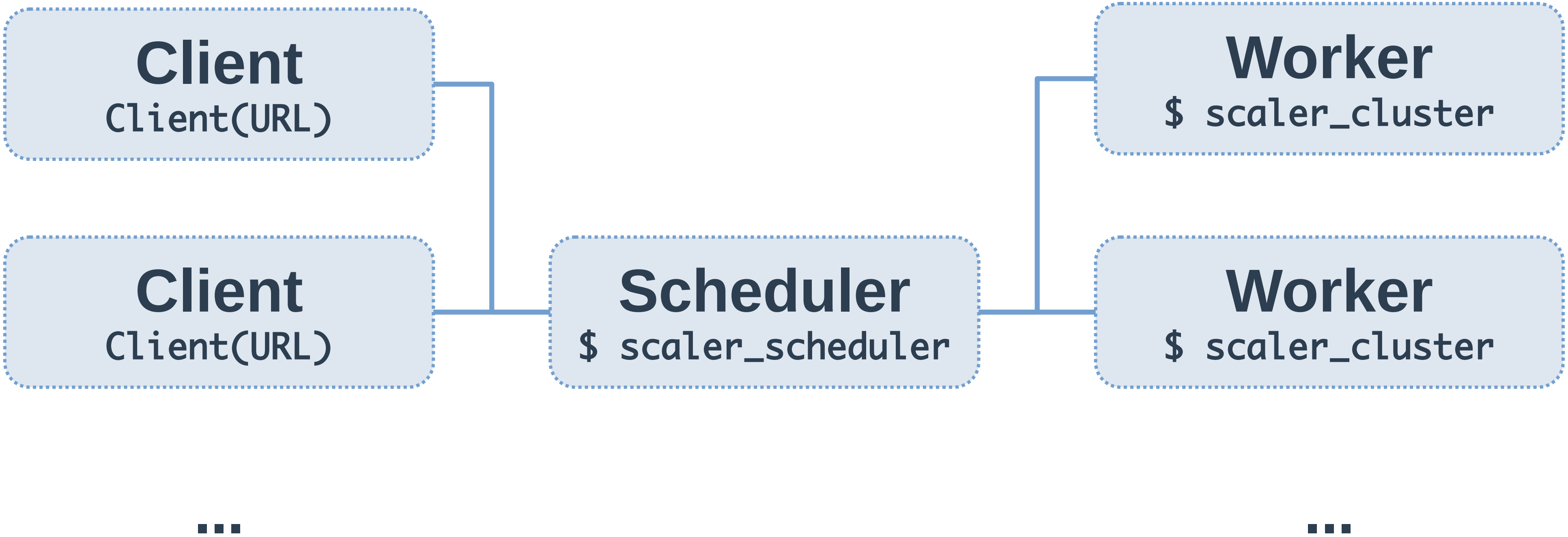
Scaler

A distributed replacement for Python's built-in concurrent.futures parallel executors

```
from scaler import Client, Future

with Client(cluster_URL) as executor:
    a: Future[float] = executor.submit(math.sqrt, 9)
    b: Future[float] = executor.submit(math.sqrt, 16)

    print(a.result() + b.result())  # prints "7.0"
```

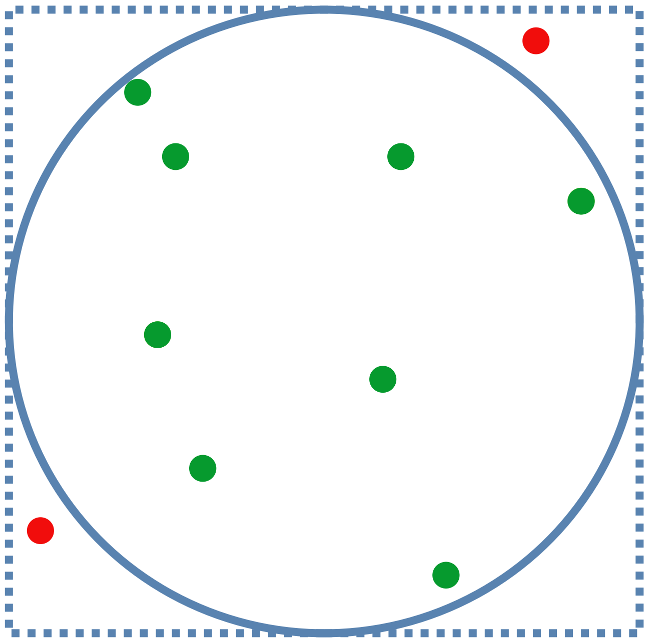
Computes these functions on a remote cluster

Scaler - Architecture



Scaler – Demo

Approximating π using Monte-Carlo



$$\pi \approx 4 * N_{\text{in circle}} / N_{\text{total}}$$
$$\approx 4 * 8 / 10$$
$$\approx 3.2$$

Scaler – Demo

```
def is_in_circle(x: float, y: float) -> bool:
    return x**2 + y**2 <= 1

def monte_carlo_pi(n_points: int) -> float:
    # Generates random X, Y coordinates within [-1..1]
    xs = [random.uniform(-1, 1) for i in range(0, n_points)]
    ys = [random.uniform(-1, 1) for i in range(0, n_points)]

    in_circle = [1 for x, y in zip(xs, ys) if is_in_circle(x, y)]

    return 4 * len(in_circle) / n_points
```

Scaler – Demo

```python
def is_in_circle(x: float, y: float) -> bool:
    return x**2 + y**2 <= 1

def monte_carlo_pi(n_points: int) -> float:
    # Generates random X, Y coordinates within [-1..1]
    xs = [random.uniform(-1, 1) for i in range(0, n_points)]
    ys = [random.uniform(-1, 1) for i in range(0, n_points)]

    in_circle = [1 for x, y in zip(xs, ys) if is_in_circle(x, y)]

    return 4 * len(in_circle) / n_points
```

Scaler – Demo

```
>>> monte_carlo_pi(1)
4.0
>>> monte_carlo_pi(10)
3.2
>>> monte_carlo_pi(100)
3.04
>>> monte_carlo_pi(1_000)
3.196
>>> monte_carlo_pi(10_000)
3.148
>>> monte_carlo_pi(100_000)
3.14176
```

Scaler – Demo

```python
def monte_carlo_pi_distributed(executor: Executor, n_points: int) -> float:
    n_tasks = 100
    n_points_per_task = n_points // n_tasks
    futures = [
        executor.submit(monte_carlo_pi, n_points_per_task)
        for _ in range(0, n_tasks)
    ]
    return sum(f.result() for f in futures) / n_tasks
```

Scaler – Demo

```python
def monte_carlo_pi_distributed(executor: Executor, n_points: int) -> float:
    n_tasks = 100
    n_points_per_task = n_points // n_tasks
    futures = [
        executor.submit(monte_carlo_pi, n_points_per_task)
        for _ in range(0, n_tasks)
    ]
    return sum(f.result() for f in futures) / n_tasks
```

Scaler – Demo

```
>>> %timeit -n 1 -r 1 monte_carlo_pi(1_000_000_000)
7min 18s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

>>> local_pool = ProcessPoolExecutor(max_workers=8)
>>> monte_carlo_pi_distributed(local_pool, 1_000_000_000)
3.14165369
>>> %timeit monte_carlo_pi_distributed(local_pool, 1_000_000_000)
57.6 s ± 16.92 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

>>> client = scaler.Client(scheduler_URL)
>>> monte_carlo_pi_distributed(client, 1_000_000_000)
3.14160956
>>> %timeit  monte_carlo_pi_distributed(client, 1_000_000_000)
12.7 s ± 174 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Scaler – Demo

```
>>> %timeit -n 1 -r 1 monte_carlo_pi(1_000_000_000)
7min 18s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

>>> local_pool = ProcessPoolExecutor(max_workers=8)
>>> monte_carlo_pi_distributed(local_pool, 1_000_000_000)
3.14165369
>>> %timeit monte_carlo_pi_distributed(local_pool, 1_000_000_000)
57.6 s ± 16.92 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

>>> client = scaler.Client(scheduler_URL)
>>> monte_carlo_pi_distributed(client, 1_000_000_000)
3.14160956
>>> %timeit  monte_carlo_pi_distributed(client, 1_000_000_000)
12.7 s ± 174 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

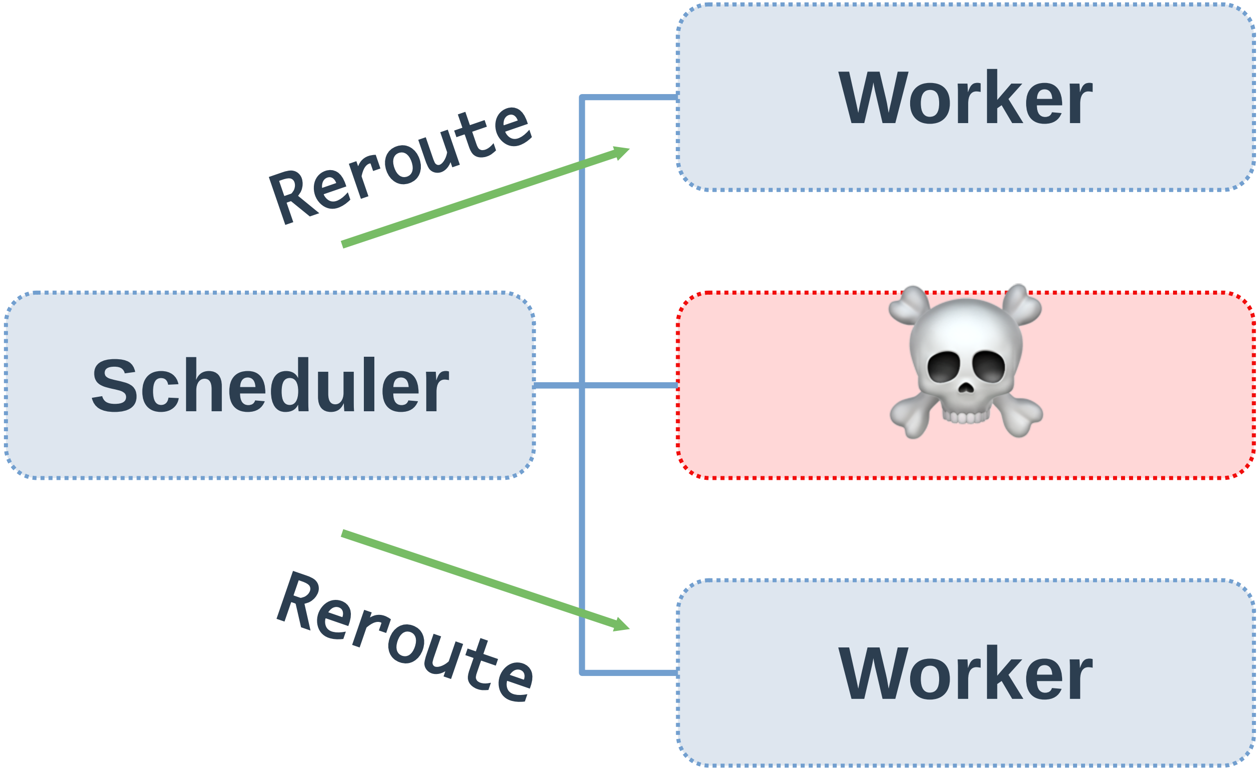
Scaler – Demo

```
>>> %timeit -n 1 -r 1 monte_carlo_pi(1_000_000_000)
7min 18s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

>>> local_pool = ProcessPoolExecutor(max_workers=8)
>>> monte_carlo_pi_distributed(local_pool, 1_000_000_000)
3.14165369
>>> %timeit monte_carlo_pi_distributed(local_pool, 1_000_000_000)
57.6 s ± 16.92 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

>>> client = scaler.Client(scheduler_URL)
>>> monte_carlo_pi_distributed(client, 1_000_000_000)
3.14160956
>>> %timeit  monte_carlo_pi_distributed(client, 1_000_000_000)
12.7 s ± 174 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

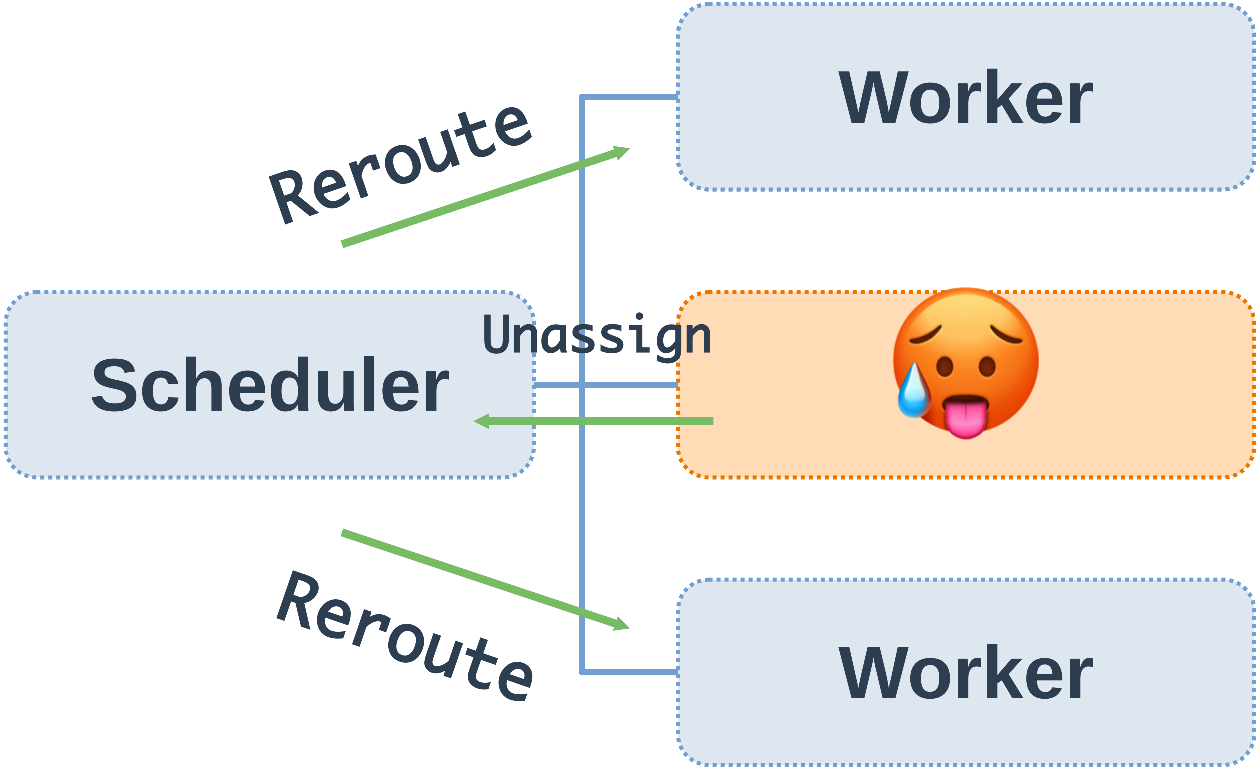
Scaler – scaler_top

```
scheduler          | object_manager       | task_manager        |      scheduler_sent           | scheduler_received
      cpu   4.0% |      num_of_objs 1055 |   unassigned      0 | WorkerHeartbeatEcho 61,338 |    WorkerHeartbeat 61,338
      rss  36.0M |          obj_mem  28K |      running     54 | ClientHeartbeatEcho    403 |    ClientHeartbeat    403
 rss_free 506.5G |                       |      success 1,026 |                Task  1,100 |  ObjectInstruction  1,053
                                         |       failed     20 |      ObjectResponse  1,163 |               Task  1,100
                                         |     canceled      0 |          TaskResult  1,046 |      ObjectRequest  1,163
                                         |    not_found      0 |                             |         TaskResult  1,046
-----------------------------------------------------------------------------------------------------------------------
Shortcuts: worker[n] agt_cpu[C] agt_rss[M] cpu[c] rss[m] rss_free[F] free[f] sent[w] queued[d] suspended[s] lag[l]

Total 135 worker(s)
                   worker agt_cpu agt_rss  [cpu]    rss os_rss_free free sent queued suspended   lag ITL |        client_manager
4899|scaler-demo|2ef7d59+    0.0%   34.0M 100.9% 432.5M      506.0G  999    1      0         0 0.2ms 111 | b'83751|Client|0af40'+ 54
4875|scaler-demo|1251b44+    0.0%   34.0M 100.4% 336.6M      505.0G  999    1      0         0 0.2ms 111 |
4871|scaler-demo|ec8caaf+    0.0%   34.0M 100.4% 614.6M      504.7G  999    1      0         0 0.3ms 111 |
4880|scaler-demo|fbdd38a+    0.0%   34.0M 100.4% 394.6M      505.4G  999    1      0         0 0.2ms 111 |
4900|scaler-demo|3aa51e3+    0.0%   34.0M 100.4%  98.4M      505.7G  999    1      0         0 0.2ms 111 |
4896|scaler-demo|90b595c+    0.0%   34.0M 100.4% 835.2M      485.6G 1000    0      0         0 0.2ms 111 |
4898|scaler-demo|7c7b45b+    0.0%   34.0M 100.4% 805.8M      485.6G  999    1      0         0 0.2ms 111 |
4918|scaler-demo|1e26637+    0.0%   34.0M 100.4% 818.9M      485.6G  999    1      0         0 0.2ms 111 |
4903|scaler-demo|b79c468+    0.0%   34.0M 100.4% 820.8M      485.6G  999    1      0         0 0.2ms 111 |
4909|scaler-demo|9b0389e+    0.5%   34.0M 100.4% 808.2M      485.6G  999    1      0         0 0.2ms 111 |
4913|scaler-demo|d27a8d8+    0.0%   34.0M 100.4% 812.9M      485.6G  999    1      0         0 0.3ms 111 |
4912|scaler-demo|e35daf7+    0.0%   34.0M 100.4% 822.2M      485.6G  999    1      0         0 0.2ms 111 |
4914|scaler-demo|7e3bf07+    0.5%   34.0M 100.4% 814.8M      485.6G  999    1      0         0 0.2ms 111 |
4922|scaler-demo|b656554+    0.0%   34.0M 100.4% 840.1M      485.6G 1000    0      0         0 0.2ms 111 |
4928|scaler-demo|dcdd6c3+    0.0%   34.0M 100.4% 821.6M      485.5G  999    1      0         0 0.2ms 111 |
4932|scaler-demo|ce428c8+    0.0%   34.0M 100.4% 824.2M      485.5G 1000    0      0         0 0.2ms 111 |
4935|scaler-demo|ce7d2ba+    0.0%   34.0M 100.4% 819.3M      485.5G  999    1      0         0 0.3ms 111 |
4936|scaler-demo|903cb35+    0.0%   34.0M 100.4% 818.3M      485.5G  999    1      0         0 0.3ms 111 |
4947|scaler-demo|7458f81+    0.0%   34.0M 100.4% 811.3M      485.5G  999    1      0         0 0.2ms 111 |
4948|scaler-demo|c13fb62+    0.0%   34.0M 100.4% 851.5M      485.5G 1000    0      0         0 0.2ms 111 |
4949|scaler-demo|5b517a2+    0.0%   34.0M 100.4% 820.2M      485.5G  999    1      0         0 0.2ms 111 |
4955|scaler-demo|3f3bbba+    0.0%   34.0M 100.4% 816.1M      485.6G  999    1      0         0 0.2ms 111 |
4952|scaler-demo|404bdbe+    0.0%   34.0M 100.4% 822.0M      485.5G  999    1      0         0 0.2ms 111 |
```

Scaler – Failure recovery



Scaler – Dynamic load balancing



Parfun

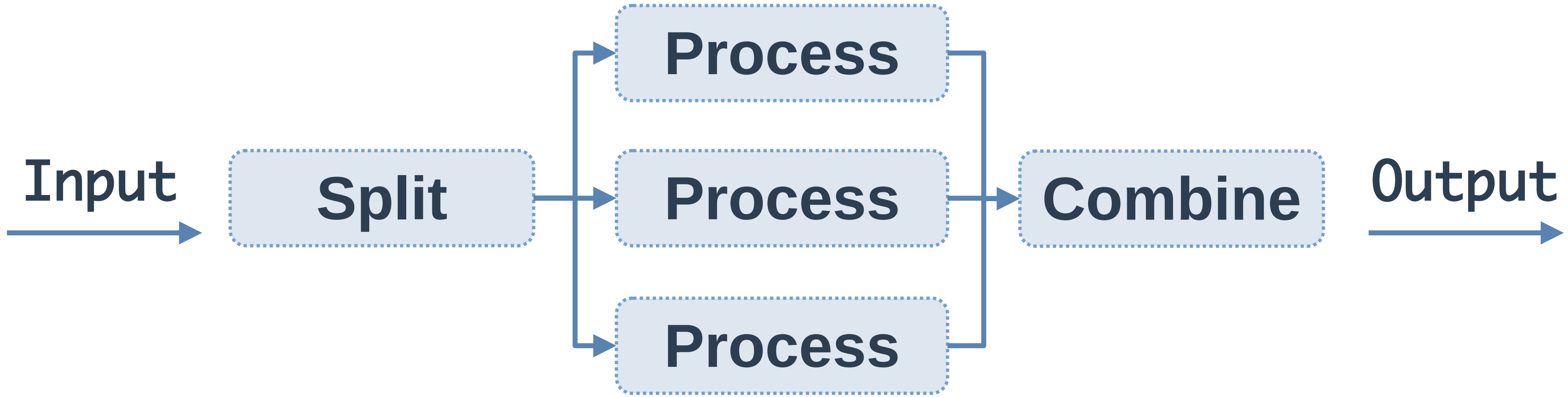
A hassle-free **map-reduce decorator**

Parfun – Count words in text

```
from collections import Counter

def count_words(lines: List[str]) -> Counter:
    counter = Counter()
    for line in lines:
        for word in line.split():
            counter[word] += 1

    return counter
```

```
>>> count_words(open("small_text.txt").readlines())
Counter({'the': 117,
         'and': 106,
         'of': 90,
         'to': 83,
         'in': 42,
         'right': 33,
        ...})
```

Parfun – Map reduce



Parfun – @parfun

```
from parfun import parfun
from parfun.partition.api import per_argument
from parfun.partition.collection import list_by_chunk

@parfun(
    split=per_argument(
        lines=list_by_chunk
    ),
    combine_with=sum,
)
def count_words(lines: List[str]) -> Counter:
    ...
```

```
>>> count_words(open("very_large_file.txt").readlines())
Counter({'the': 11700,
       ...
```

Parfun – @parfun

```python
from parfun import parfun
from parfun.partition.api import per_argument
from parfun.partition.collection import list_by_chunk

@parfun(
    split=per_argument(
        lines=list_by_chunk
    ),
    combine_with=sum,
)
def count_words(lines: List[str]) -> Counter:
    ...
```

```python
>>> count_words(open("very_large_file.txt").readlines())
Counter({'the': 11700,
        ...
```

Parfun – Find the optimal batch size

How to find the **optimal task batch size**?

- Too **small: overheads will large**
  - Communication, IPC, synchronization ...
- Too **large: parallelism will be low**

Parfun – Find the optimal batch size

Use **Machine-Learning**!

```
count_words()
   total CPU execution time: 0:00:00.174216.
   compute time: 0:00:00.165855 (95.20%)
      min.: 0:00:00.017239
      max.: 0:00:00.020540
      avg.: 0:00:00.018428
   total parallel overhead: 0:00:00.008361 (4.80%)
      total partitioning: 0:00:00.006238 (3.58%)
      average partitioning: 0:00:00.000693
      total combining: 0:00:00.002123 (1.22%)
   maximum speedup (theoretical): 8.48x
   total partition count: 9
      estimator state: running
        estimated partition size: 1638
```

Pargraph

A declarative **distributed graph engine**

Pargraph – Declarative graphs

```python
def generate_and_send_report(data_file_path: str, user_table: str) -> bool:
    data = read_data_file(data_file_path)

    processed_data = process_data(data)
    report = create_report(processed_data)

    users = read_postgres_table(user_table)
    email_list = extract_emails(users)

    success = send_report(report, email_list)

    return success
```

Pargraph – Declarative graphs

```
def generate_and_send_report(data_file_path: str, user_table: str) -> bool:
    data = read_data_file(data_file_path)

    processed_data = process_data(data)
    report = create_report(processed_data)

    users = read_postgres_table(user_table)
    email_list = extract_emails(users)

    success = send_report(report, email_list)

    return success
```

Can run concurrently !

Pargraph – Declarative graphs

```
from pargraph import delayed, graph

@delayed
def read_data_file(file_path: str) -> str:
    ...

@delayed
def read_postgres_table(table: str) -> List[Tuple]:
    ...

@delayed
def extract_emails(table_content: List[Tuple]) -> List[str]:
    ...


...

@graph
def generate_and_send_report(data_file_path: str, user_table: str) -> bool:
    ...
```

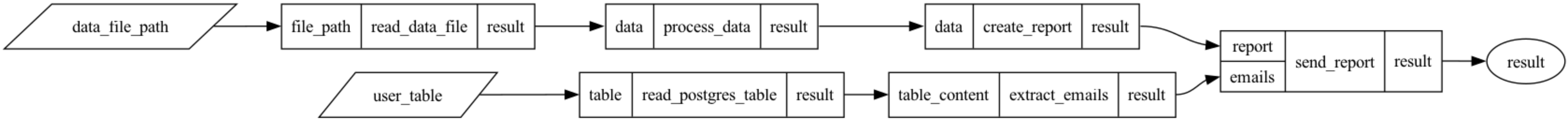
Pargraph – Declarative graphs

```python
from pargraph import delayed, graph

@delayed
def read_data_file(file_path: str) -> str:
    ...

@delayed
def read_postgres_table(table: str) -> List[Tuple]:
    ...

@delayed
def extract_emails(table_content: List[Tuple]) -> List[str]:
    ...


...

@graph
def generate_and_send_report(data_file_path: str, user_table: str) -> bool:
    ...
```

Leaf nodes

Pargraph – Declarative graphs

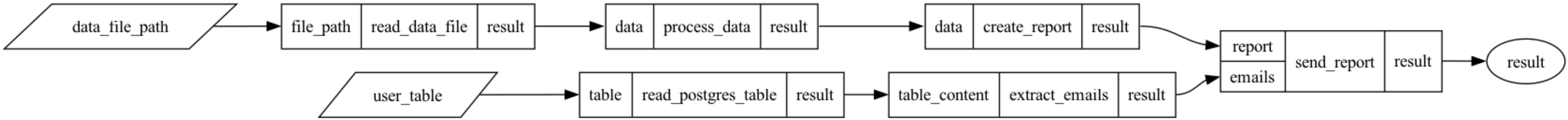
```
>>> generate_and_send_report.to_graph()
```




```
with Client(scheduler_url) as client:
    client.get(generate_and_send_report.to_graph(data_file_path=..., ...))
```

Pargraph – Declarative graphs

```
>>> generate_and_send_report.to_graph()
```




```
with Client(scheduler_url) as client:
    client.get(generate_and_send_report.to_graph(data_file_path=..., ...))
```

Thank you !

Q & A

Raphael J.
https://raphaelj.be