Implicit Parallelization Libraries

Inspiration: NESL

> For parallel computing to succeed in the mainstream parallel programming needs to be made as easy as sequential programming, or at least almost as easy. Currently many problems that can be solved in a couple dozen lines of sequential code require hundreds or sometimes thousands of lines of code to be solved efficiently in parallel. Furthermore, the parallel code is typically much harder to understand, modify and debug than its sequential counterpart. This has limited parallel programming to experts, and to applications in which the performance is absolutely critical.

Programming Parallel Algorithms
Guy E. Blelloch,
Communications of the ACM, 39(3),
March 1996.

Explicit vs. Implicit

Computing progress moves from explicit to implicit

From handcrafting code to letting computers do the work

Freeing up human creativity for more complex tasks

Two Challenges to Program Parallelization

1. Create tasks from individual functions or programs

2. Map the tasks onto machine models – cores, machines, grids

For parallel programming to succeed
we must make both as implicit as possible

Three Libraries to Solve the Two Challenges

@parfun

Function-level parallelization library

Program-level parallel graph computation library

Lightweight distributed computation engine

Function-Level Parallelization

Explicit vs. Implicit Vectorization in C

Explicit Vectorization

```
void add(int *X, int *Y, int *Z) {
    __mm128i *vecX = (__m128i*)X;
    __mm128i *vecY = (__m128i*)Y;
    __mm128i *vecZ = (__m128i*)Z;
    for (int i=0; i<n/4; i++) {
        _mm_store_si128(
            vecZ++,
            _mm_add_epi32(*vecX++, *vecY++)
        );
    }
}
```

Implicit Vectorization

```
void add(int *X, int *Y, int *Z) {
    for (int i = 0; i < n; i++) {
        Z[i] = X[i] + Y[i];
    }
}
```

Vectorization in Python

Pure Python does not support vectorized operations
But NumPy and pandas do!

```
def calc_ratio(row):
    return 100 * (row["x"] / row["y"])

# Not vectorized
df.apply(calc_ratio, axis="index")  # 5.6435 secs

# Vectorized
100 * (df["x"] / df["y"])           # 0.0043 secs
```

Magic!

Example: Market Cap Computation

Serial Market Cap Computation

```
def calculate_market_cap_usd(
    companies: pd.DataFrame, exchange_rates: pd.DataFrame
) -> pd.DataFrame:
    companies_with_exchange_rates = companies.merge(exchange_rates, on="currency")
    companies_with_exchange_rates["market_cap_usd"] = (
        companies_with_exchange_rates["share_price"]
        * companies_with_exchange_rates["num_shares"]
        * companies_with_exchange_rates["exchange_rate"]
    )
    return companies_with_exchange_rates
```

Parallel Market Cap Computation

Users must:

- Partition data
- Manage compute resources
- Schedule tasks
- Combine data

Ugh! Magic is missing

```python
def calculate_market_cap_usd_using_multiprocess(
    companies: pd.DataFrame, exchange_rates: pd.DataFrame
) -> pd.DataFrame:

    partition_data = [
        (partition, exchange_rates)
        for partition in np.array_split(
            companies, len(companies.index) // 1000
        )
    ]

    with multiprocessing.Pool(processes=5) as pool:
        results = pool.starmap(
            calculate_market_cap_usd, partition_data
        )

    return pd.concat(results, ignore_index=True)
```

@parfun

Parfun is a library providing helpers to run a Python function in parallel

Parfun Market Cap Computation

```
def calculate_market_cap_usd(
    companies: pd.DataFrame, exchange_rates: pd.DataFrame
) -> pd.DataFrame:
    companies_with_exchange_rates = companies.merge(exchange_rates, on="currency")
    companies_with_exchange_rates["market_cap_usd"] = (
        companies_with_exchange_rates["share_price"]
        * companies_with_exchange_rates["num_shares"]
        * companies_with_exchange_rates["exchange_rate"]
    )
    return companies_with_exchange_rates
```

Decorating Serial Functions with @parfun

```
@parfun(
    split=per_argument(companies=df_by_row),
    combine_with=df_concat,
)
```

```
def calculate_market_cap_usd(
    companies: pd.DataFrame, exchange_rates: pd.DataFrame
) -> pd.DataFrame:
    companies_with_exchange_rates = companies.merge(
        exchange_rates, on="currency"
    )
    companies_with_exchange_rates["market_cap_usd"] = (
        companies_with_exchange_rates["share_price"]
        * companies_with_exchange_rates["num_shares"]
        * companies_with_exchange_rates["exchange_rate"]
    )
    return companies_with_exchange_rates
```

Parfun:

- Partitions data
- Manages compute resources
- Schedules tasks
- Combines results

split: what function arguments to partition on and how to partition them
combine_with: how to aggregate results

Conceptually the same as map-reduce in other parallelization frameworks

Configuring Backends for Parfun

```python
@parfun(
    split=per_argument(companies=df_by_row),
    combine_with=df_concat,
)
def calculate_market_cap_usd(companies: pd.DataFrame, exchange_rates: pd.DataFrame) -> pd.DataFrame:
    ...
```

Install Parfun

```
$ pip install parfun
```

Run in parallel (using python local multiprocessing; also supports other backends such as Dask and Scaler)

```
>>> set_parallel_backend("local_multiprocessing")
>>> calculate_market_cap_usd(…)
```

Run in serial (Run @parfun in a single process)

```
>>> set_parallel_backend("local_single_processing")
>>> calculate_market_cap_usd(…)
```

Disable @parfun (For debugging purposes)

```
>>> set_parallel_backend("none")
>>> calculate_market_cap_usd(…)
```

Parfun Examples (1-D Partition)

```
@parfun(
    split=all_arguments(list_by_chunk),
    combine_with=list_concat,
)
def batch_calculate_marginal_matrices_positive(positive_annual_list):
    ...
```

```
@parfun(
    split=per_argument(
        marginal=df_by_group(by=["industry_code", "country_region_code"])
    ),
    combine_with=df_concat,
)
def cumulative_pd_with_alternative_start(marginal, start_quarters_list, max_interval):
    ...
```

Parfun Examples (2-D Partition)

calculate (year *i*) x (date *j*) grid in parallel

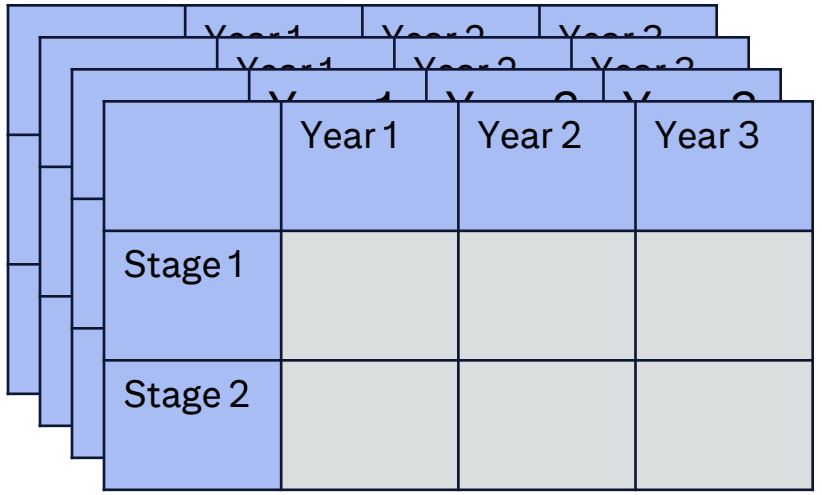
```
@parfun(
    split=multiple_arguments(
        ("year_i", "rate_date_i"),
        list_by_chunk,
    ),
    combine_with=df_concat,
)
def generate_discounting_and_qtr_rate_batch(
    year_i: List[int],
    rate_date_i: List[Annotated[pd.Period, "Q"]],
    portfolio: pd.DataFrame,
    ...
):
    ...
```

	Year 1	Year 2	Year 3
Date 1			
Date 2			
…	…	…	…

Parfun Examples (3-D Partition)

calculate (year *i*) x (stage *j*) grid in parallel, with partitioned portfolio

```
@parfun(
    split=all_arguments(transition_rate_partition_dfs_by_chunk),
    combine_with=concat_list_of_dfs_and_exception,
)
def calculate_transition_rate_for_all_years_parallel(
    portfolio_df: pd.DataFrame,
    year_i_list: List[int],
    year_j_list: List[int],
    ...
):
    ...
```



Why Parfun?

Pros

- **No extra knowledge needed about parallel programming**--user can focus on model and code design
- Uses Python generators to **reduce task overhead and memory usage**
  - Splits and combines data on-the-fly
  - Some split functions automatically estimate optimal partition sizes
- **Can be disabled** to restore sequential computation
  - Use “none” backend to run functions sequentially

Cons

- Only works on *pure functions*
- Can only parallelize *data parallel* functions where parallelization depends upon chunking the inputs

Predefined Split and Combine Functions

Split functions

- `list_by_chunk()`: Split one or multiple iterables by chunks of identical sizes
- `df_by_row()`: Split one or multiple DataFrames by chunks of identical numbers of rows
- `df_by_group()`: Split one or multiple DataFrames by groups of identical values of some columns

Combine functions

- `list_concat()`: Concatenates a collection of lists into a single list
- `unzip()`: Opposite of zip()
- `df_concat()`: Concatenate list of DataFrames into one DataFrame

@parfun

Features and Toolkits

Backend agnostic to support multiprocessing, Dask, Scaler, etc.

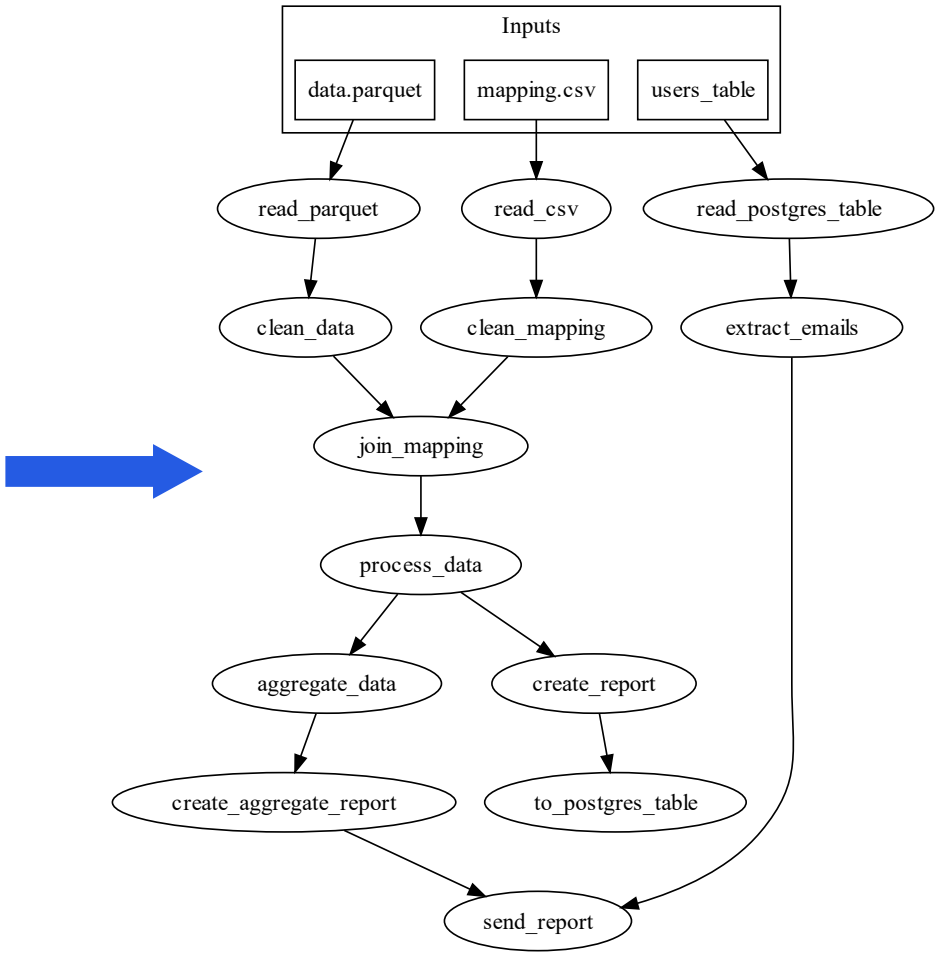
Predefined, easy-to-use Split functions

Predefined, easy-to-use Combine functions

Program-Level Parallelization

Insight: Programs Are Graphs

```python
def generate_report(data_parquet, mapping_csv, users_table):
    data_df = read_parquet(data_parquet)
    mapping_df = read_csv(mapping_csv)
    users_df = read_postgres_table(users_table)

    clean_data_df = clean_data(data_df)
    clean_mapping_df = clean_mapping(mapping_df)
    email_list = extract_emails(users_df)

    joined_data_df = join_mapping(clean_data_df, clean_mapping_df)
    processed_data = process_data(joined_data_df)

    aggregated_data = aggregate_data(processed_data)
    report = create_report(process_data)

    aggregated_report = create_aggregate_report(aggregated_data)
    to_postgres_table(report)

    send_report(aggregated_report, email_list)
```



Manual Program-Level Parallelization

When programs get larger and have more dependencies, parallelizing becomes harder

The ordering of function calls and syncs impacts the performance of your program

```
def generate_report(data_parquet, mapping_csv, users_table):
    data_df = read_parquet(data_parquet)
    mapping_df = read_csv(mapping_csv)
    users_df = read_postgres_table(users_table)

    clean_data_df = clean_data(data_df)
    clean_mapping_df = clean_mapping(mapping_df)
    email_list = extract_emails(users_df)

    joined_data_df = join_mapping(clean_data_df, clean_mapping_df)
    processed_data = process_data(joined_data_df)

    aggregated_data = aggregate_data(processed_data)
    report = create_report(process_data)

    aggregated_report = create_aggregate_report(aggregated_data)
    to_postgres_table(report)

    send_report(aggregated_report, email_list)
```

Manual Program-Level Parallelization

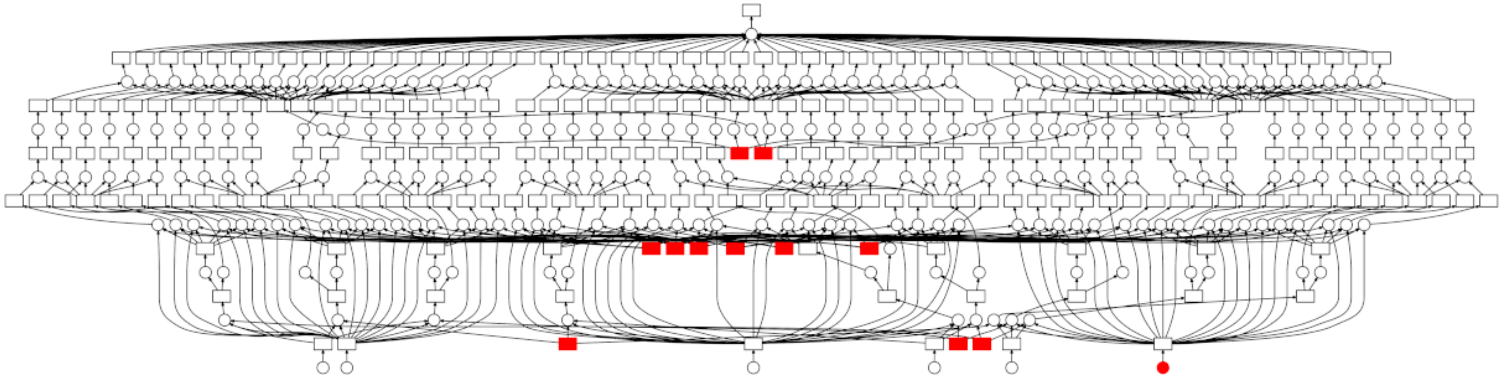
```python
def generate_report_parallel(data_parquet, mapping_csv, users_table):
    with concurrent.futures.ProcessPoolExecutor() as executor:
        data_df_future = executor.submit(read_parquet, data_parquet)
        mapping_df_future = executor.submit(read_csv, mapping_csv)
        users_df_future = executor.submit(read_postgres_table, users_table)

        concurrent.futures.wait([data_df_future, mapping_df_future, users_df_future])

        clean_data_df_future = executor.submit(clean_data, data_df_future.result())
        clean_mapping_df_future = executor.submit(clean_mapping, mapping_df_future.result())
        email_list_future = executor.submit(extract_emails, users_df_future.result())

        concurrent.futures.wait([clean_data_df_future, clean_mapping_df_future, email_list_future])

        joined_data_df_future = executor.submit(
            join_mapping, clean_data_df_future.result(), clean_mapping_df_future.result()
        )
        processed_data_future = executor.submit(process_data, joined_data_df_future.result())

        aggregated_data_future = executor.submit(aggregate_data, processed_data_future.result())
        report_future = executor.submit(create_report, processed_data_future.result())

        concurrent.futures.wait([aggregated_data_future, report_future])

        aggregated_report_future = executor.submit(create_aggregate_report, aggregated_data_future.result())
        to_postgres_table_future = executor.submit(to_postgres_table, report_future.result())

        concurrent.futures.wait([aggregated_report_future, to_postgres_table_future])

        send_report_future = executor.submit(
            send_report, aggregated_report_future.result(), email_list_future.result()
        )
        send_report_future.result()
```

When programs get larger and have more dependencies, parallelizing becomes harder

Why a Graph?

- Most programs run sequentially – one task at a time
- By converting to a graph, opportunities for parallel execution can be uncovered, which would otherwise be difficult to identify within code
- Other optimizations such as dead code elimination and computation inlining can be performed

1



1. Rocklin, Matt. *Dask: Parallel Computing in Python*. 2017, https://matthewrocklin.com/slides/strata-london-2017.html#/4.

@pargraph

Graph Computation in Parallel

Using Pargraph to Generate Graphs

To generate a graph of a program function, decorate the function with Pargraph's @graph decorator

```
@graph
def generate_report(data_parquet, mapping_csv, users_table):
    data_df = read_parquet(data_parquet)
    mapping_df = read_csv(mapping_csv)
    users_df = read_postgres_table(users_table)

    clean_data_df = clean_data(data_df)
    clean_mapping_df = clean_mapping(mapping_df)
    email_list = extract_emails(users_df)

    joined_data_df = join_mapping(clean_data_df, clean_mapping_df)
    processed_data = process_data(joined_data_df)

    aggregated_data = aggregate_data(processed_data)
    report = create_report(process_data)

    aggregated_report = create_aggregate_report(aggregated_data)
    to_postgres_table(report)

    send_report(aggregated_report, email_list)
```

Running a Function as a Graph Is Optional

A @graph decorated function behaves the same as a normal sequential Python function, allowing users to debug programs effortlessly

```
generate_report("data.parquet", "mapping.csv", "users_table")
```

To run the function in parallel as a graph, minimal code changes are needed:

```
generate_report_graph, keys = generate_report.to_graph().to_task_graph(
    "data.parquet", "mapping.csv", "users_table"
)

graph_engine.get(generate_report_graph, keys)
```

Example: TeraSort

TeraSort is a parallel sorting algorithm

1. Data is partitioned according to key range
2. Each data partition is sorted independently
3. Sorted data partitions are combined according to key range order

TeraSort Atomic Functions

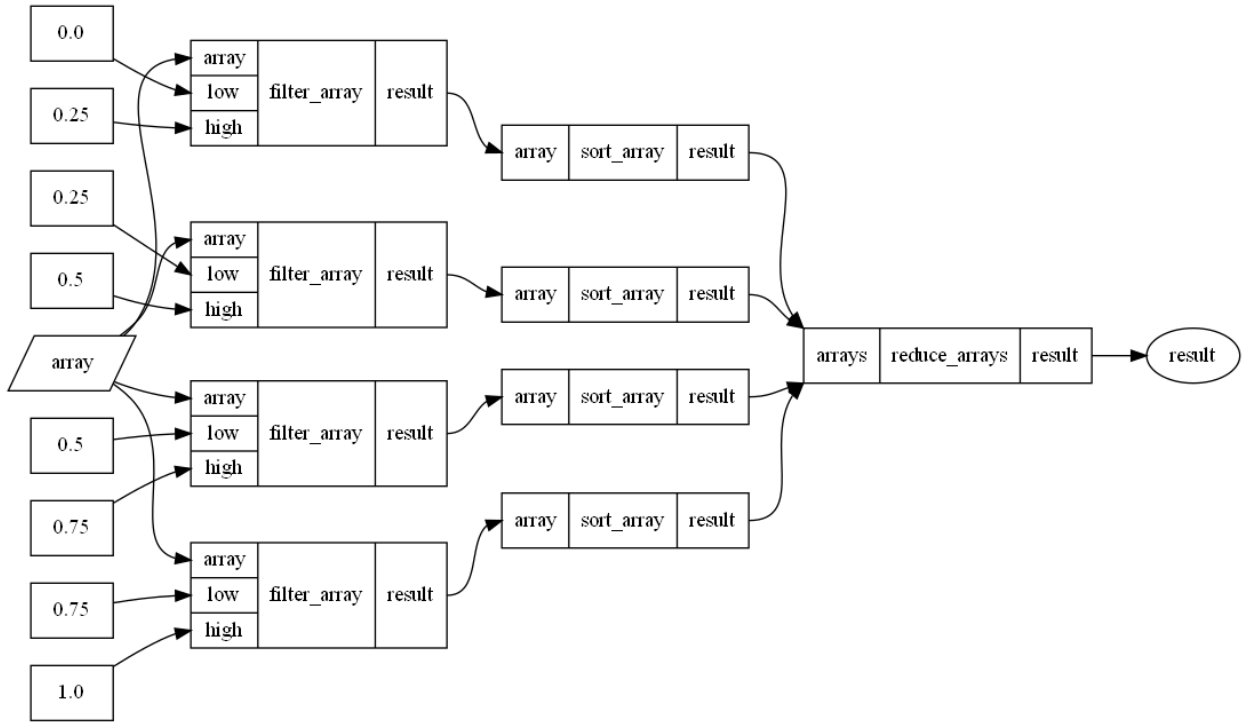
1. filter_array()
2. sort_array()
3. reduce_array()

```python
@delayed
def filter_array(array: np.ndarray, low: float, high: float) -> np.ndarray:
    return array[(low <= array) & (array < high)]


@delayed
def sort_array(array: np.ndarray) -> np.ndarray:
    return np.sort(array)


@delayed
def reduce_arrays(*arrays: np.ndarray) -> np.ndarray:
    return np.concatenate(arrays)
```

TeraSort Graph Function

```
@graph
def terasort(
    array: np.ndarray, partition_count: int
) -> np.ndarray:
    return reduce_arrays(
        *(
            sort_array(
                filter_array(
                    array,
                    i / partition_count,
                    (i + 1) / partition_count
                )
            )
            for i in range(partition_count)
        )
    )
```

With partition count 4



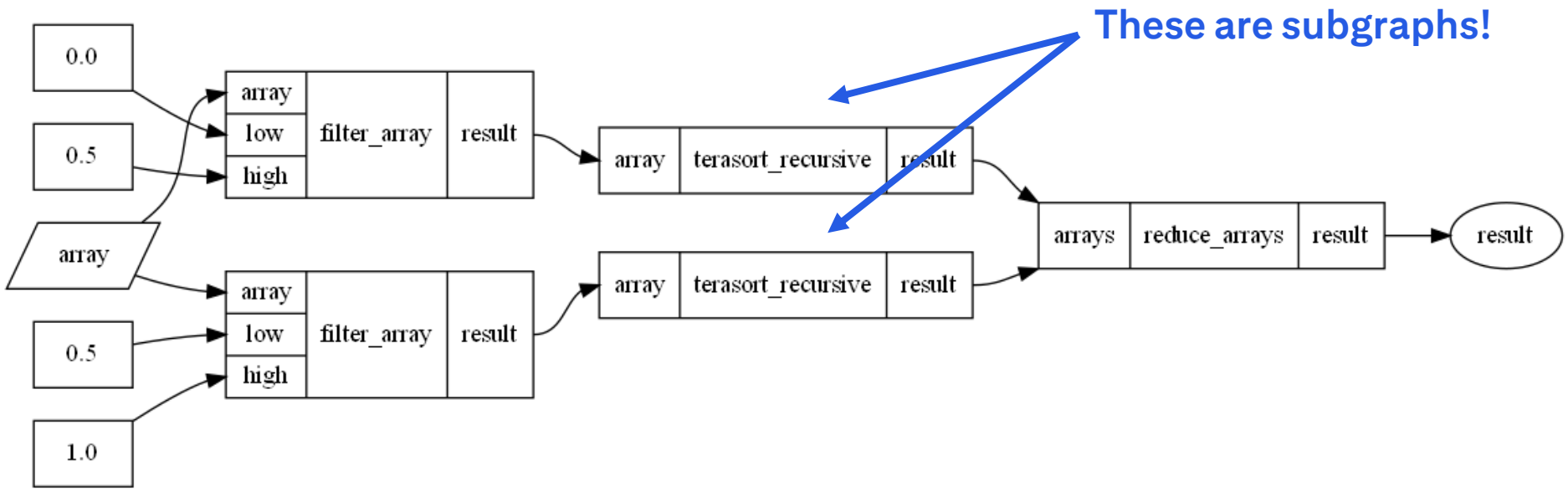
Recursive TeraSort Graph Function

Graph functions can be called within graph functions to create more complex graph structures

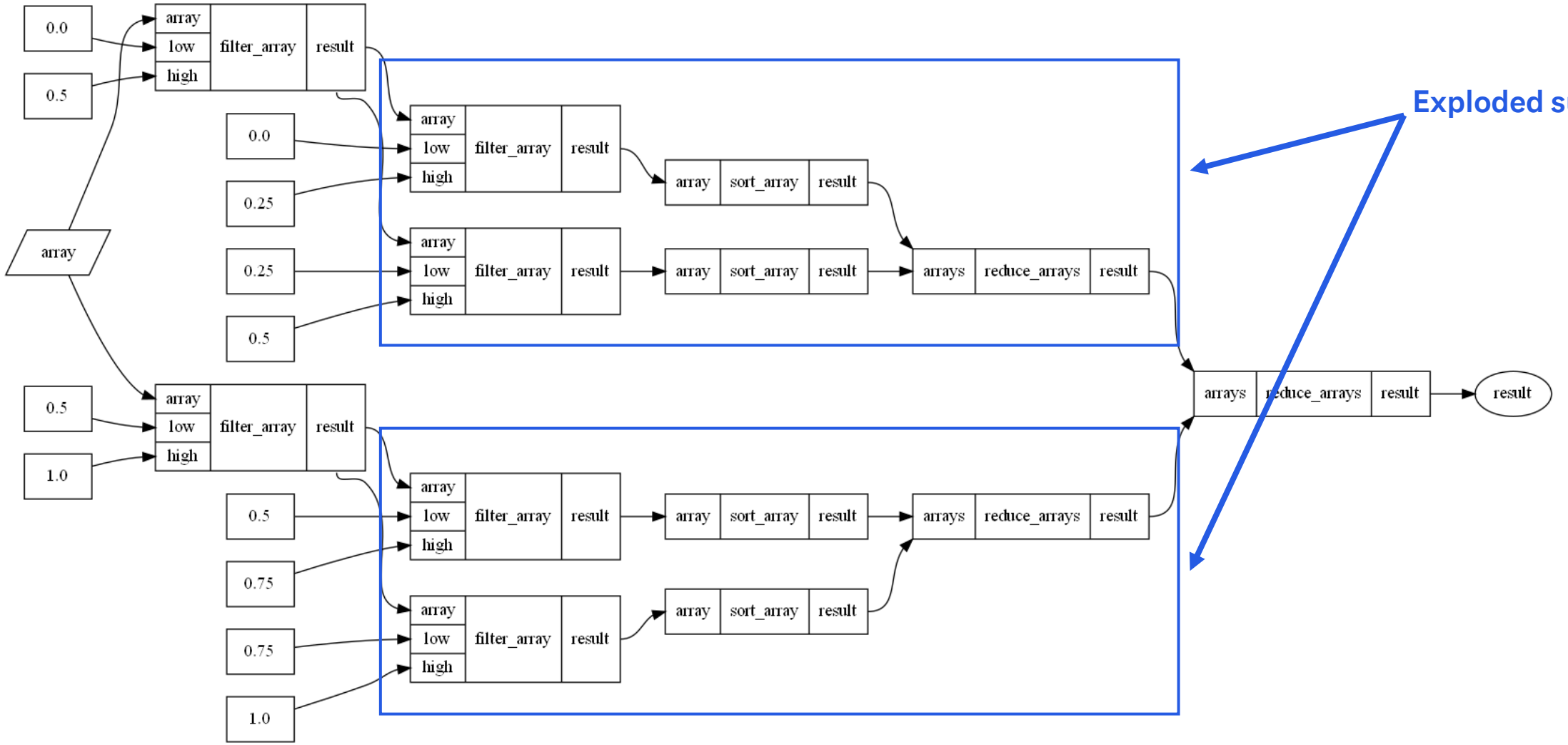
```
@graph
def terasort_recursive(
    array: np.ndarray, partition_counts: List[int], _low: float = 0, _high: float = 1
) -> np.ndarray:
    if len(partition_counts) == 0:
        return sort_array(array)

    partition_count, *partition_counts = partition_counts

    sorted_partitions = []
    for i in range(partition_count):
        low = _low + (_high - _low) * (i / partition_count)
        high = _low + (_high - _low) * ((i + 1) / partition_count)
        sorted_partitions.append(
            terasort_recursive(filter_array(array, low, high), partition_counts, low, high)
        )

    return reduce_arrays(*sorted_partitions)
```

Recursive TeraSort Graph Function

With partition counts 2, 2



Recursive TeraSort Graph Function



However, Pargraph Is NOT a Silver Bullet

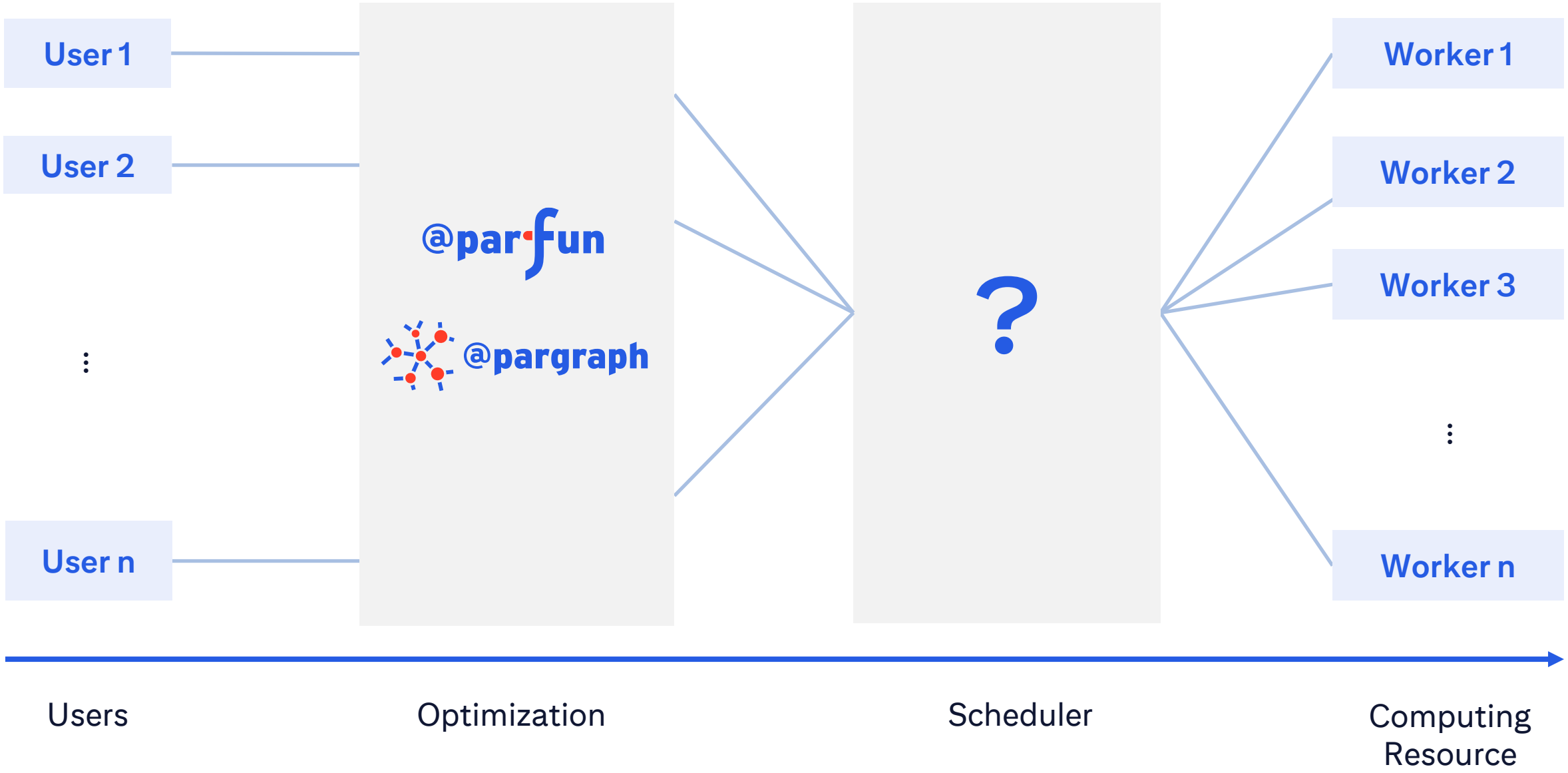
Not all Python functions can be converted to graphs

Generally, functions must be:

- Pure (have no side effects)
- Have named parameters, no variadic arguments

Now, how do we map tasks onto compute units – cores, processors, machines, grids?

How Tasks Are Mapped to Compute



We Need a Task Scheduler

Parfun and Pargraph split functions and programs into individual tasks

These tasks must be scheduled onto compute – cores, processors and machines

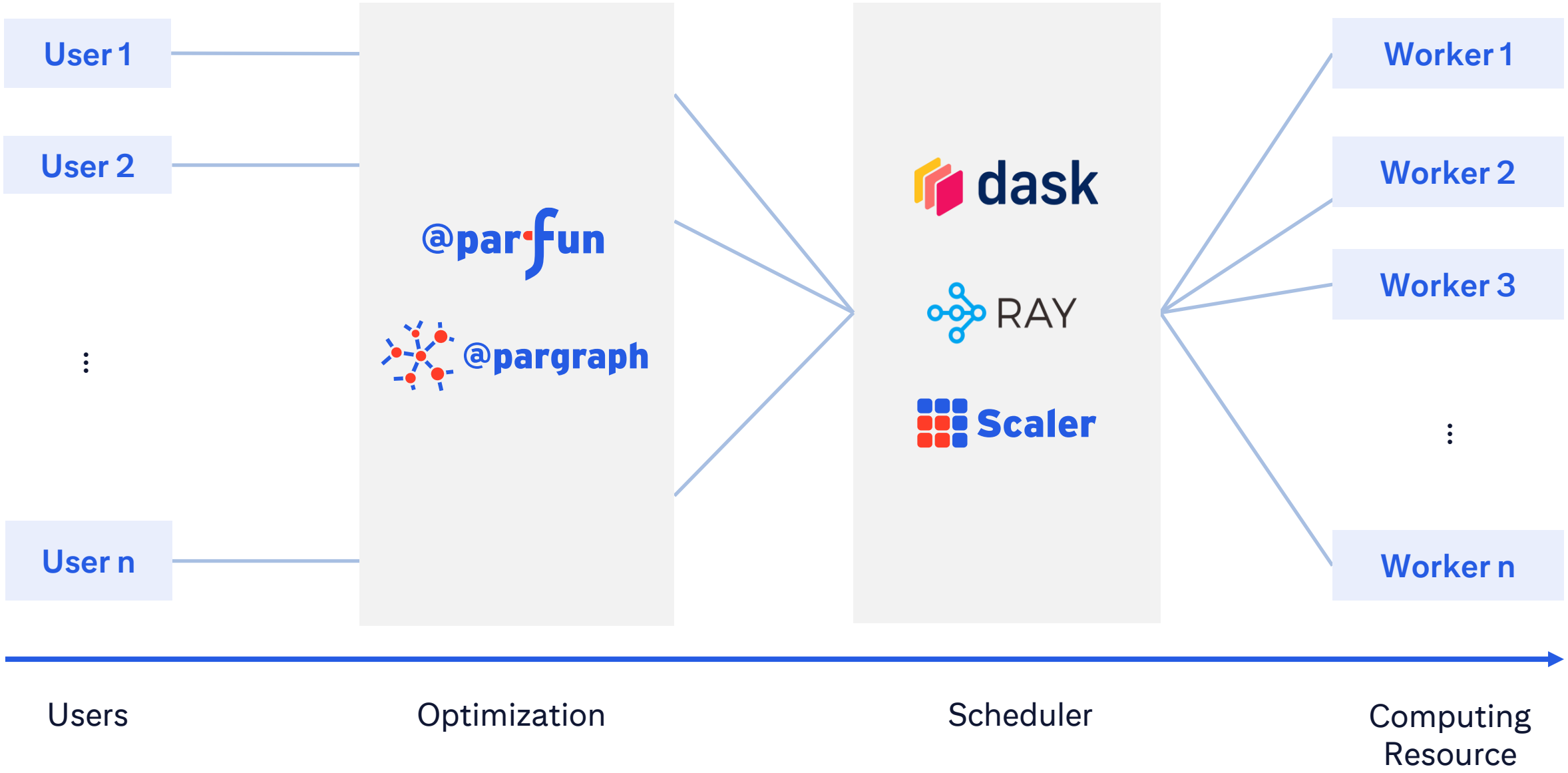
We need a generic task scheduler which can scale between Servers and the Grid

Requirements

- Efficiency: low latency task scheduling and rebalancing
- Simple coordination protocol: support for custom clients, workers and scheduler implementations
- Graph support: scheduler should support graphs as a type of task

Efficient and reliable distributed computation engine

How Tasks Are Mapped to Compute



Why not Dask?

- Dask has been the mainstay of distributed programming in the Python ecosystem for both in-core and out-of-core computations
- Dask's increasing focus on out-of-core dataframes means its core implementation is increasingly complex
- Not ideal or efficient for simple task scheduling

High complexity & overhead

- High feature-set to support dataframe-centric compute
- 5.2x the task scheduling overhead compared to Scaler

Inefficient

- Cannot perform load rebalance
- Large jobs cause hanging

Unreliable

- Occasional deadlocks
- Hard to debug

Comparing Scaler, Dask, Multiprocessing

Feature	Scaler	Dask	Multiprocessing
Multiple hosts	Yes	Yes	No
Worker monitoring	Yes	Yes	No
IO protocol	ZMQ (TCP/Inproc/IPC)	TCP	IPC
Reschedule on worker failure	Yes	Yes	No
Global task rebalance	Yes	No	No
Task graph support	Yes (Large graph support*)	Yes	No
API support	Function	Function and DataFrame API	Function
Serialization	Customizable serialization (with cloudpickle as default)	cloudpickle	Standard pickle
Lines of Code	3.7k	152.2k	0.7k

* Scaler uses GraphBLAS (high-performance sparse matrix library) which improves task-scheduling performance for large graphs (10k+ nodes)

What about Ray or Spark?

Ray

- Complicated scheduler architecture—each machine has its own scheduler
- Poor graph scheduling performance
- Focus is distributed compute for machine learning applications

Spark

- Requires hosting a separate Spark cluster
- Large-scale code changes needed to conform to Spark APIs
- Focus is large out-of-core datasets and implicit data parallelism

Benchmark with Sample Compute Pipeline

Backend	Duration	Change
Multiprocessing	51m 17s	Baseline
Scaler	51m 45s	+ 0.91%
Scaler Graph	29m 20s	- 42.80%
Dask	1h 34m 27s	+ 84.17%
Dask Graph*	2h 58m 05s	+ 247.25%

* The cause of the severe performance degradation in Dask is yet to be determined.

We believe it is due to Dask's scheduling overhead for tasks submitted in real-time.

Thanks
Q/A