Parallelization in Python — Dask

Arnav Sood

NYU

April 22, 2016

Dask Hierarchy

Data Structures: Job fits easily into operations on common data structures.

Dask Hierarchy

Data Structures: Job fits easily into operations on common data structures.

Imperatives: Job is more complex, but doesn't warrant an entire custom graph.

Dask Hierarchy

Data Structures: Job fits easily into operations on common data structures.
Imperatives: Job is more complex, but doesn't warrant an entire custom graph.
Custom Graphs: Intricate job.

Dask Hierarchy

Data Structures: Job fits easily into operations on common data structures.
Imperatives: Job is more complex, but doesn't warrant an entire custom graph.
Custom Graphs: Intricate job.

Schedulers: Take the above specs, and run as efficiently as possible.

Chunked Algorithms

```
# Try chunked matrix operations
import dask.array as da
import numpy as np
baseArray = np.random.rand(10000,10000)
# Make positive-definite
baseArray = np.dot(baseArray,baseArray.transpose())
smallChunks = da.from_array(baseArray,chunks=(100))
largeChunks = da.from_array(baseArray,chunks=(1000))
```

Chunked Algorithms

```
# Try chunked matrix operations
import dask.array as da
import numpy as np
baseArray = np.random.rand(10000,10000)
# Make positive-definite
baseArray = np.dot(baseArray,baseArray.transpose())
smallChunks = da.from_array(baseArray,chunks=(100))
largeChunks = da.from_array(baseArray,chunks=(1000))
```

```
In [2]: %time np.linalg.cholesky(baseArray)
        %time da.linalg.cholesky(smallChunks)
        %time da.linalg.cholesky(largeChunks)

CPU times: user 21.5 s, sys: 849 ms, total: 22.4 s
Wall time: 6.43 s
CPU times: user 385 ms, sys: 26.4 ms, total: 412 ms
Wall time: 440 ms
CPU times: user 444 µs, sys: 1 µs, total: 445 µs
Wall time: 449 µs
```

Lazy Evaluation

```
npOnes = np.ones((10000,10000))
%time np.exp(npOnes)[1:100,1:100]

daskOnes = da.ones((10000,10000), chunks=(100))
%time da.exp(daskOnes)[1:100,1:100]
```

Lazy Evaluation

```
npOnes = np.ones((10000,10000))
%time np.exp(npOnes)[1:100,1:100]

daskOnes = da.ones((10000,10000), chunks=(100))
%time da.exp(daskOnes)[1:100,1:100]
```

Result: 10× speedup. (970ms → 90ms)

Ghosting

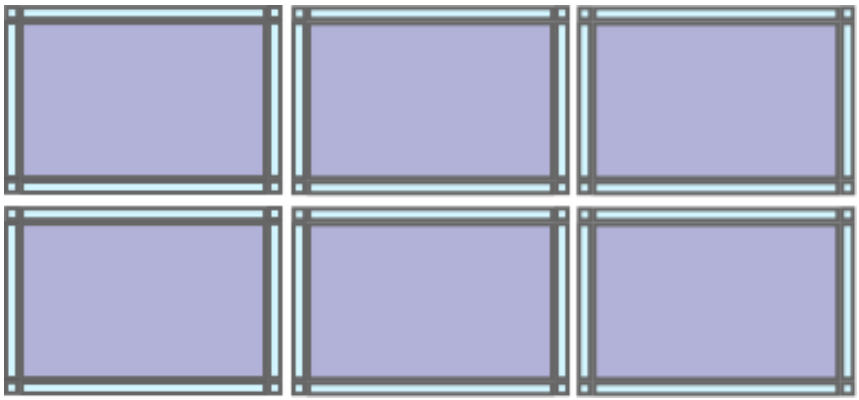
Some operations require communication of borders with neighboring blocks (i.e., functions evaluate with respect to neighbors).

Ghosting

Some operations require communication of borders with neighboring blocks (i.e., functions evaluate with respect to neighbors).
Examples: Evaluating a partial derivative, sliding max.

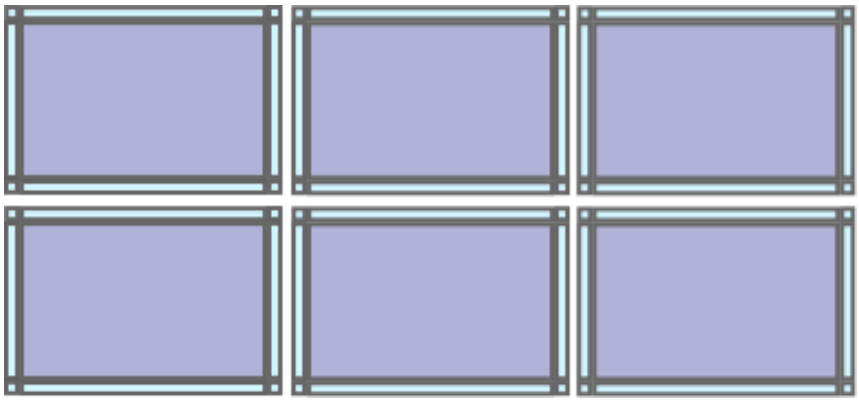
Ghosting

Some operations require communication of borders with neighboring blocks (i.e., functions evaluate with respect to neighbors).
Examples: Evaluating a partial derivative, sliding max.
Solution: Ghosting. Expands each blocks by neighbor's borders.



Ghosting

Some operations require communication of borders with neighboring blocks (i.e., functions evaluate with respect to neighbors).
Examples: Evaluating a partial derivative, sliding max.
Solution: Ghosting. Expands each blocks by neighbor's borders.



Can specify boundary conditions in {periodic, reflect, length}.
More on this in notebook.

Internals: Dictionary

```
import itertools slice =
itertools.islice(smallChunks.dask.items(), 0, 1) for
key, value in slice:  print(key,value)
```

Internals: Dictionary

```
import itertools slice =
itertools.islice(smallChunks.dask.items(), 0, 1) for
key, value in slice:  print(key,value)
```

Result:

Key: (from-array-8c463a3b962efd2e81b47e1832acfb81', 83, 94)
Value: (<function getarray at 0x1083260d0>,
'from-array-8c463a3b962efd2e81b47e1832acfb81', (slice(8300, 8400, None), slice(9400, 9500, None))

Internals: Spilling to Disk

Sometimes, computations will exceed memory.

Internals: Spilling to Disk

Sometimes, computations will exceed memory.
Solution A: Get more memory (hard).

Internals: Spilling to Disk

Sometimes, computations will exceed memory.
Solution A: Get more memory (hard).
Solution B: Use `Chest()`.
Creates dictionary-like object which pickles and spills to disk.
Data retrieved when needed.

Internals: Spilling to Disk

Sometimes, computations will exceed memory.
Solution A: Get more memory (hard).
Solution B: Use `Chest()`.
Creates dictionary-like object which pickles and spills to disk.
Data retrieved when needed.
Syntax: `cache = Chest()`, `cache=Chest(path,memBound)`

API

Subset of NumPy API.

API

Subset of NumPy API.

Good things: Lots of linear algebra (e.g., Cholesky, inverse, LU, QR, SVD), random arrays, fast arithmetic.

API

Subset of NumPy API.

Good things: Lots of linear algebra (e.g., Cholesky, inverse, LU, QR, SVD), random arrays, fast arithmetic.

Bad things: No sorting, or value-dependent shapes (i.e., indexing one array with another). Kind of like Julia.

API

Subset of NumPy API.

Good things: Lots of linear algebra (e.g., Cholesky, inverse, LU, QR, SVD), random arrays, fast arithmetic.

Bad things: No sorting, or value-dependent shapes (i.e., indexing one array with another). Kind of like Julia.

Takeaway: Use if you can.

Reference: dir(dask.array)

Motivation

Problem: encode dependency graph from previously, as a composition of Python data structures.

Motivation

Problem: encode dependency graph from previously, as a composition of Python data structures.
Dask Answer: Dict!

Motivation

Problem: encode dependency graph from previously, as a composition of Python data structures.
Dask Answer: Dict!
Recall Example: `println(2+(2+3))`

Motivation

Problem: encode dependency graph from previously, as a composition of Python data structures.
Dask Answer: Dict!
Recall Example: `println(2+(2+3))`
Dask Graph: {'y':(add,2,x),'x':(2,3)}
Note: Constructor doesn't reference Dask.

Some Lingo

Graph: Such a dictionary.
Key: Hashable values, that aren't tasks. (i.e., pieces of data).
Tasks: Tuple, with a callable first element (something which needs to be computed).
Arguments: Non-function elements of a task. Could be other keys, literals, tasks, or lists thereof.

A Closer Look at Tasks

How does this work?

A Closer Look at Tasks

How does this work?
In Python, like in Julia, functions are **first-class objects.**
The tuple $(f, arg1, arg2, ...)$ is also an object, and its data can be inspected by the interpreter.

A Closer Look at Tasks

How does this work?
In Python, like in Julia, functions are **first-class objects.**
The tuple $(f, arg1, arg2, ...)$ is also an object, and its data can be inspected by the interpreter.
LISP users will (apparently) recognize this as an s-expression.
Quick aside: Functional programming. `https://xkcd.com/297/`.

Graph Methods

`get`: Like Julian `fetch`. Has the scheduler go in and actually retrieve the value — the backend can be arbtrarily complex (i.e, parallel).
Actually a scheduler method (i.e. `dask.threaded.get`)
`compute`: For a collection, outputs the result of a computation (i.e., bypasses scheduler `get`:)

```
timeSum = timeData.sum().compute()
```

Graph Optimization

Recall: Dask will run an **entire graph**, regardless of whether you only want some input.
Easy optimizations can greatly improve performance.

Graph Optimization

Recall: Dask will run an **entire graph**, regardless of whether you only want some input.
Easy optimizations can greatly improve performance.
Two general kinds of optimization:

- Simplify computation.

Graph Optimization

Recall: Dask will run an **entire graph**, regardless of whether you only want some input.
Easy optimizations can greatly improve performance.
Two general kinds of optimization:

- Simplify computation.
- Improve parallelism.

Graph Optimization

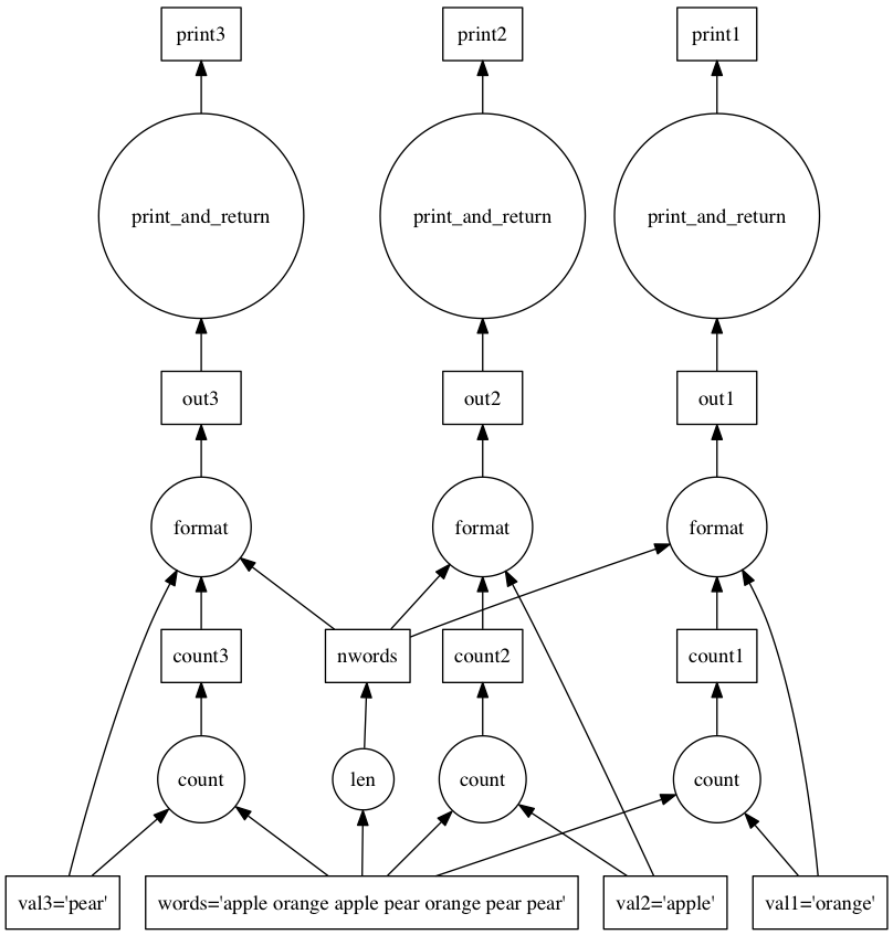
Goal: `results = get(dsk,['print1','print2'])`

What shape makes a graph optimal?

Graph Optimization

Goal: `results = get(dsk,['print1','print2'])`

What shape makes a graph optimal?

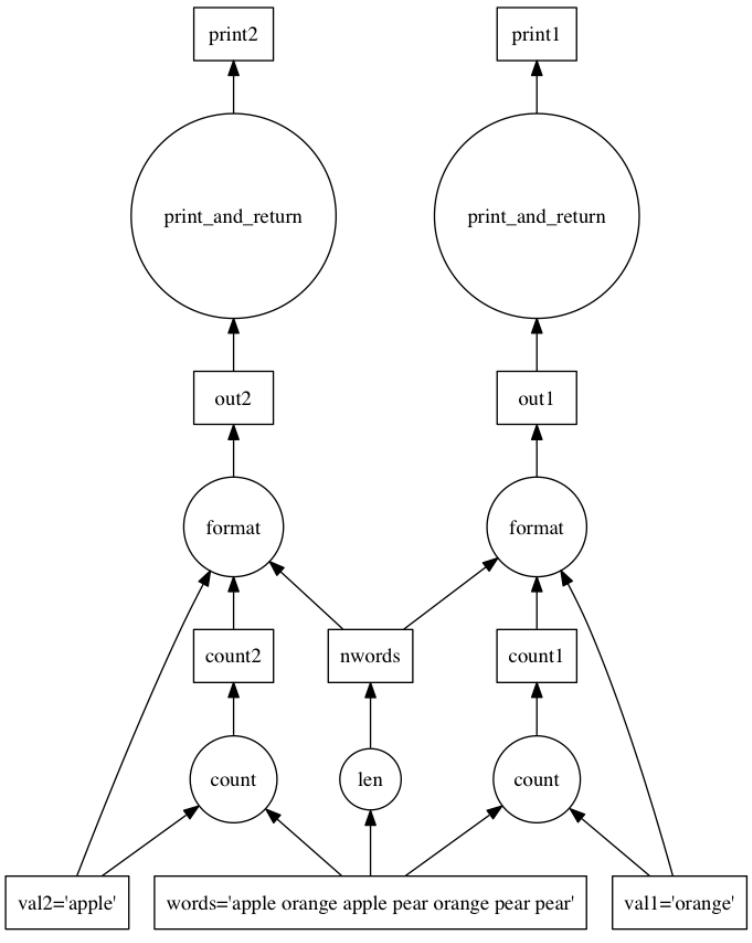


Graph Optimization - Cull

```
from dask.optimize import cull;
results=get(cull(dsk,args),args...)
```

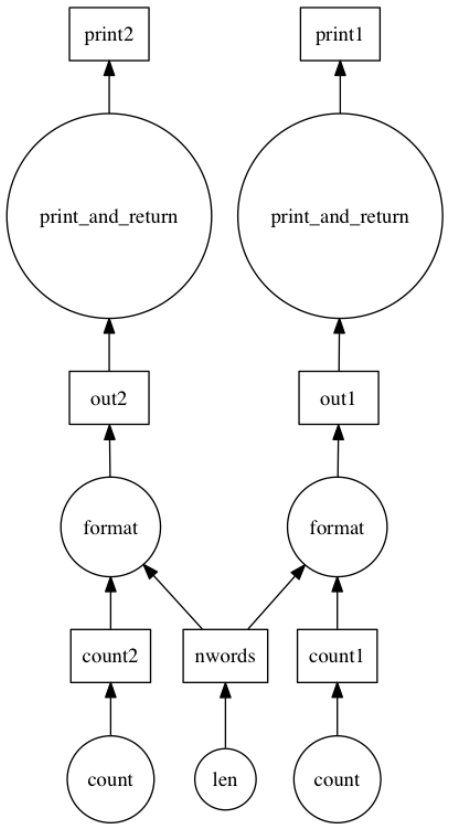
Graph Optimization - Cull

```
from dask.optimize import cull;
results=get(cull(dsk,args),args...)
```

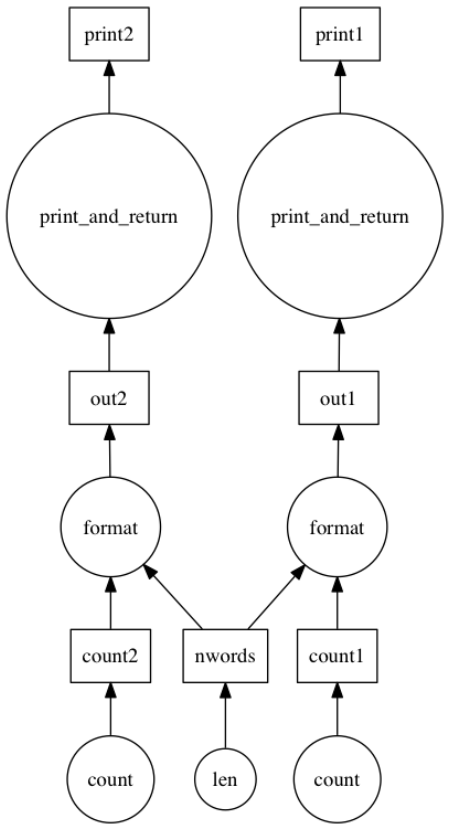


Graph Optimization - Inlining Constants

Graph Optimization - Inlining Constants



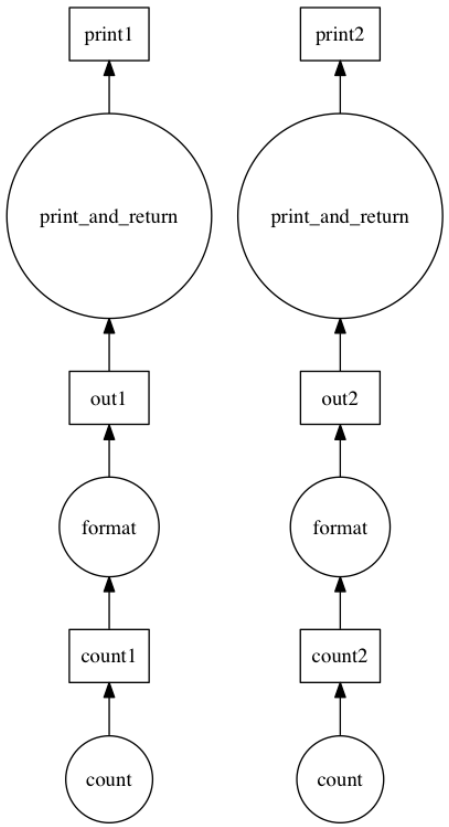
Graph Optimization - Inlining Constants



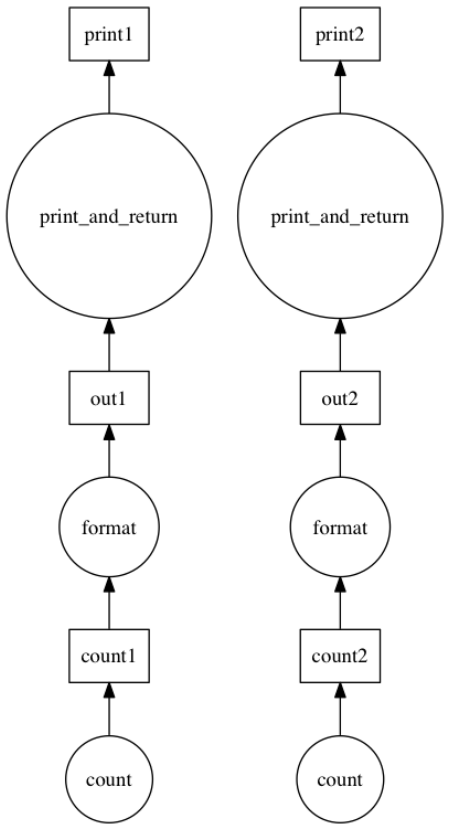
Code: dsk2 = inline(dsk1)

Graph Optimization - Inlining Functions

Graph Optimization - Inlining Functions



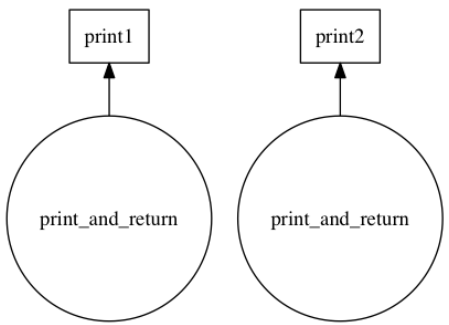
Graph Optimization - Inlining Functions



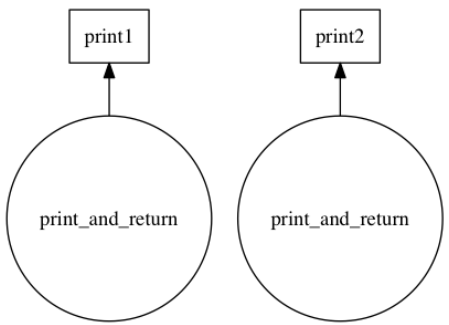
Code: $dsk3 = inline_functions(dsk2, [len, str.split])$

Graph Optimization - Reduce Serialization

Graph Optimization - Reduce Serialization



Graph Optimization - Reduce Serialization



Code: `dsk4 = fuse(dsk3)`

Dask Schedulers

- *dask.threaded.get* — a scheduler which delegates tasks to a bunch of threads

Dask Schedulers

- *dask.threaded.get* — a scheduler which delegates tasks to a bunch of threads
- *dask.multi_processing.get* — a scheduler which delegates tasks to a bunch of processes

Dask Schedulers

- *dask.threaded.get* — a scheduler which delegates tasks to a bunch of threads
- *dask.multi_processing.get* — a scheduler which delegates tasks to a bunch of processes
- *dask.async.get_sync* — a synchronous scheduler, useful for debugging
- *distributed.Executor.get* — a scheduler for executing distributed projects (i.e., across multiple machines). This is a separate GitHub repo (distributed).

How Scheduling Works

I/O-bound code vs. CPU-bound code.

How Scheduling Works

I/O-bound code vs. CPU-bound code.
Dask Tiebreakers:

- Tiebreaker 1: Tasks placed on a stack, so LIFO ordering.

How Scheduling Works

I/O-bound code vs. CPU-bound code.
Dask Tiebreakers:

- Tiebreaker 1: Tasks placed on a stack, so LIFO ordering.
- Tiebreaker 2 (deciding between tasks to place on stack): Use the ordering of tasks induced by a depth-first search on the graph.

How Scheduling Works

I/O-bound code vs. CPU-bound code.
Dask Tiebreakers:

- Tiebreaker 1: Tasks placed on a stack, so LIFO ordering.
- Tiebreaker 2 (deciding between tasks to place on stack): Use the ordering of tasks induced by a depth-first search on the graph.
- Tiebreaker 3 (how to choose between children): Prioritize those children upon whom the most data depends.

Overhead and Scaling — Message Passing

Message passing is **slow**.

Overhead and Scaling — Message Passing

Message passing is **slow**. Induces a natural constraint — bigger, more meaningful tasks.

Overhead and Scaling — Message Passing

Message passing is **slow**. Induces a natural constraint — bigger, more meaningful tasks.

Key performance takeaways for Dask (threaded):

- 1 ms overhead per task
- 100 ms startup time
- Linear scaling with number of dependencies per task
- Constant scaling with number of tasks

Overhead and Scaling — Message Passing

Message passing is **slow**. Induces a natural constraint — bigger, more meaningful tasks.
Key performance takeaways for Dask (threaded):

- 1 ms overhead per task
- 100 ms startup time
- Linear scaling with number of dependencies per task
- Constant scaling with number of tasks

This should be surprising! Remember, number of edges grows as follows: $\binom{n}{2} p$

Bag Specification

Motivation: Implements parallel operations, like `map`, `frequencies`, on groups of objects.

Bag Specification

Motivation: Implements parallel operations, like `map`, `frequencies`, on groups of objects.
Spec: Unordered collection, allows repeats.

Bag Specification

Motivation: Implements parallel operations, like `map`, `frequencies`, on groups of objects.
Spec: Unordered collection, allows repeats.
Contrasts with:
`Set`: Unordered collection, no repeats.
`List`: Ordered collection, has repeats.

Bags Objects

Constructor: Like `Array`. `npartitions` argument analogous to chunks.

Bags Objects

Constructor: Like `Array`. `npartitions` argument analogous to chunks.

Do not load data in serial, then call constructor. Calling constructor on a file parallelizes loading step, reduces overhead.

Bags Objects

Constructor: Like `Array`. `npartitions` argument analogous to chunks.

Do not load data in serial, then call constructor. Calling constructor on a file parallelizes loading step, reduces overhead.

Writable to text, to DataFrame.

Bags API

Set operations: `Any`, `All`, `Max`, `Mean`, `etc.`

Bags API

Set operations: `Any`, `All`, `Max`, `Mean`, `etc.`

`Fold`: Parallel `reduce`. Takes one function to reduce partitions, and another to combine results across partitions.

Bags API

Set operations: `Any`, `All`, `Max`, `Mean`, `etc.`
`Fold`: Parallel `reduce`. Takes one function to reduce partitions, and another to combine results across partitions.
Other Functions: `topk`*, Cartesian product between two bags.

Shuffling

Some operations require much inter-worker communication.

Shuffling

Some operations require much inter-worker communication. Dask handles this by creating a special memory server as a focal point.

Shuffling

Some operations require much inter-worker communication.
Dask handles this by creating a special memory server as a focal point.
Expensive. Recommended workflow is to clean using Bags, and then perform ordered operations using `DataFrame` or `Array`.
Dask Implementation: https://github.com/dask/partd

Known Limitations

- Uses multiprocessing scheduler, not thread-backed one.

Known Limitations

- Uses multiprocessing scheduler, not thread-backed one.
- GroupBy is slow.

Known Limitations

- Uses multiprocessing scheduler, not thread-backed one.
- GroupBy is slow.
- Bag operations : Array operations :: Python operations: NumPy operations.

General Overview

Is a stack of Pandas DataFrames, like Arrays were a grid of NumPy arrays (why stack?)

General Overview

Is a stack of Pandas DataFrames, like Arrays were a grid of NumPy arrays (why stack?)
Implements small but commonly-used subset of Pandas API.

General Overview

Is a stack of Pandas DataFrames, like Arrays were a grid of NumPy arrays (why stack?)
Implements small but commonly-used subset of Pandas API.
Natural hierarchy of problems, in terms of parallelizability.

Known Limitations

Uses thread-backed scheduler by default. Pandas is more fond of the GIL than NumPy, so speedups may be slower across cores. Consider switching to another scheduler.

Known Limitations

Uses thread-backed scheduler by default. Pandas is more fond of the GIL than NumPy, so speedups may be slower across cores. Consider switching to another scheduler.

Setting a new index by an unsorted column is expensive (why?)

Known Limitations

Uses thread-backed scheduler by default. Pandas is more fond of the GIL than NumPy, so speedups may be slower across cores. Consider switching to another scheduler.

Setting a new index by an unsorted column is expensive (why?)

Many operations require setting such an index.