Dask: Parallel Computation with Blocked algorithms and Task Scheduling

Matthew Rocklin[‡][*]

https://www.youtube.com/watch?v=1kkFZ4P-XHg

Abstract—Dask enables parallel and out-of-core computation. We couple blocked algorithms with dynamic and memory aware task scheduling to achieve a parallel and out-of-core NumPy clone. We show how this extends the effective scale of modern hardware to larger datasets and discuss how these ideas can be more broadly applied to other parallel collections.

Index Terms—parallelism, NumPy, scheduling

Introduction

The Scientific Python stack [Oli07] rarely leverages parallel computation. Code built off of NumPy [vdW11] or Pandas [McK10] generally runs in a single thread on data that fits comfortably in memory. Advances in hardware in the last decade in multi-core processors and solid state drives provide significant and yet largely untapped performance advantages.

However, the Scientific Python stack consists of hundreds of software packages, papers, PhD theses, and developer-years. This stack is a significant intellectual and financial investment that, for the most part, does not align well with modern hardware. We seek software solutions to parallelize this software stack without triggering a full rewrite.

This paper introduces `dask`, a specification to encode parallel algorithms, using primitive Python dictionaries, tuples, and callables. We use `dask` to create `dask.array` a parallel N-dimensional array library that copies the NumPy interface, uses all of the cores in a modern processor, and manages data well from disk. Dask.array serves both as a general library for parallel out-of-core `ndarrays` and also as a demonstration that we can parallelize complex codebases like NumPy in a straightforward manner using blocked algorithms and task scheduling.

We first define `dask` graphs and give a trivial example of their use. We then share the design of `dask.array` a parallel `ndarray`. Then we discuss dynamic task scheduling and policies to minimize memory footprint. We then give two examples using `dask.array` on computational problems. We then briefly discuss `dask.bag` and `dask.dataframe`, two other collections in the `dask` library. We finish with thoughts about extension of this approach into the broader Scientific Python ecosystem.

* Corresponding author: mrocklin@gmail.com
‡ Continuum Analytics

Modern Hardware

Hardware has changed significantly in recent years. The average personal notebook computer (the bulwark of most scientific development) has roughly four physical cores and a solid state drive (SSD). The four physical cores present opportunities for linear speedup of computationally bound code. We refer to algorithms that use multiple cores simultaneously as *parallel*. The solid state drives have high read bandwidths and low seek times which enables them to serve as large and cheap extensions of physical memory. We refer to systems that efficiently use disk as extensions of memory as *out-of-core*.

Modern workstations extend these trends to include sixteen to sixty-four cores, hundreds of gigabytes of RAM, and RAID arrays of SSDs offering 2GB/s read bandwidths. These systems rival small clusters in scale but continue to offer the convenience of single-machine administration and shared-memory computing. This system rivals the performance of massively parallel distributed systems up to a surprisingly large scale while maintaining a low maintenance and programming cost.

Dask Graphs

Normally humans write programs and then compilers/interpreters interpret them (e.g. `python`, `javac`, `clang`). Sometimes humans disagree with how these compilers/interpreters choose to interpret and execute their programs. In these cases humans often bring the analysis, optimization, and execution of code into the code itself.

Commonly a desire for parallel execution causes this shift of responsibility from compiler to human developer. In these cases we often represent the structure of our program explicitly as data within the program itself.

Dask is a specification that encodes task schedules with minimal incidental complexity using terms common to all Python projects, namely dicts, tuples, and callables. Ideally this minimum solution is easy to adopt and understand by a broad community.

We define a dask graph as a Python dictionary mapping keys to tasks or values. A key is any Python hashable, a value is any Python object that is not a task, and a task is a Python tuple with a callable first element.

Example

Consider the following simple program

```
def inc(i):
    return i + 1
```

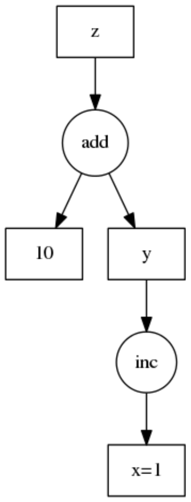

Fig. 1: *A simple dask dictionary*

```
def add(a, b):
    return a + b

x = 1
y = inc(x)
z = add(y, 10)
```

We encode this as a dictionary below:

```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

While less pleasant than our original code this representation can be analyzed and executed by other Python code, not just the CPython interpreter. We don't recommend that users write code in this way, but rather that it is an appropriate target for automated systems. Also, in non-toy examples the execution times are likely much larger than for `inc` and `add`, warranting the extra complexity.

Specification

We represent a computation as a directed acyclic graph of tasks with data dependencies. Dask is a specification to encode such a graph using ordinary Python data structures, namely dicts, tuples, functions, and arbitrary Python values.

A **dask graph** is a dictionary mapping identifying keys to values or tasks. We explain these terms after showing a complete example:

```
{'x': 1,
 'y': 2,
 'z': (add, 'x', 'y'),
 'w': (sum, ['x', 'y', 'z'])}
```

A **key** can be any hashable value that is not a task.

```
'x'
('x', 2, 3)
```

A **task** is a tuple with a callable first element. Tasks represent atomic units of work meant to be run by a single worker.

```
(add, 'x', 'y')
```

We represent a task as a tuple such that the *first element is a callable function* (like `add`), and the succeeding elements are *arguments* for that function.

An **argument** may be one of the following:

1) Any key present in the dask like `'x'`
2) Any other value like `1`, to be interpreted literally
3) Other tasks like `(inc, 'x')`
4) List of arguments, like `[1, 'x', (inc, 'x')]`

So all of the following are valid tasks

```
(add, 1, 2)
(add, 'x', 2)
(add, (inc, 'x'), 2)
(sum, [1, 2])
(sum, ['x', (inc, 'x')])
(np.dot, np.array([...]), np.array([...]))
```

The dask spec provides no explicit support for keyword arguments. In practice we combine these into the callable function with `functools.partial` or `toolz.curry`.

Dask Arrays

The `dask.array` submodule uses dask graphs to create a NumPy-like library that uses all of your cores and operates on datasets that do not fit in memory. It does this by building up a dask graph of blocked array algorithms.

The `dask.array` submodule is not the first library to implement a "Big NumPy Clone". Other partial implementations exist including Biggus an out-of-core `ndarray` specialized for climate science, Spartan [Pow14] a distributed memory `ndarray`, and Distarray a distributed memory `ndarray` that interacts well with other distributed array libraries like Trillinos. There have also been numerous projects in traditional high performance computing space including Elemental [Pou13], High Performance Fortran, etc.. Finally Theano [Ber10], an array compiler in Python with powerful optimizations and GPU support, statically schedules and reasons about array computations and has proven particularly valuable in machine learning applications.

Each of these implementations focuses on a particular application or problem domain. Dask.array distinguishes itself in that it focuses on a very general class of NumPy operations and streaming execution through dynamic task scheduling.

Blocked Array Algorithms

Blocked algorithms compute a large result like "take the sum of these trillion numbers" with many small computations like "break up the trillion numbers into one million chunks of size one million, sum each chunk, then sum all of the intermediate sums." Through tricks like this we can evaluate one large problem by solving very many small problems.

Blocked algorithms have proven useful in modern numerical linear algebra libraries like Flame [Gei08] and Plasma [Agu09] and more recently in data parallel systems like Dryad [Isa07] and Spark [Zah10]. These compute macroscopic operations with a collection of related in-memory operations.

Dask.array takes a similar approach to linear algebra libraries but focuses instead on the more pedestrian `ndarray` operations, like arithmetic, reductions, and slicing common in interactive use.

Example: `arange`

Dask array functions produce `Array` objects that hold on to dask graphs. These dask graphs use several `numpy` functions to achieve the full result. In the following example one call to `da.arange` creates a graph with three calls to `np.arange`

```
>>> import dask.array as da
>>> x = da.arange(15, chunks=(5,))
```

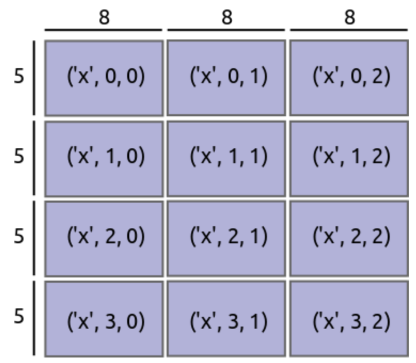

Fig. 2: *A dask array*

```
>>> x       # Array object metadata
dask.array<x-1, shape=(15,), chunks=((5, 5, 5)), dtype=int64>
>>> x.dask  # Every dask array holds a dask graph
{('x', 0): (np.arange, 0, 5),
 ('x', 1): (np.arange, 5, 10),
 ('x', 2): (np.arange, 10, 15)}
```

Further operations on `x` create more complex graphs

```
>>> z = (x + 100).sum()
>>> z.dask
{('x', 0): (np.arange, 0, 5),
 ('x', 1): (np.arange, 5, 10),
 ('x', 2): (np.arange, 10, 15),
 ('y', 0): (add, ('x', 0), 100),
 ('y', 1): (add, ('x', 1), 100),
 ('y', 2): (add, ('x', 2), 100),
 ('z', 0): (np.sum, ('y', 0)),
 ('z', 1): (np.sum, ('y', 1)),
 ('z', 2): (np.sum, ('y', 2)),
 ('z',): (sum, [('z', 0), ('z', 1), ('z', 2)])}
```

Dask.array also holds convenience functions to execute this graph, completing the illusion of a NumPy clone

```
>>> z.compute()
1605
```

Array metadata

In the example above `x` and `z` are both `dask.array.Array` objects. These objects contain the following data

1) A dask graph, `.dask`
2) Information about shape and chunk shape, called `.chunks`
3) A name identifying which keys in the graph correspond to the result, `.name`
4) A dtype

The second item here, `chunks`, deserves further explanation. A normal NumPy array knows its `shape`, a dask array must know its shape and the shape of all of the internal NumPy blocks that make up the larger array. These shapes can be concisely described by a tuple of tuples of integers, where each internal tuple corresponds to the lengths along a single dimension.

In the example above we have a 20 by 24 array cut into uniform blocks of size 5 by 8. The `chunks` attribute describing this array is the following:

```
chunks = ((5, 5, 5, 5), (8, 8, 8))
```

Where the four fives correspond to the heights of the blocks along the first dimension and the three eights correspond to the widths of the blocks along the second dimension. This particular example has uniform sizes along each dimension but this need not be the case. Consider the chunks of the following example operations

```
>>> x[::2].chunks
((3, 2, 3, 2), (8, 8, 8))

>>> x[::2].T.chunks
((8, 8, 8), (3, 2, 3, 2))
```

Every `dask.array` operation, like `add`, slicing, or `transpose` must take the graph and all metadata, add new tasks into the graph and determine new values for each piece of metadata.

Capabilities and Limitations

Adding subgraphs and managing metadata for most of NumPy is difficult but straightforward. At present `dask.array` is around 5000 lines of code (including about half comments and doc-strings). It encompasses most commonly used operations including the following:

- Arithmetic and scalar mathematics, `+, *, exp, log, ...`
- Reductions along axes, `sum(), mean(), std(), sum(axis=0), ...`
- Tensor contractions / dot products / matrix multiply, `tensordot`
- Axis reordering / transpose, `transpose`
- Slicing, `x[:100, 500:100:-2]`
- Fancy indexing along single axes with lists or NumPy arrays, `x[:, [10, 1, 5]]`
- A variety of utility functions, `bincount, where, ...`

However `dask.array` is unable to handle any operation whose shape can not be determined ahead of time. Consider for example the following common NumPy operation

```
x[x > 0]  # can not determine shape of output
```

The shape of this array depends on the number of positive elements in `x`. This shape is not known given only metadata; it requires knowledge of the values underlying `x`, which are not available at graph creation time. Note however that this case is fairly rare; for example it is possible to determine the shape of the output in all other cases of slicing and indexing, e.g.

```
x[10::3, [1, 2, 5]]  # can determine shape of output
```

Dynamic Task Scheduling

We now discuss how `dask` executes task graphs. How we execute these graphs strongly impacts performance. Fortunately we can tackle this problem with a variety of approaches without touching the graph creation problem discussed above. Graph creation and graph execution are separable problems. The dask library contains schedulers for single-threaded, multi-threaded, multi-process, and distributed execution.

Current dask schedulers all operate *dynamically*, meaning that execution order is determined during execution rather than ahead of time through static analysis. This is good when runtimes are not known ahead of time or when the execution environment contains uncertainty. However dynamic scheduling does preclude certain clever optimizations.

Dynamic task scheduling has a rich literature and numerous projects, both within the Python ecosystem with projects like

Spotify's Luigi for bulk data processing and projects without the ecosystem like DAGuE [Bos12] for more high performance task scheduling. Additionally, data parallel systems like Dryad or Spark contain their own custom dynamic task schedulers.

None of these solutions, nor much of the literature in dynamic task scheduling, suited the needs of blocked algorithms for shared memory computation. We needed a lightweight, easily installable Python solution that had latencies in the millisecond range and was mindful of memory use. Traditional task scheduling literature usually focuses on policies to expose parallelism or chip away at the critical path. We find that for bulk data analytics these are not very relevant as parallelism is abundant and critical paths are comparatively short relative to the depth of the graph.

The logic behind dask's schedulers reduces to the following situation: A worker reports that it has completed a task and that it is ready for another. We update runtime state to record the finished task, mark which new tasks can be run, which data can be released, etc.. We then choose a task to give to this worker from among the set of ready-to-run tasks. This small choice governs the macro-scale performance of the scheduler.

Instead of these metrics found in the literature we find that for out-of-core computation we need to choose tasks that allow us to release intermediate results and keep a small memory footprint. This lets us avoid spilling intermediate values to disk which hampers performance significantly. After several other policies we find that the policy of *last in, first out* is surprisingly effective. That is we select tasks whose data dependencies were most recently made available. This causes a behavior where long chains of related tasks trigger each other, forcing the scheduler to finish related tasks before starting new ones. We implement this with a simple stack, which can operate in constant time.

We endeavor to keep scheduling overhead low at around 1ms per task. Updating executing state and deciding which task to run must be made very quickly. To do this we maintain a great deal of state about the currently executing computation. The set of ready-to-run tasks is commonly quite large, in the tens or hundreds of thousands in common workloads and so in practice we must maintain enough state so that we can choose the right task in constant time (or at least far sub-linear time).

Finally, power users can disregard the dask schedulers and create their own. Dask graphs are completely separate from the choice of scheduler and users may select the right scheduler for their class of problem or, if no ideal scheduler exists, build one anew. The default single-machine scheduler is about three hundred significant lines of code and has been adapted to single-threaded, multi-threaded, multi-processing, and distributed computing variants.

Example: Matrix Multiply

We benchmark dask's blocked matrix multiply on an out-of-core dataset. This demonstrates the following:

1) How to interact with on-disk data
2) The blocked algorithms in dask.array achieve similar performance to modern BLAS implementations on compute-bound tasks

We set up a trivial input dataset

```
import h5py
f = h5py.File('myfile.hdf5')
A = f.create_dataset(name='/A',
          shape=(200000, 4000), dtype='f8',
          chunks=(250, 250), fillvalue=1.0)
B = f.create_dataset(name='/B',
          shape=(4000, 4000), dtype='f8',
          chunks=(250, 250), fillvalue=1.0)
out = f.create_dataset(name='/out',
          shape=(4000, 4000), dtype='f8',
          chunks=(250, 250))
```

Performance (GFLOPS)	NumPy	Dask.array
ATLAS BLAS	6	18
OpenBLAS (one)	11	23
OpenBLAS (four)	22	11

TABLE 1: *Matrix Multiply GigaFLOPS for NumPy/Dask.array and for ATLAS and OpenBLAS with one and four threads*

The Dask convenience method, `da.from_array`, creates a graph that can pull data from any object that implements NumPy slicing syntax. The `da.store` function can then store a large result in any object that implements NumPy setitem syntax.

```
import dask.array as da
a = da.from_array(A, chunks=(1000, 1000))
b = da.from_array(B, chunks=(1000, 1000))

c = a.dot(b)  # another dask Array, not yet computed
c.store(out)  # Store result into output space
```

Results: We do this same operation in different settings.

We use either use NumPy or `dask.array`:

1) Use NumPy on a big-memory machine
2) Use dask.array in a small amount of memory, pulling data from disk, using four threads

We compare different BLAS implementations:

1) ATLAS BLAS, single threaded, unblocked
2) OpenBLAS, single threaded
3) OpenBLAS, multi-threaded

For each configuration we compute the number of floating point operations per second.

We note the following

1) Compute-bound tasks are computationally bound by memory; we don't experience a slowdown
2) Dask.array can effectively parallelize and block ATLAS BLAS for matrix multiplies
3) Dask.array doesn't significantly improve when using an optimized BLAS, presumably this is because we've already reaped most of the benefits of blocking and multi-core
4) One should not mix multiple forms of multi-threading. Four dask.array threads each spawning multi-threaded OpenBLAS DGEMM calls results in worse performance.

Example: Meteorology

Performance is secondary to capability. In this example we use `dask.array` to manipulate climate datasets that are larger than memory. This example shows the following:

1) Use `concatenate` and `stack` to manage large piles of HDF5 files (a common case)
2) Use reductions and slicing to manipulate stacks of arrays

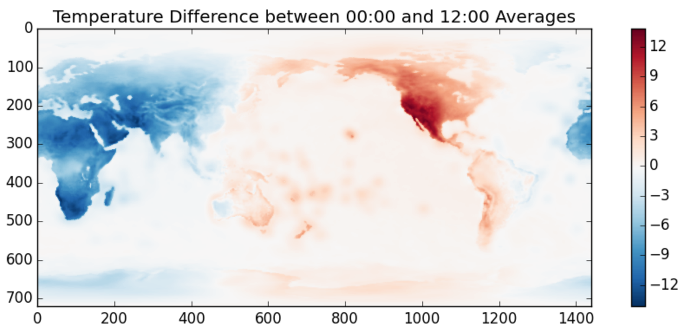

***Fig. 3:** We use typical NumPy slicing and reductions on a large volume of data to show the average temperature difference between noon and midnight for year 2014*

3) Interact with other libraries in the ecosystem using the `__array__` protocol.

We start with a typical setup, a large pile of NetCDF files.:

```
$ ls
2014-01-01.nc3  2014-03-18.nc3  2014-06-02.nc3
2014-01-02.nc3  2014-03-19.nc3  2014-06-03.nc3
2014-01-03.nc3  2014-03-20.nc3  2014-06-04.nc3
2014-01-04.nc3  2014-03-21.nc3  2014-06-05.nc3
...             ...             ...
```

Each of these files contains the temperature at two meters above ground over the earth at quarter degree resolution, every six hours.

```
>>> from netCDF4 import netCDF4
>>> t = Dataset('2014-01-01.nc3').variables['t2m']
>>> t.shape
(4, 721, 1440)
```

We can collect many of these files together using `da.concatenate`, resulting in a single large array.

```
>>> from glob import glob
>>> filenames = sorted(glob('2014-*.nc3'))
>>> temps = [Dataset(fn).variables['t2m']
...          for fn in filenames]

>>> import dask.array as da
>>> arrays = [da.from_array(t, blockshape=(4,200,200))
...           for t in temps]
>>> x = da.concatenate(arrays, axis=0)

>>> x.shape
(1464, 721, 1440)
```

We can now play with this array as though it were a NumPy array. Because dask.arrays implement the `__array__` protocol we can dump them directly into functions of other libraries. These libraries will trigger computation when they call `np.array(...)` on their input.

```
>>> from matplotlib import imshow
>>> imshow(x[::4].mean(axis=0) - x[2::4].mean(axis=0)
...        , cmap='RdBu_r')
```

This computation took about a minute on an old notebook computer. It was bound by disk access. Meteorological cases tend to be I/O bound rather than compute bound, taking more advantage of `dask`'s memory-aware schedulers rather than parallel computation. In other cases, such as parallel image processing, this trend is reversed.

Other Collections

The dask library contains parallel collections other than `dask.array`. We briefly describe `dask.bag` and `dask.dataframe`

- `dask.array` = `numpy` + `threading`
- `dask.bag` = `toolz` + `multiprocessing`
- `dask.dataframe` = `pandas` + `threading`

Bag

A *bag* is an unordered collection with repeats. It is like a Python list but does not guarantee the order of elements. Because we typically compute on Python objects in `dask.bag` we are bound by the Global Interpreter Lock and so switch from using a multi-threaded scheduler to a multi-processing one.

The `dask.bag` API contains functions like `map` and `filter` and generally follows the PyToolz API. We find that it is particularly useful on the front lines of data analysis, particularly in parsing and cleaning up initial data dumps like JSON or log files because it combines the streaming properties and solid performance of projects like `cytoolz` with the parallelism of multiple processes.

```
>>> import dask.bag as db
>>> import json
>>> b = db.from_filenames('2014-*.json.gz')
...       .map(json.loads)

>>> alices = b.filter(lambda d: d['name'] == 'Alice')
>>> alices.take(3)
({'name': 'Alice', 'city': 'LA',  'balance': 100},
 {'name': 'Alice', 'city': 'LA',  'balance': 200},
 {'name': 'Alice', 'city': 'NYC', 'balance': 300},

>>> dict(alices.pluck('city').frequencies())
{'LA': 10000, 'NYC': 20000, ...}
```

DataFrame

The `dask.dataframe` module implements a large dataframe out of many Pandas DataFrames. The interface should be familiar to users of Pandas.

```
>>> import dask.dataframe as dd
>>> df = dd.read_csv('nyc-taxi-*.csv.gz')

>>> g = df.groupby('medallion')
>>> g.trip_time_in_secs.mean().head(5)
medallion
0531373C01FD1416769E34F5525B54C8     795.875026
867D18559D9D2941173AD7A0F3B33E77     924.187954
BD34A40EDD5DC5368B0501F704E952E7     717.966875
5A47679B2C90EA16E47F772B9823CE51     763.005149
89CE71B8514E7674F1C662296809DDF6     869.274052
Name: trip_time_in_secs, dtype: float64
```

Currently `dask.dataframe` uses the threaded scheduler but does not achieve the same parallel performance as `dask.array` due to the GIL. We are enthusiastic about ongoing work in Pandas itself to release the GIL.

The dask dataframe can compute efficiently on *partitioned* datasets where the different blocks are well separated along an index. For example in time series data we may know that all of January is in one block while all of February is in another. Join, groupby, and range queries along this index are significantly faster when working on partitioned datasets.

Dask.dataframe benefits users by providing trivial access to larger-than-memory datasets and, where Pandas does release the GIL, parallel computation.

Dask for General Computing

The higher level collections `dask.array/bag/dataframe` demonstrate the flexibility of the dask graph specification to encode sophisticated parallel algorithms and the capability of the dask schedulers to execute those graphs intelligently on a multi-core machine. Opportunities for parallel execution extend beyond beyond `ndarrays` and dataframes.

In the beginning of this document we gave the following toy example to help define dask graphs.

```
d = {'x': 1,
     'y': (inc, 'x'),
     'z': (add, 'y', 10)}
```

While this example of dask graphs is trivial it represents a broader class of free-form computations that don't fit neatly into a single high-level abstraction like arrays or dataframes but are instead just a bunch of related Python functions with data dependencies. In this context Dask offers a lightweight spec and range of schedulers as well as excellent error reporting and diagnostic facilities. In private projects we have seen great utility and performance from using the dask threaded scheduler to refactor and execute existing processing pipelines on large multi-core computers.

Low Barrier to Entry

The simplicity of dask graphs (no classes or frameworks) presents a very low barrier to entry. Users only need to understand basic concepts common to Python (or indeed most modern languages) like dictionaries, tuples, and functions as variables. As an example consider the work in [Tep15] in which the authors implement out-of-core parallel non-negative matrix factorizations on top of dask.array without significant input from dask core developers. This demonstrates that algorithmic domain experts can implement complex algorithms with dask and achieve good results with a minimum of framework investment.

To demonstrate complexity we present the graph of an out-of-core singular value decomposition contributed by those authors to the `dask.array.linalg` library.

```
>>> import dask.array as da
>>> x = da.ones((5000, 1000), chunks=(1000, 1000))
>>> u, s, v = da.svd(x)
```

This algorithm is complex enough without having to worry about software frameworks. Mathematical experts were able to implement this without having to simultaneously develop expertise in a complex parallel programming framework.

Final Thoughts

Extend the Scale of Convenient Data: The dask collections (`array`, `bag`, `dataframe`) provide reasonable access to parallelism and out-of-core execution. These significantly extend the scale of data that is convenient to manipulate.

Low Barrier to Entry: More importantly these collections demonstrate the feasibility of dask graphs to describe parallel algorithms and of the dask schedulers to execute those algorithms efficiently in a small space. The lack of a more baroque framework drastically reduces the barrier to entry and the ability of developers to use dask within their own libraries.

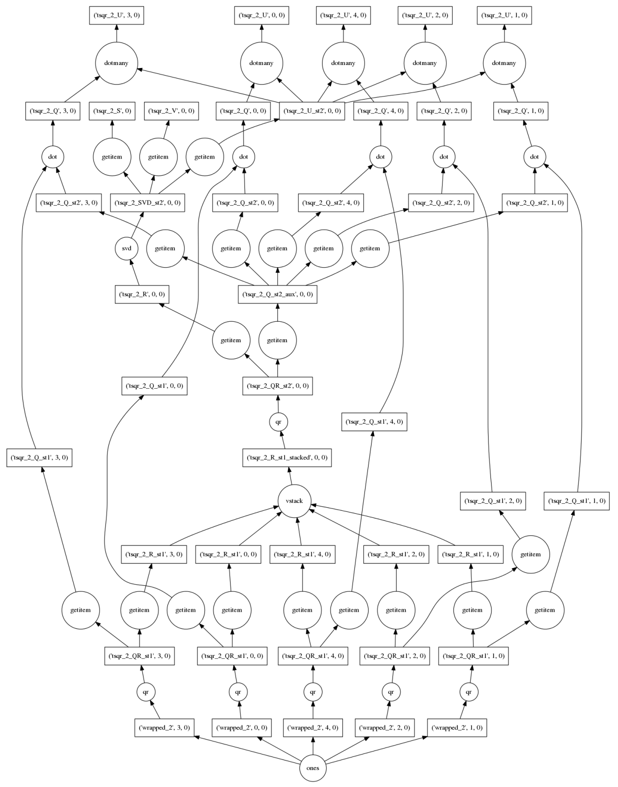

Fig. 4: *Out-of-core parallel SVD*

Administrativa and Links

Dask is available on github, PyPI, and is now included in the Anaconda distribution. It is BSD licensed, runs on Python 2.6 to 3.4 and is tested against Linux, OSX, and Windows.

This document was compiled from numerous blogposts that chronicle dask's development and go more deeply into the computational concerns encountered during dask's construction.

Dask is used on a daily basis, both as a dependency in other projects in the SciPy ecosystem (xray, scikit-image, ...) and also in production in private business.

- http://dask.pydata.org/en/latest
- http://github.com/ContinuumIO/dask
- http://matthewrocklin.com/blog
- http://pypi.python.org/pypi/dask/

Acknowledgements

Dask has had several contributors, both in terms of code and in terms of active use and reporting. Some notable contributions follow (roughly ordered by chronological involvement):

- Stephan Hoyer - Patiently used and bug-fixed `dask.array`
- Erik Welch - Implemented many of the graph optimizations
- Mariano Tepper - Implemented the `dask.array.linalg` module
- Wesley Emeneker - Worked on some of slicing
- Peter Steinberg - Worked on some of rechunking
- Jim Crist - Implemented rewrite rule optimizations
- Blake Griffith - Integrated `dask.array` with `scikit-image` and has done a variety of bug-fixing, particularly around `dask.distributed`

- Min Regan-Kelley - Provided guidance around `ZeroMQ` during the construction of `dask.distributed`
- Phillip Cloud - Improved `dask.dataframe`

References

[Oli07] Travis E. Oliphant. Python for Scientific Computing, Computing in Science & Engineering, 9, 10-20 (2007), DOI:10.1109/MCSE.2007.58

[vdW11] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation, Computing in Science & Engineering, 13, 22-30 (2011)

[McK10] Wes McKinney. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[Isa07] Isard, Michael, et al. "Dryad: distributed data-parallel programs from sequential building blocks." ACM SIGOPS Operating Systems Review. Vol. 41. No. 3. ACM, 2007.

[Zah10] Zaharia, Matei, et al. "Spark: cluster computing with working sets." Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. Vol. 10. 2010. APA

[But09] Buttari, Alfredo, et al. "A class of parallel tiled linear algebra algorithms for multicore architectures." Parallel Computing 35.1 (2009): 38-53. APA

[Bos12] Bosilca, George, et al. "DAGuE: A generic distributed DAG engine for high performance computing." Parallel Computing 38.1 (2012): 37-51. APA

[Van08] Van De Geijn, Robert A., and Enrique S. Quintana-Ortí. "The science of programming matrix computations." (2008). APA

[Pou13] Poulson, Jack, et al. "Elemental: A new framework for distributed memory dense matrix computations." ACM Transactions on Mathematical Software (TOMS) 39.2 (2013): 13. APA

[Tep15] Mariano Tepper and Guillermo Sapiro, "Compressed Nonnegative Matrix Factorization is Fast and Accurate", 2015.

[Agu09] Agullo, Emmanuel, et al. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects." Journal of Physics: Conference Series. Vol. 180. No. 1. IOP Publishing, 2009. APA

[Gei08] Van De Geijn, Robert A., and Enrique S. Quintana-Ortí. "The science of programming matrix computations." (2008). APA

[Ber10] Bergstra, James, et al. "Theano: A CPU and GPU math compiler in Python." Proc. 9th Python in Science Conf. 2010. APA

[Pow14] Power, Russell. Abstractions for In-memory Distributed Computation. Diss. New York University, 2014. APA