Contents

On the Daskening of yt

authors: Chris Havlin, Madicken Munk, Kacper Kowalik and Matthew Turk

a SciPy2021 poster.

The yt package provides a powerful suite of analysis and visualization tools for the physical sciences. In addition to expanding yt into a more domain-agnostic tool, a number of ongoing efforts seek to better leverage advancements from across the Python open source ecosystem within yt's infrastructure. In this work, we present our efforts to incorporate Dask within the yt codebase to handle both lazy serial and parallel workflows, improving both the developer and end-user experience. We compare both user experience and performance between yt's existing serial and MPI-parallel processing methods and their "Daskified" counterparts.

note: this jupyter book can be viewed in a browser at chrishavlin.github.io/scipy2021/ or a pdf can be downloaded here.

Overview

The yt package is a cross-domain tool for visualization and analysis of data. While still used primarily within the computational astrophysics community, a number of recent efforts have focused on transforming yt into a more domain agnostic tool. Part of the effort to expand the yt platform includes leveraging recent advancements from across the open source python ecosystem. This presentation focuses on advancements in using Dask within the yt framework to improve both serial and parallel workflows.

While yt already supports lazy serial and MPI-parallel computations, refactoring yt to use Dask has a number of potential benefits. For the yt user, an underlying Dask framework allows for parallel computation with minimal setup and a consistent experience across machines from laptops to HPC environments. It also allows for easier custom parallel workflows, for example by returning Dask arrays rather than MPI iterable objects (while still operating in MPI environments with dask-mpi) from which a user can use more array-like manipulations while still working in parallel. For the developer, a Dask refactor will simplify the codebase and improve the development process, particularly for implementing new parallel algorithms.

In this presentation, we will show our latest efforts to leverage Dask within the yt framework, building off of previous reports on Dask-yt prototyping shown at RHytHM2020 (Leveraging Dask in yt) and the yt-blog (Dask and yt: a pre-YTEP). These reports described a number of separate experimental prototypes including a "Daskified" particle reader, a binned-statistic calculation using yt's already-optimized parallel statistic methods with Dask delayed arrays and unyt arrays with Dask support. In this presentation, we will present a more fully integrated prototype directly within the yt API. We will show comparisons in use and performance between the existing chunk framework and the new Daskified versions for both single-processor serial and parallel computations.

The poster is divided into a number of sections:

- Daskified unyt arrays: demonstrating a daskified version of the arrays underlying yt.
- Daskified reads: a discussion of yt IO and an implementaiton of a daskified particle reader for yt.
- Daskified computations: demonstrating some daskified reductions.
- Dasken(yt): some final thoughts on further Daskening.
- Repository notes: notes on the code used in this poster.

Daskified unyt arrays

Throughout yt, data is stored using unit-aware unyt arrays. A unyt array is a subclass of a standard numpy ndarray wrapped with operations that track units. So a part of the Daskening of yt relies on adding Dask support to unyt arrays (PR 185). As this has potential users beyond yt users, it is worth walking through its usage. We also describe the general approach to implementing Dask support for unyt arrays.

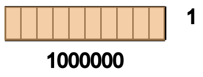
example usage

At present, the primary way to create a `unyt_dask_array` is through the `unyt_from_dask` function, which accepts a standard Dask array and a unit along with all of the optional parameters for `unyt.unyt_array`:

```python
from unyt import dask_array as unyt_dask_array, unyt_quantity, unyt_array
from dask import array as da
import numpy as np
```

```python
x1 = unyt_dask_array.unyt_from_dask(da.random.random((1e6,), chunks=(1e5)), 'm')
x1
```

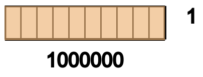
	Array	Chunk
Bytes	8.00 MB	800.00 kB
Shape	(1000000,)	(100000,)
Count	10 Tasks	10 Chunks
Type	float64	numpy.ndarray
Units	m	m



So we’ve created what looks like a Dask array of 10 chunks, but with an extra units attribute. Like a `unyt_array`, we can convert units:

```python
x1.to('cm')
```

	Array	Chunk
Bytes	8.00 MB	800.00 kB
Shape	(1000000,)	(100000,)
Count	20 Tasks	10 Chunks
Type	float64	numpy.ndarray
Units	cm	cm

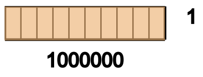


and we can operate on multiple arrays, and any necessary unyt conversions will be applied automatically:

```python
x2 = unyt_dask_array.unyt_from_dask(0.001 * da.random.random((1e6,), chunks=(1e5)),
'km')
```

```python
x = (x1 + x2).to('m')
x
```

	Array	Chunk
Bytes	8.00 MB	800.00 kB
Shape	(1000000,)	(100000,)
Count	60 Tasks	10 Chunks
Type	float64	numpy.ndarray
Units	m	m



but in both these cases, we’re still working with delayed arrays. Once we call `compute`, we’ll get back either a plain `unyt_array` or `unyt_quanity` depending on the operation:

```python
x.mean().compute()
```

```
unyt_quantity(0.99956126, 'm')
```

```python
x[50:60].compute()
```

```
unyt_array([0.895046  , 0.9392074 , 0.98590878, 0.5823761 , 0.26222106,
            1.74925792, 0.41219676, 0.38931168, 0.71809784, 0.44802312], 'm')
```

Design

Designing the unyt_dask_array is an interesting problem. A standard Dask array is what's called a Dask Collection– broken link and it implements a large number of array operations to allow computation over a chunked array. A unyt_array is a direct subclass of a numpy.ndarray with implementations of numpy array protocols in order to wrap operations in units and specify the correct units behavior for different operations. Given that the cross-chunk operations implemented by Dask are fairly complex, we created the unyt_dask_array as a subclass of a standard Dask array. All of the units-related operations are then handled using hidden unyt_quantity attributes and decorators and anytime the units-related operation indicates a unit conversion, those conversion factors are multiplied onto the dask array.

A performance comparison

Using unyt dask arrays comes with the enhanced performance expected from using dask arrays, but it's important that ensure that our extra tracking of units is not significantly undermining performance relative to normal dask arrays.

We can do some initial testing by creating arrays for each of our array flavors, a plain numpy.ndarray, a plain unyt.unyt_array, a plain dask.array and a unyt_dask_array, and comparing execution time of an operation that causes a change in units:

```
array_shape = (int(1e8), )
chunk_size = 1e6

plain_numpy = np.ones(array_shape[0])
plain_unyt = unyt_array(plain_numpy,'m')
plain_dask = da.ones(array_shape[0], chunks = (chunk_size,))
unyt_dask = unyt_dask_array.unyt_from_dask(plain_dask,'m')
```

And for each of our arrays, we'll compute the time:

```
%%timeit
(plain_numpy ** 2).mean()
```

```
215 ms ± 9.69 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
(plain_unyt ** 2).mean()
```

```
209 ms ± 6.58 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
(plain_dask ** 2).mean().compute()
```

```
79.7 ms ± 3.13 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit
(unyt_dask ** 2).mean().compute()
```

```
70.5 ms ± 2.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Operations with unit conversions will incur an extra penalty:

```
%%timeit
(plain_unyt.to('cm') ** 2).mean()
```

```
368 ms ± 11.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
(unyt_dask.to('cm') ** 2).mean().compute()
```

```
101 ms ± 414 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

but we can see that (1) better performance for unyt_dask arrays than plain unyt_array and (2) similar performance for our unyt_dask and plain dask.array arrays. We show some more complete performance testing below.

An aside on when to convert units

As a small aside, it's worth a reminder that when stringing together operations you can sometimes save on computation by delaying the scalar operation until after any reductions. Every unit conversion is an elementwise-multiplication of a constant and so if that multiplciation by a constant can be delayed until after a reduction, you can save on the number of multiplications. For example, the following operations are equivalent:

```
result = ( ( 100 * plain_numpy )** 2).mean()
result_convert_after = (plain_numpy** 2).mean() * (100 **2)

print([result, result_convert_after, result == result_convert_after])
```

```
[10000.0, 10000.0, True]
```

since our unit conversions are simply scalar multiplications, the unit equivalent would be:

```
result = (plain_unyt.to('cm')** 2).mean()
result_convert_after = (plain_unyt** 2).mean().to('cm * cm')

print([result, result_convert_after, result == result_convert_after])
```

```
[unyt_quantity(10000., 'cm**2'), unyt_quantity(10000., 'cm**2'), array(True)]
```

So when converting units, converting after a computation:

```
%%timeit
(plain_unyt ** 2).mean().to('cm*cm')
```

```
211 ms ± 7.75 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

will be faster than converting before the operation:

```
%%timeit
(plain_unyt.to('cm') ** 2).mean()
```

```
376 ms ± 8.16 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

and in the case of our `unyt_dask` arrays:

```
%%timeit
(unyt_dask ** 2).to('cm*cm').mean().compute()
```

```
104 ms ± 3.28 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

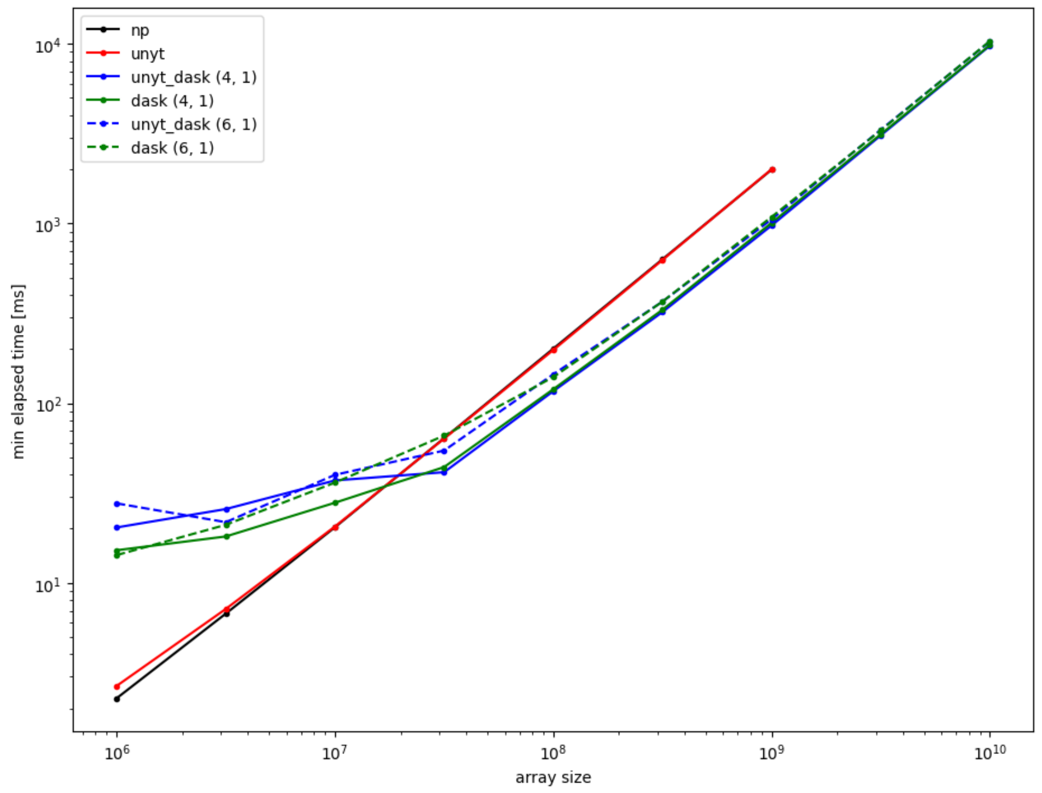
```
%%timeit
(unyt_dask.to('cm') ** 2).mean().compute()
```

```
108 ms ± 4.41 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

So we see that in operations where it's possible, it is worth putting off unit conversions until after array reductions so that we spend less time on multiplying by constants.

Final performance comparison

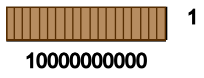
Returning to the general question of unyt-dask array performance, we want to be sure that our unyt-dask arrays are preforming on-par with standard dask arrays. Towards that end, we've run a suite of performance tests measuring the time to execute `(x ** 2).mean()` vs size of the array, `x`. The full code is available at `/code/test_daskunyt.py` and the following figure captures the results:



The y-axis is the minimum execution time of the operation, x-axis is the size of the array. The black and red curves are standard numpy and unyt arrays, respectively. The blue and green curves are dask and unyt-dask arrays for different number of workers (4 and 6, both single-threaded). The chunksize is fixed at 1e7 for all runs. So we can see that at small array sizes, standard `numpy` and `unyt_array` arrays are faster (not surprising), but as array size increases, the `dask` and `unyt_dask` arrays are faster. We can also see that the unyt-dask arrays perform similarly to plain dask arrays and at larger array sizes provide a decent speedup compared to plain unyt arrays. Furthermore, the unyt-dask arrays allows computation on larger-than memory arrays. The largest array tests would require 80 Gb of memory if using a plain unyt array:

```
unyt_dask_array.unyt_from_dask(da.ones((1e10,), chunks=(1e7)),'m')
```

	Array	Chunk
Bytes	80.00 GB	80.00 MB
Shape	(10000000000,)	(10000000,)
Count	1000 Tasks	1000 Chunks
Type	float64	numpy.ndarray
Units	m	m



While the unyt-dask arrays may be quite useful to the general SciPy community, we are experimenting with using them directly within yt where the “chunks” of the dask-arrays are decided using yt’s spatial indexing of datasets. We discuss this in the following section.

A Daskified yt : reading data

how yt works and how Dask can help

For discrete particle datasets, yt leverages several approaches to achieve efficient lazy loading of large datasets. When a user then reads in data, for example with code like

```
ds = yt.load_sample()
sp = ds.sphere(ds.domain_center, 0.25)
density = sp[("PartType0", "Density")]
```

yt will:

1. generate a spatial bitmap index for the dataset
2. use bitmap indexing to find data_files that overlap with a selection
3. iterate over the `data_files`, for each of `data_files`: a. open the file b. generate a mask using the selection object and a particle's coordinates c. read the data, applying the mask

Step 0 is a one-time step for a dataset that uses Morton Indexing, a spatial bitmap-indexing algorithm, to efficiently map a regions of a physical domain to the data whether on-disk or elsewhere (see the yt 4.0 paper, in prep). So as the first step of a selection, yt will first find the bitmap indices that intersect a selection object in order to quickly identify only those files that contain data within the selector and avoid reading datafiles unnecessarily. To further enhance read times, yt applies data selections at read to datafiles individually so that only data satisfying the selection criteria is returned (rather than reading the whole array into memory and then applying a selection).

This approach can be readily adapted to dask arrays in such a way that simplifies yt's code, particularly yt's frontend infrastructure. In yt, a frontend corresponds to data-specific methods for reading data. At present, implementing a new frontend requires writing an implementation of step 2 above, which requires some knowledge of yt's internal working (the data_files, selector objects) as well as knowledge of a frontend's native data structures (to read the raw data). With Dask, however, we can maintain the same efficiency while separating step 2 above into separate methods.

In simplified pseudo-code, this looks like:

```
def _read_particle_fields(self, fields, selector_obj, ):

        # assemble the self.data_files that intersect the selector
        dfi, nfiles = self._identify_file_masks(dobj)
        data_file_subset = [self.data_files[df_i] for df_i in dfi]

        # get a dask array where each chunk is a data_file
        data = self.io._read_from_datafiles(data_file_subset, fields_to_read)

        # apply a selector object to each chunk of the dask array
        data = self.apply_selector_mask(fields_to_return, selector_obj) # uses map_blocks function

        return data
```

where `read_from_datafiles` constructs dask arrays from delayed reads (again, pseudo-code):

```
def _read_from_datafiles(data_files, fields_to_read):
    delayed_array = []
    for data_file in data_files:
        shape, dtype = self._get_data_file_ptype_counts(data_file, fields_to_read)
        vals = dask_delayed(frontend_read_particle_fields)(data_file, field_info)  # this is the 
actual io call
        delayed_array.append(dask_array.from_delayed(vals, shape, dtype=dtype))
    return dask.array.concatenate(delayed_array, axis=0)
```

What's nice about the above Daskified approach is that while `_read_particle_fields` is written linearly (read in data then apply selector to data) but because the data arrays here are dask arrays, the selector is applied by-chunk using `dask.array.map_blocks` so that we are **still** handling each chunk separately. This greatly simplifies frontend development: to write the io methods for a new frontend, one need only implement a function to read from a single `data_file` (called `frontend_read_particle_fields` above)! Additionally, we can directly return Dask arrays, allowing a user to easily analyze their data in parallel when yt does not already have an optimized option.

Examples: simple reads and selections

These exmaples use the scipy2021 branch (see setup) to load in a Gadget HDF5 dataset:

```
%%time
import yt
ds = yt.load_sample("snapshot_033")
ad = ds.all_data()
```

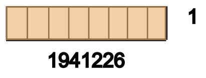
```
yt : [INFO     ] 2021-06-25 15:36:48,317 Files located at
/home/chris/hdd/data/yt_data/yt_sample_sets/snapshot_033.tar.gz.untar/snapshot_033/sn
ap_033.
yt : [INFO     ] 2021-06-25 15:36:48,317 Default to loading snap_033.0.hdf5 for
snapshot_033 dataset
yt : [INFO     ] 2021-06-25 15:36:48,404 Parameters: current_time              =
4.343952725460923e+17 s
yt : [INFO     ] 2021-06-25 15:36:48,405 Parameters: domain_dimensions         = [1 1
1]
yt : [INFO     ] 2021-06-25 15:36:48,405 Parameters: domain_left_edge          = [0.
0. 0.]
yt : [INFO     ] 2021-06-25 15:36:48,406 Parameters: domain_right_edge         = [25.
25. 25.]
yt : [INFO     ] 2021-06-25 15:36:48,406 Parameters: cosmological_simulation   = 1
yt : [INFO     ] 2021-06-25 15:36:48,406 Parameters: current_redshift          =
-4.811891664902035e-05
yt : [INFO     ] 2021-06-25 15:36:48,407 Parameters: omega_lambda              =
0.762
yt : [INFO     ] 2021-06-25 15:36:48,407 Parameters: omega_matter              =
0.238
yt : [INFO     ] 2021-06-25 15:36:48,408 Parameters: omega_radiation           = 0.0
yt : [INFO     ] 2021-06-25 15:36:48,408 Parameters: hubble_constant           = 0.73
yt : [INFO     ] 2021-06-25 15:36:48,498 Allocating for 4.194e+06 particles
Loading particle index:  92%|█████████ | 11/12 [00:00<00:00, 163.65it/s]
```

```
CPU times: user 2.9 s, sys: 184 ms, total: 3.09 s
Wall time: 3.08 s
```

```
%%time
field = ("PartType0","Density")
data = ad[field]
data
```

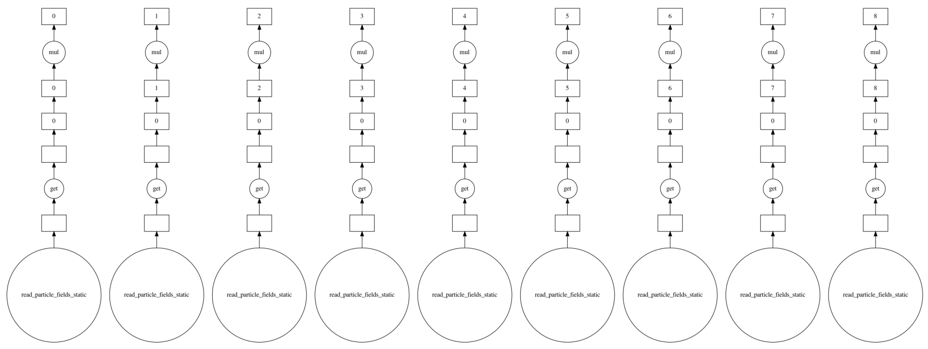
```
CPU times: user 143 µs, sys: 26 µs, total: 169 µs
Wall time: 175 µs
```

	Array	Chunk
Bytes	15.53 MB	2.10 MB
Shape	(1941226,)	(262144,)
Count	45 Tasks	9 Chunks
Type	float64	numpy.ndarray
Units	code_mass/code_length**3	code_mass/code_length**3



So we see we get back a unyt-dask array with 9 chunks nearly instantaneously. And those 9 chunks are nicely parallelized:

```
data.visualize()
```



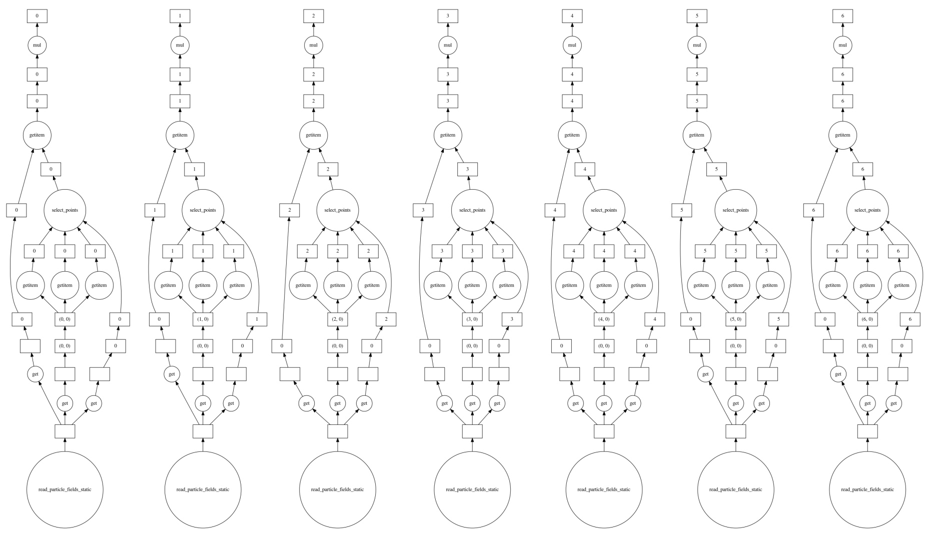
Comparing to a sphere selection centered at the domain center:

```
sp = ds.sphere(ds.domain_center, ds.quan(0.25,'code_length'))
spData = sp[field]
spData
```

	Array	Chunk
Bytes	unknown	unknown
Shape	(nan,)	(nan,)
Count	119 Tasks	7 Chunks
Type	float64	numpy.ndarray
Units	code_mass/code_length**3	code_mass/code_length**3

we see we only have 7 chunks since the particle indexing has eliminated 2 of our chunks. We also see that we don't know the final shape yet, because we are applying an array mask within our graph. Our graph in this case is still nicely parallel but has some extra complications as it applies the selector object to each chunk (this gets skipped when reading in `all_data`), which requires reading the coordinates and particle smoothing length for our field:

```
spData.visualize()
```



Comparing the execution time between the two, the selector application requires a bit of extra work:

```
%%timeit
ad = ds.all_data()
data = ad[field]
_ = data.compute()
```

```
26.5 ms ± 5.73 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit
sp = ds.sphere(ds.domain_center, ds.quan(0.25,'code_length'))
spData = sp[field]
_ = spData.compute()
```

```
126 ms ± 2.3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

These execution times are comparable to standard yt. After checking out `main` and re-running (restart the notebook, then run just these cells):

```
%%time
import warnings
warnings.filterwarnings("ignore")  # avoiding the ambiguous field deprecieation
warnings for this frontend.

import yt
ds = yt.load_sample("snapshot_033")
ad = ds.all_data()
```

```
yt : [INFO     ] 2021-06-25 15:53:10,462 Sample dataset found in
'/home/chris/hdd/data/yt_data/yt_sample_sets/snapshot_033/snap_033.0.hdf5'
yt : [INFO     ] 2021-06-25 15:53:10,580 Parameters: current_time              =
4.343952725460923e+17 s
yt : [INFO     ] 2021-06-25 15:53:10,581 Parameters: domain_dimensions         = [1 1
1]
yt : [INFO     ] 2021-06-25 15:53:10,581 Parameters: domain_left_edge          = [0.
0. 0.]
yt : [INFO     ] 2021-06-25 15:53:10,582 Parameters: domain_right_edge         = [25.
25. 25.]
yt : [INFO     ] 2021-06-25 15:53:10,583 Parameters: cosmological_simulation   = 1
yt : [INFO     ] 2021-06-25 15:53:10,583 Parameters: current_redshift          =
-4.811891664902035e-05
yt : [INFO     ] 2021-06-25 15:53:10,583 Parameters: omega_lambda              =
0.762
yt : [INFO     ] 2021-06-25 15:53:10,584 Parameters: omega_matter              =
0.238
yt : [INFO     ] 2021-06-25 15:53:10,584 Parameters: omega_radiation           = 0.0
yt : [INFO     ] 2021-06-25 15:53:10,584 Parameters: hubble_constant           = 0.73
yt : [INFO     ] 2021-06-25 15:53:10,686 Allocating for 4.194e+06 particles
Loading particle index:  92%|█████████ | 11/12 [00:00<00:00, 241.55it/s]
```

```
CPU times: user 3.74 s, sys: 31.4 ms, total: 3.78 s
Wall time: 3.77 s
```

```python
field = ("PartType0","Density")
```

```python
%%timeit
ad = ds.all_data()
data = ad[field]
```

```
12.8 ms ± 792 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```python
%%timeit
sp = ds.sphere(ds.domain_center, ds.quan(0.25,'code_length'))
spData = sp[field]
```

```
145 ms ± 3.21 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

So we see the Daskified all_data read is a tad slower while the selector application is a hair faster, but they're well within an acceptable range. If we spin up a dask Client, our read will happen in parallel:

```python
from dask.distributed import Client
```

```python
c = Client(n_workers = 4, threads_per_worker=1)
```

```python
c
```

Client

Scheduler: tcp://127.0.0.1:40239
Dashboard: http://127.0.0.1:8787/status

Cluster

Workers: 4
Cores: 4
Memory: 33.24 GB

```python
import yt
ds = yt.load_sample("snapshot_033")
```

```
yt : [INFO     ] 2021-06-25 16:52:10,780 Files located at
/home/chris/hdd/data/yt_data/yt_sample_sets/snapshot_033.tar.gz.untar/snapshot_033/sn
ap_033.
yt : [INFO     ] 2021-06-25 16:52:10,782 Default to loading snap_033.0.hdf5 for
snapshot_033 dataset
yt : [INFO     ] 2021-06-25 16:52:10,903 Parameters: current_time              =
4.343952725460923e+17 s
yt : [INFO     ] 2021-06-25 16:52:10,904 Parameters: domain_dimensions         = [1 1
1]
yt : [INFO     ] 2021-06-25 16:52:10,905 Parameters: domain_left_edge          = [0.
0. 0.]
yt : [INFO     ] 2021-06-25 16:52:10,905 Parameters: domain_right_edge         = [25.
25. 25.]
yt : [INFO     ] 2021-06-25 16:52:10,906 Parameters: cosmological_simulation   = 1
yt : [INFO     ] 2021-06-25 16:52:10,906 Parameters: current_redshift          =
-4.811891664902035e-05
yt : [INFO     ] 2021-06-25 16:52:10,907 Parameters: omega_lambda              =
0.762
yt : [INFO     ] 2021-06-25 16:52:10,908 Parameters: omega_matter              =
0.238
yt : [INFO     ] 2021-06-25 16:52:10,909 Parameters: omega_radiation           = 0.0
yt : [INFO     ] 2021-06-25 16:52:10,909 Parameters: hubble_constant           = 0.73
yt : [INFO     ] 2021-06-25 16:52:11,000 Allocating for 4.194e+06 particles
Loading particle index:  92%|█████████ | 11/12 [00:00<00:00, 145.95it/s]
```

```
field = ("PartType0","Density")
```

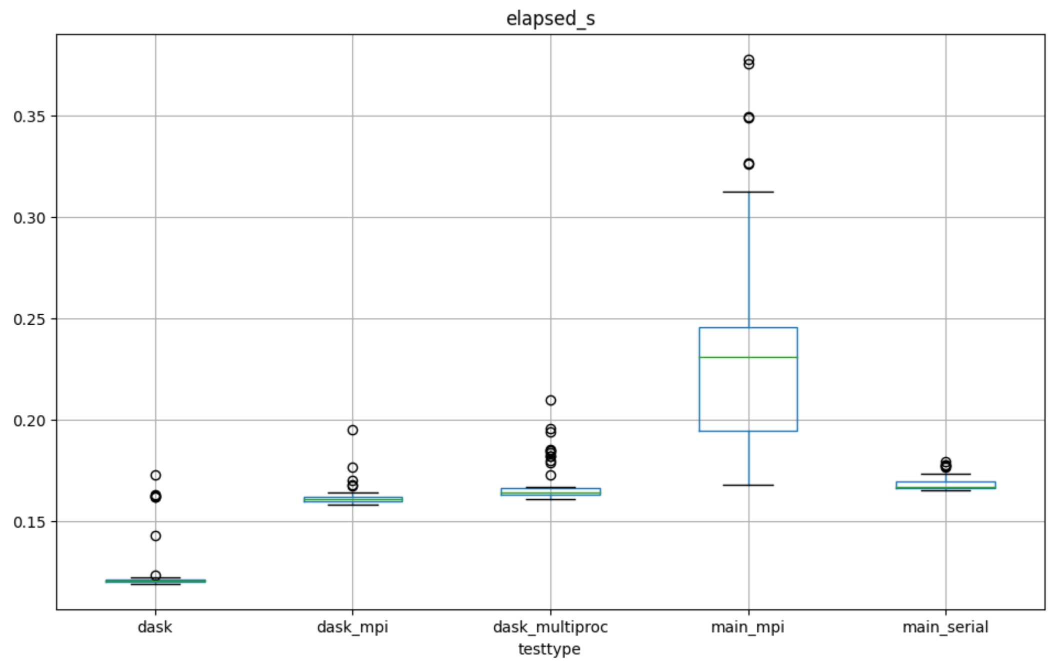
```
%%timeit
sp = ds.sphere(ds.domain_center, ds.quan(0.25,'code_length'))
spData = sp[field].compute()
```

```
419 ms ± 20.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
c.close()
```

So we see our parallel read is a bit slower. This isn't surprising as using multiple workers will incur some overhead communication. But in general these time differences are fairly small.

In the `./code/` directory of [this poster's repository](), we've written a number of scripts to measure the read time for a single field within a sphere for the `snapshot_033` dataset for all available methods, including MPI which is easier to test from command line. The following image shows a boxplot of the read time (excluding dataset loadtime and index build) for each method:



The test types are defined as follows:

test type	description
`dask`	default client configuration
`dask_mpi`	uses the dask-mpi scheduler with 4 workers
`dask_multiproc`	uses standard dask client with 4 workers, 1 thread per worker
`main_mpi`	main yt branch with MPI using 4 processors
`main_serial`	main yt branch, standard read

The boxplot for each test type shows the median as a horizontal green line, the inter-quartile range (IQR) from the first to third quartile as a blue box, +/- 1.5 * IQR as the horizontal black lines and outliers as black circles.

50-60 tests were performed for each case. The results show that the daskified read is at least as fast as the version on main. The extra communication results in some larger outliers (black circles) but the minimum read times are on par or slightly faster in the dask test cases. The large spread on when executing on main with MPI is surprising and may be due to machine setup, but in any case the similar read times for all dask methods and the standard main read is encouraging.

In the following section, we conduct similar tests for a daskified yt operation that uses the daskified particle reader.

A Daskified yt : using data

Much of the power of yt lies in what happens to data after it is read in. Many of the operations are designed to interface with the chunk iterator described in the previous section so that calculations can be applied to chunks separately and then aggregated without loading an entire data array into memory.

daskified `derived_quantity` calculations

Some of the simpler yt operations are defined by the `derived_quantities` class, which registers `quantities` to datasets. For example,

```
import yt
ds = yt.load_sample("snapshot_033")
sp = ds.sphere(ds.domain_center, 0.8)
sp.quantities.extrema([("PartType0", "Density"), ("PartType0", "Temperature")])
```

returns the max/min values for each field. Other `quantities` include operations for calculating weighted averages, spatial locations of extrema values, angular momentum and more. All of these operations work within yt’s chunking context with `process_chunk` reduction operations applied to each chunk and then aggregated.

direct dask-array approach

Given that the daskified particle reader described in the previous section can return a delayed `unyt_dask` array, it is straightfowrd to reduce the code complexity of the `derived_quantity` calculations with standard dask operations. Consider finding the current class for finding the extrema of a field:

```
class Extrema(DerivedQuantity):
    # <<<< docstring deleted >>>>>

    def count_values(self, fields, non_zero):
        self.num_vals = len(fields) * 2

    def __call__(self, fields, non_zero=False):
        fields = list(iter_fields(fields))
        rv = super().__call__(fields, non_zero)
        if len(rv) == 1:
            rv = rv[0]
        return rv

    def process_chunk(self, data, fields, non_zero):
        vals = []
        for field in fields:
            field = data._determine_fields(field)[0]
            fd = data[field]
            if non_zero:
                fd = fd[fd > 0.0]
            if fd.size > 0:
                vals += [fd.min(), fd.max()]
            else:
                vals += [
                    array_like_field(data, HUGE, field),
                    array_like_field(data, -HUGE, field),
                ]
        return vals

    def reduce_intermediate(self, values):
        # The values get turned into arrays here.
        return [
            self.data_source.ds.arr([mis.min(), mas.max()])
            for mis, mas in zip(values[::2], values[1::2])
        ]
```

The superclass `DervideQuantity`'s `__call__()` method will iterate over the chunks, calling `process_chunk` for each one to find each chunks min/max value. Those min/max values are then agreggated with the `reduce_intermediate` function. But when our particle IO returns a `unyt_dask` array, we do not need to explicitly loop over chunks and just call `.min()` and `.max()` on our dask arrays. Our modified class assembles a list of delayed actions:

```
class Extrema(DerivedQuantity):

    # <<<< trimmed >>>>

    def _get_delayed(self, fields, non_zero):
        if non_zero:
            dl = []
            for field in fields:
                data = self.data_source[field]
                msk = data != 0.
                dl.append((data[msk].min(), data[msk].max()))
        else:
            dl = [(self.data_source[field].min(), self.data_source[field].max()) for field in
fields]
        return dl

    def __call__(self, fields, non_zero=False):
        return super().__call__(fields, non_zero)
```

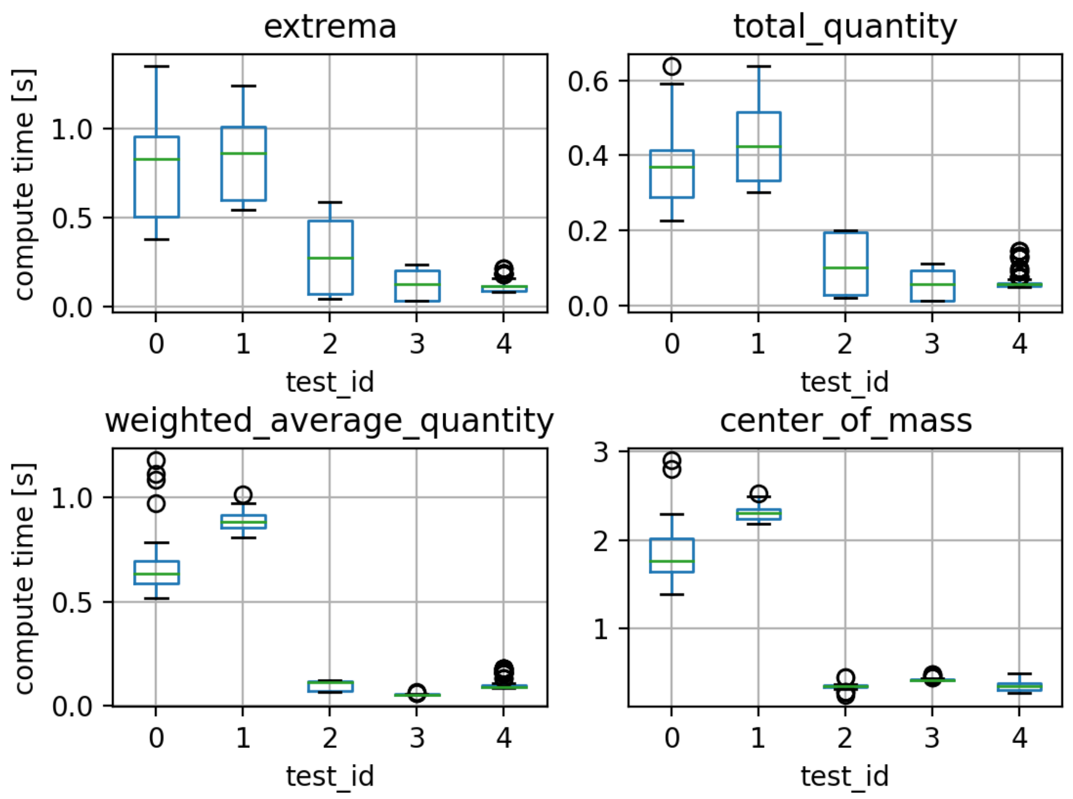
and those delayed actions are computed in the `super().__call()__`. Other more complex calculations can be simplified as well – the weighted averages for example can be replaced with calls to Dask's implementation of `average`.

In practice however, the performance of this naive dask-array approach is not quite as good as the yt-native approach. Here we compare execution times for different `derived_quantity` operations between methods and testing conditions (the test type) as follows:

The `test_id` refers to the following configurations:

`test_id`	test type	notes
0	`dask_multiproc`	4 workers, 1 thread per worker
1	`dask_mpi`	4 workers
2	`dask`	no explicit client spinup (defualt `Client`)
3	`main_serial`	standard yt
4	`main_mpi`	standard yt, 4 workers

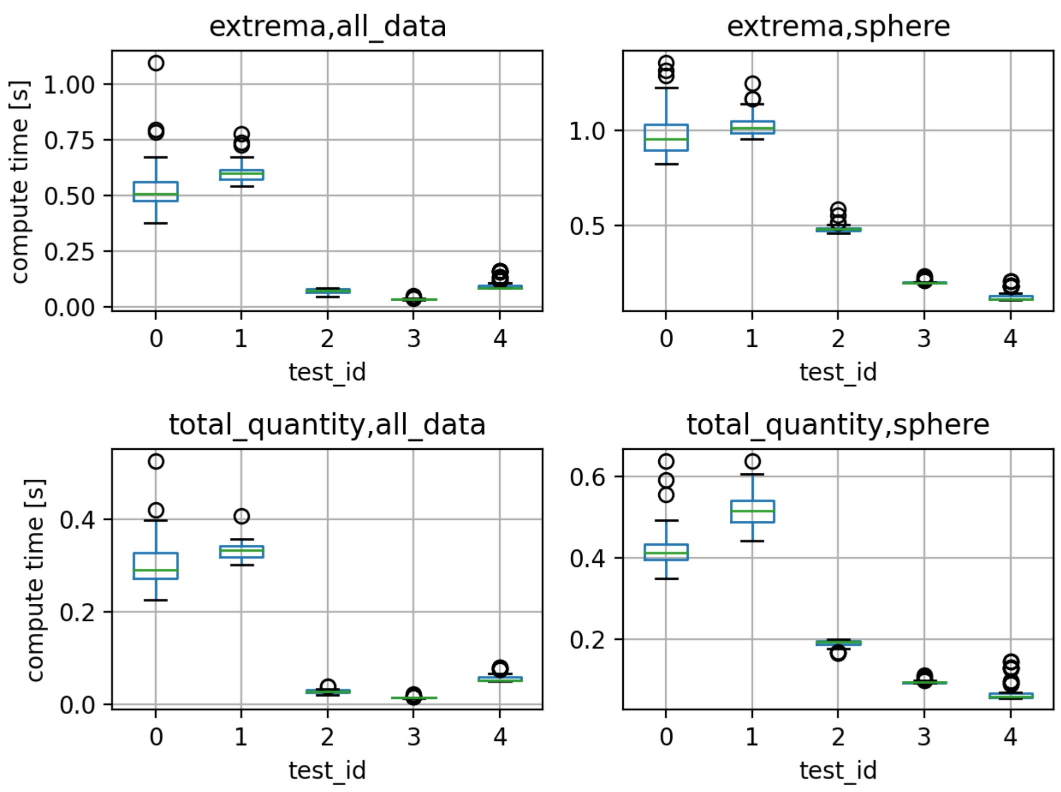
The `dask_` tests use the daskified code branch (link), the `main_` tests use the current `main` branch of yt. Operations are tested on both `all_data` and `sphere` selector objects and include: `extrema`, `total_quantity`, `weighted_average_quantity` and `center_of_mass`. The code for the tests is available in the `./code` directory of this poster's repository (link, files are `./code/test_quantities_*.py`). Grouping only by the test type, the elapsed execution time for each operation are:



The above tests group together multiple tests over `sphere` and `all_data` selector objects:

```
sp = ds.sphere(ds.domain_center, 0.8)
ad = ds.all_data()
```

Of the test operations that are currently function for both these operations, we can additionall compare execution times between the selector objects:



From the above comparisons, it is clear that the dask-array approach is generally not performing as well as the native yt approach. The main reason for this, is that the native yt methods were carefully written to optimize chunk iteration. For example, when calculating the extrema for multiple fields, each chunk is read only once during which the data for each field is aggregated and then later reduced to the final extrema values. In the dask-array approach, the task graph includes a read for each field being aggregated. The `weighted_average_quantities` and `center_of_mass` daskified calculations perform well compared to their native yt approaches for the default dask `Client`. To improve our approach, we may be able to use a delayed workflow.

dask-delayed approach

In previous work (overview, detailed notebook), we compared different methods of calculating a yt profile (a binned statistic) and showed that we actually got the best performance when using a dask delayed workflow that directly uses yt's already optimized algorithm as opposed to a `dask array.map_blocks` implementation. We're currently in the process of repeating the above `derived_quantities` test with a similar delayed work flow approach for calculating derived quantities.

Dasken(yt)

In the previous sections we've demonstrated a number of daskified components within yt. In general these approaches show promise for reducing yt's code complexity, expanding the distributed computing power outside of an MPI framework and introducing improvements that may be useful beyond yt (i.e., `unyt_dask` arrays). As development continues, we will be further integrating with the yt community: in addition to the existing `unyt_dask` PR, primary development will move to yt's `daskening` development branch with an accompanying YTEP, with a target release of yt 5.0 for dask support. If you're interested in Dask development within yt, reach out on slack or via email (chavlin@illinois.edu)!

Repository notes

The dask functionality for `yt` and `unyt` shown in this poster rely on several branches that are in progress. For reproducibility, the version of the yt branch used for the demonstrating and testing used in this presentation is at scipy2021_dask. Additionally the `dask` branch relies on the `unyt` PR #185 branch.

Both branches must be installed from source to reproduce the notebooks here.

The native yt tests are run using the scipy2021_main branch, a frozen copy of the yt development branch from July 1, 2021. Switching branches will likely require rebuilding after switching.

notes on this presentation

This SciPy2021 poster was created using Jupypter book, the source repository at https://github.com/chrishavlin/scipy2021_ytDaskening includes instructions for building and deploying the Jupyter book.