daisy Documentation

Release 0.1.0

Joe Jevnik

January 22, 2017

Contents

```
dask + lazy = daisy
```

A dask backend for lazy

CHAPTER 1

What is `daisy`?

`daisy` is an experiment to finally use lazy for something useful. `daisy` is meant to be an alternative to `dask.delayed()` for automatically creating computation graphs from functions.

CHAPTER 2

Example

Given the following setup:

```
from daisy import autodask, inline, register_get
from dask import delayed
from dask.threaded import get
from lazy import strict
import numpy as np


@inline
def f(a, b):
    return a + b


def g(a, b):
    return f(f(a, b), f(a, b))

autodask_g = autodask(g, inline=True)
delayed_g = delayed(g)


register_get(get)


arr = np.arange(1000000)
```

To start, let's make sure these all do the same thing:

```
>>> (g(arr, arr) == delayed_g(arr, arr).compute()).all()
True

>>> (g(arr, arr) == autodask_g(arr, arr)).all()
True
```

Now we will run some not very scientific profiling runs:

```
In [1]: %timeit g(arr, arr)
100 loops, best of 3: 9.34 ms per loop

In [2]: %timeit delayed_g(arr, arr).compute()
100 loops, best of 3: 10.2 ms per loop

In [3]: %timeit strict(autodask_g(arr, arr))
100 loops, best of 3: 3.63 ms per loop
```

2.1 Why is this faster?

This is a very good case for autodask because we can dramatically reduce the amount of work we are doing. In the normal function and `dask.delayed` cases we will fall `f(a, b)` twice, and then add those together. In the `autodask` case will will just directly execute `a + b` once, and then add that to itself. We have totally removed `f` from the graph, and instead just use + directly.

We have used a very large input here to see a speedup. One goal I have is to reduce the overhead to make this work for smaller inputs and smaller expressions. I would like to try this with real workloads to see if the amount of reduced work causes as dramatic of speedups.

2.2 More shared work

Let's look at a more radical example:

```
from daisy import inline, autodask, ltree_to_dask
from lazy.tree import LTree

@inline
def f(a, b):
    return a + b

@inline
def g(a, b):
    return a + b + 1

def h(a, b):
    return f(a, b) + g(a, b)
```

```
In [1]: (h(arr, arr) == autodask_h(arr, arr)).all()
Out[1]: True

In [2]: %timeit h(arr, arr)
100 loops, best of 3: 9.02 ms per loop

In [3]: %timeit strict(autodask_h(arr, arr))
100 loops, best of 3: 5.9 ms per loop
```

The reason this is faster is that we can actually share the work of computing `a + b` even though they are in totally separate functions!

```
In [4]: from lazy.tree import LTree

In [5]: from daisy import ltree_to_dask

In [6]: ltree_to_dask(LTree.parse(autodask_h(arr, arr)))[0]
Out[6]:
{'4876ef4b-832a-4058-94f7-29a6fb998ea6': <wrapped-function add>,
 '5a2bee49-2a31-4e01-887f-bfaef7ebb27a': 1,
 'add-39c81b36-ad91-4c2e-93c7-2a74d485fd7b': (<function dask.compatibility.apply>,
  '4876ef4b-832a-4058-94f7-29a6fb998ea6',
  ['add-d581fba1-d73f-42db-8e41-9bff1c803941',
   'add-54f2153f-4cbe-4dfc-babe-cbde4c7d66c1'],
  (dict, [])),
 'add-54f2153f-4cbe-4dfc-babe-cbde4c7d66c1': (<function dask.compatibility.apply>,
  '4876ef4b-832a-4058-94f7-29a6fb998ea6',
```

```
  ['add-d581fba1-d73f-42db-8e41-9bff1c803941',
   '5a2bee49-2a31-4e01-887f-bfaef7ebb27a'],
  (dict, [])),
 'add-d581fba1-d73f-42db-8e41-9bff1c803941': (<function dask.compatibility.apply>,
  '4876ef4b-832a-4058-94f7-29a6fb998ea6',
  ['f174fab9-9eb1-4448-991c-5437bd2d709e',
   'f174fab9-9eb1-4448-991c-5437bd2d709e'],
  (dict, [])),
 'f174fab9-9eb1-4448-991c-5437bd2d709e': array([     0,      1,      2, ..., 999997, 999998, 999999])
```

The key point here is that we only ever have `a + b` once in this graph.

CHAPTER 3

Contents

3.1 API Reference

3.1.1 Main

daisy.**autodask**(*func*, *, *inline*)

Mark that a function should lazily build up a call graph to be executed by dask.

Parameters

- **func** (*callable*) – The function to transform.
- **inline** (*bool*) – Should the function be inlined into other `autodask` functions? This should normally be True unless the function is strict on the argument.

Returns **transformed** – `func` with the transformations needed to build the call graph.

Return type callable

Notes

`autodask` transforms a function to build up a call graph which can be executed by dask. This is very similar to `dask.delayed()` which provides an imperitive API to dask.

Functional purity

`autodask` may only be applied to functions which a pure functions of their inputs. This means that a function must always be safe to memoize.

There is no guarantee about the execution order of `autodask` defered code. Repeated calls to a function with the same arguments may only be computed a single time.

Note: Things to be on the look out for when checking if a function is pure:

•IO

•Mutating structures

•Reading or writing to shared state (please stop this)

•Randomness

IO may be okay if you are alright with only executing the call once and in an undefined order. You may force the partial order of execution by explicitly passing the results of one IO call into the other calls that must follow it.

Building up our task graph

Unlike `dask.delayed()`, `autodask` is **lazy by default**. This means that `f(a, b)` will automatically turn into a dask task graph like:

```
{'name': (f, a, b)}
```

Note: `f`, `a` and, `b` may also be deferred computations themselves.

Dask will perform best if we can encode more information into the task graph before feeding it to dask. To do this, we can pass `inline=True` to autodask before decorating. If a function is inlineable then instead of defering the computation, we will enter the code and add the body of that function to the dask graph. For example, imagine we have defined `f` like:

```
@autodask(inline=True)
def f(a, b):
    return a + b + 1
```

When calling this function we know that it is safe to replace the task `(f, a, b)` with the task graph:

```
{'name_1': (add, a, b),
 'result': (add, 'name_1', 1)}
```

This will give dask more information to optimize the expression. We can also use this to collapse shared work. For example, imagine we have

```
@autodask(inline=True)
def g(a, b):
    return f(a, b) + f(a, b)
```

Because `f` is inlineable, we will enter the code and see what it adds to the graph. Because we are doing the same work twice, we can reduce it to a more simple task graph that will look more like:

```
{'name_1': (add, a, b),
 'f_result': (add, 'name_1', 1),
 'result': (add, 'f_result', 'f_result')}
```

This shows that we will not duplicate the work needed to add compute `f(a, b)` twice.

When it is unsafe to pass inline=True

There is no default for `inline` because it is a very important decision! On the one hand, we almost *always* want to pass `inline=True`; however, there are cases when inlining is not possible, and attempting to do so will give much worse performance.

Functions cannot be inlined into the graph if they are strict on their inputs. This means that to return a final defered computation they must scrutinize at least one of the inputs and normalize it to a concrete value.

There are many operations which will force computation, here are some common cases:

Branching on the input

```
def f(x):
    if p(x):
        return x + 1
```

```
        else:
            return x - 1
```

Iterating over the input with a for loop

```
def f(xs):
   total = 0
   for x in xs:
       total += 0
   return total
```

Explicitly strictly evaluating an input

```
def f(x):
    return strict(x)
```

Differences with dask.delayed

Lazy by default vs eager by default

While both `autodask` and `dask.delayed()` serve the same purpose, they go about it in different ways. `dask.delayed()` is **strict** by default. This means that by default, most functions will be entered immediatly instead of creating a task. This can be bad if the function does not know how to work with the `dask.delayed.Delayed` object or is strict on an input. Here is an example of a function in the `dask.delayed()` API:

```
@dask.delayed
def f(a, b):
    # lazy call: this will create a node like ``(f, a, b)`` in the
    # resulting task graph
    c = delayed(g)(a, b)

    # strict call: this will enter the code ``h`` immediatly and add the
    # body to the graph. This may not be safe!
    return h(c, b)
```

`autodask` takes a different approach and is **lazy** by default. This means that by default function calls just create a new task for the graph and are not executed eagerly. Here is the same function in the `autodask` API:

```
@autodask
def f(a, b):
    # lazy call: this will create a node like ``(f, a, b)`` in the
    # resulting task graph **unless ``g`` is an inline function**!
    c = g(a, b)

    # strict call: this will enter the code ``h`` immediatly and add the
    # body to the graph. This may not be safe!
    return inline(h)(c, b)
```

One advantage of the `autodask` approach is that that the potentially unsafe operation is called out explicitly, while we choose a more conservative graph construction strategy by default. We also allow functions to opt-in to inlining if they know it is safe to do so.

Magic

`autodask` uses much darker magic than `dask.delayed()`. This is nice because it allows us to do things like translate:

```
@autodask(inline=True)
def f(a, b):
    return a is b
```

into a dask graph like:

```
{'result': (operator.is_, a, b)}
```

We can also defer things like comprehensions and even literal construction.

> **Warning:** The magic required for `autodask` may be too much for people. It will not be easy to debug! `dask.delayed()` is a much more reasonable solution for most cases. You have been warned.

See also:

daisy.inline(), `lazy.strict()`, `dask.delayed()`

class `daisy.`**`inline`**(*func*)

A box that denotes that a function should be inlined in autodask.

Parameters **`func`** (*callable*) – The function to wrap.

Notes

`inline` can allow non-`autodask` functions to be inlined into the task graph. This is nice if you know that a function is a pure computation of its inputs and does not need to scrutinize an input to return a final computation.

Functions cannot be inlined into the graph if they are strict on their inputs. This means that to return a final defered computation they must scrutinize at least one of the inputs and normalize it to a concrete value.

There are many operations which will force computation, here are some common cases:

Branching on the input

```
def f(x):
    if p(x):
        return x + 1
    else:
        return x - 1
```

Iterating over the input with a for loop

```
def f(xs):
   total = 0
   for x in xs:
       total += 0
   return total
```

Explicitly strictly evaluating an input

```
def f(x):
    return strict(x)
```

See also:

daisy.autodask()

`daisy.`**`register_get`**(*get*)

Register the `get` function which will be used to evaluate `autodaskthunk` generated dask graphs.

By default, `dask.get()` will be used.

Parameters **`get`** (*callable[dict, str, any]*) – The get function.

Returns **`get`** – The `get` function unchanged.

Return type callable[dict, str, any]

3.1.2 Miscellaneous

class daisy.**autodaskthunk**
A thunk which is evaluated with dask.

Parameters

- **func** (*callable*) – The code for the closure.
- ***args** – The free variables.
- ****kwargs** – The free variables.

daisy.**ltree_to_dask**(*node*)
Convert an lazy.tree.LTree into a dask task graph.

Parameters **node** (*LTree*) – The node to convert into a dask graph.

Returns **dask** – The equivalent dask task graph.

Return type dict[str, any]

Notes

This function does common subexpression folding to produce a minimal graph.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

A

I

L

R