daisy Documentation

Release v0.2

Jan Funke, Tri Nguyen, Carolin Malin-Mayor, Arlo Sheridan, Philip

Feb 17, 2023

Contents

CHAPTER 1

API Reference

1.1 Convenience API

daisy.**run_blockwise**(*tasks*)

Schedule and run the given tasks.

Args:

list_of_tasks: The tasks to schedule over.

Return:

bool: *True* if all blocks in the given *tasks* were successfully run, else *False*

1.2 Block-wise Task Scheduling

class daisy.**Block**(*total_roi*, *read_roi*, *write_roi*, *block_id=None*, *task_id=None*)

Describes a block to process with attributes:

Attributes:

read_roi (*class:Roi*):

The region of interest (ROI) to read from.

write_roi (*class:Roi*):

The region of interest (ROI) to write to.

status (`BlockStatus`):

Stores the processing status of the block. Block status should be updated as it goes through the lifecycle of scheduler to client and back.

block_id (`int`):

A unique ID for this block (within all blocks tiling the total ROI to process).

task_id (`int`):

The id of the Task that this block belongs to.

Args:

total_roi(*class:Roi*):

The total ROI that the blocks are tiling, needed to find unique block IDs.

read_roi (*class:Roi*):

The region of interest (ROI) to read from.

write_roi (*class:Roi*):

The region of interest (ROI) to write to.

block_id (`int`, optional):

The ID to assign to this block. The ID is normally computed from the write ROI and the total ROI, such that each block has a unique ID.

task_id (`int`, optional):

The id of the Task that this block belongs to. Defaults to None.

class daisy.**Scheduler**(*tasks: List[daisy.task.Task], count_all_orphans=True*)

This is the main scheduler that tracks states of tasks.

The Scheduler takes a list of tasks, and upon request will provide the next block available for processing.

args:

tasks: the list of tasks to schedule. If any of the tasks have upstream dependencies these will be recursively enumerated and added to the scheduler.

count_all_orphans: bool: Whether to guarantee accurate counting of all orphans. This can be inefficient if your dependency tree is particularly deep rather than just wide, so consider flipping this to False if you are having performance issues. If False, orphaned blocks will be counted as "pending" in the task state since there is no way to tell the difference between the two types without enumerating all orphans.

class daisy.**Client**(*context=None*)

Client code that runs on a remote worker providing task management API for user code. It communicates with the scheduler through TCP/IP.

Scheduler IP address, port, and other configurations are typically passed to `Client` through an environment variable named 'DAISY_CONTEXT'.

Example usage:

def blockwise_process(block): ...

def main(): client = Client() while True:

with client.acquire_block() as block:

if block is None: break

blockwise_process(block) block.state = BlockStatus.SUCCESS # (or FAILED)

class daisy.**Context**(***kwargs*)

class daisy.**Task**(*task_id*, *total_roi*, *read_roi*, *write_roi*, *process_function*, *check_function=None*, *init_callback_fn=None*, *read_write_conflict=True*, *num_workers=1*, *max_retries=2*, *fit='valid'*, *timeout=None*, *upstream_tasks=None*)

Definition of a `daisy` task that is to be run in a block-wise fashion.

Args:

name (`string`):

The unique name of the task.

total_roi (*class:daisy.Roi*):

The region of interest (ROI) of the complete volume to process.

read_roi (*class:daisy.Roi*):

The ROI every block needs to read data from. Will be shifted over the `total_roi` to cover the whole volume.

write_roi (*class:daisy.Roi*):

The ROI every block writes data from. Will be shifted over the `total_roi` to cover the whole volume.

process_function (function):

A function that will be called as:

```
process_function(block)
```

with `block` being the shifted read and write ROI for each location in the volume.

If `read_write_conflict` is `True`, the callee can assume that there are no read/write concurencies, i.e., at any given point in time the ``read_roi`` does not overlap with the `write_roi` of another process.

check_function (function, optional):

A function that will be called as:

```
check_function(block)
```

This function should return `True` if the block was completed. This is used internally to avoid processing blocks that are already done and to check if a block was correctly processed.

If a tuple of two functions is given, the first one will be called to check if the block needs to be run, and if so, the second one will be called after it was run to check if the run succeeded.

init_callback_fn (function, optional):

A function that Daisy will call once when the task is started. It will be called as:

```
init_callback_fn(context)
```

Where *context* is the *daisy.Context* string that can be used by the daisy clients to connect to the server.

read_write_conflict (`bool`, optional):

Whether the read and write ROIs are conflicting, i.e., accessing the same resource. If set to `False`, all blocks can run at the same time in parallel. In this case, providing a `read_roi` is simply a means of convenience to ensure no out-of-bound accesses and to avoid re-computation of it in each block.

fit (`string`, optional):

How to handle cases where shifting blocks by the size of `write_roi` does not tile the `total_roi`. Possible options are:

"valid": Skip blocks that would lie outside of `total_roi`. This is the default:

```
|--------------------------|     total ROI

|rrrr|wwwwww|rrrr|              block 1
       |rrrr|wwwwww|rrrr|       block 2
                                no further block
```

"overhang": Add all blocks that overlap with `total_roi`, even if they leave it. Client code has to take care of save access beyond `total_roi` in this case.:

```
|--------------------------|     total ROI

|rrrr|wwwwww|rrrr|              block 1
       |rrrr|wwwwww|rrrr|       block 2
              |rrrr|wwwwww|rrrr|  block 3 (overhanging)
```

"shrink": Like "overhang", but shrink the boundary blocks' read and write ROIs such that they are guaranteed to lie within `total_roi`. The shrinking will preserve the context, i.e., the difference between the read ROI and write ROI stays the same.:

```
|--------------------------|     total ROI

|rrrr|wwwwww|rrrr|              block 1
       |rrrr|wwwwww|rrrr|       block 2
              |rrrr|www|rrrr|    block 3 (shrunk)
```

num_workers (int, optional):

The number of parallel processes to run.

max_retries (int, optional):

The maximum number of times a task will be retried if failed (either due to failed post check or application crashes or network failure)

timeout (int, optional):

Time in seconds to wait for a block to be returned from a worker. The worker is killed (and the block retried) if this time is exceeded.

class daisy.**DependencyGraph**(*tasks*)

1.3 Geometry

1.3.1 Coordinate

class daisy.**Coordinate**

A `tuple` of integers.

Allows the following element-wise operators: addition, subtraction, multiplication, division, absolute value, and negation. All operations are applied element wise and support both Coordinates and Numbers. This allows to perform simple arithmetics with coordinates, e.g.:

```
shape = Coordinate(2, 3, 4)
voxel_size = Coordinate(10, 5, 1)
size = shape*voxel_size # == Coordinate(20, 15, 4)
size * 2 + 1 # == Coordinate(41, 31, 9)
```

Coordinates can be initialized with any iterable of ints, e.g.:

```
Coordinate((1,2,3))
Coordinate([1,2,3])
Coordinate(np.array([1,2,3]))
```

Coordinates can also pack multiple args into an iterable, e.g.:

```
Coordinate(1,2,3)
```

is_multiple_of(*coordinate*)

Test if this coordinate is a multiple of the given coordinate.

round_division(*other*)

Will always round down if self % other == other / 2.

1.3.2 Roi

class daisy.**Roi**(*offset*, *shape*)

A rectangular region of interest, defined by an offset and a shape. Special Cases:

An infinite/unbounded ROI: offset = (None, None, ...) shape = (None, None, ...)

An empty ROI (e.g. output of intersecting two non overlapping Rois): offset = (None, None, ...) shape = (0, 0, ...)

A ROI that only specifies a shape is not supported (just use Coordinate).

There is no guessing size of offset or shape (expanding to number of dims of the other).

Basic Operations: Addition/subtraction (Coordinate or int) - shifts the offset elementwise (alias for shift)

Multiplication/division (Coordiante or int) - multiplies/divides the offset and the shape, elementwise

Roi Operations: Intersect, union

Similar to *Coordinate*, supports simple arithmetics, e.g.:

```
roi = Roi((1, 1, 1), (10, 10, 10))
voxel_size = Coordinate((10, 5, 1))
roi * voxel_size = Roi((10, 5, 1), (100, 50, 10))
scale_shift = roi*voxel_size + 1 # == Roi((11, 6, 2), (101, 51, 11))
```

Args:

offset (array-like of `int`):

The offset of the ROI. Entries can be `None` to indicate there is no offset (either unbounded or empty).

shape (array-like of `int`):

The shape of the ROI. Entries can be `None` to indicate unboundedness.

begin
Smallest coordinate inside ROI.

center
Get the center of this ROI.

contains(*other*)
Test if this ROI contains `other`, which can be another *Roi*, *Coordinate*, or `tuple`.

copy()
Create a copy of this ROI.

dims
The the number of dimensions of this ROI.

empty
Test if this ROI is empty.

end
Smallest coordinate which is component-wise larger than any inside ROI.

get_bounding_box()
Alias for `to_slices()`.

grow(*amount_neg=0*, *amount_pos=0*)
Grow a ROI by the given amounts in each direction:

Args:

amount_neg (*Coordinate* or `int`):

Amount (per dimension) to grow into the negative direction. Passing in a single integer grows that amount in all dimensions. Defaults to zero.

amount_pos (*Coordinate* or `int`):

Amount (per dimension) to grow into the positive direction. Passing in a single integer grows that amount in all dimensions. Defaults to zero.

intersect(*other*)
Get the intersection of this ROI with another *Roi*.

intersects(*other*)
Test if this ROI intersects with another *Roi*.

shift(*by*)
Shift this ROI.

size
Get the volume of this ROI. Returns `None` if the ROI is unbounded.

snap_to_grid(*voxel_size*, *mode='grow'*)
Align a ROI with a given voxel size.

Args:

voxel_size (*Coordinate* or `tuple`):

The voxel size of the grid to snap to.

mode (string, optional):

How to align the ROI if it is not a multiple of the voxel size. Available modes are 'grow', 'shrink', and 'closest'. Defaults to 'grow'.

to_slices()
Get a `tuple` of `slice` that represent this ROI and can be used to index arrays.

unbounded
Test if this ROI is unbounded.

union(*other*)
Get the union of this ROI with another *Roi*.

1.4 Arrays

1.5 Graphs

1.5.1 Graph

1.5.2 MongoDbGraphProvider

CHAPTER 2

Release notes

2.1 Release v0.2

2.1.1 Major changes

- Use Tornado for worker-scheduler communication. Communication between scheduler and workers is now using `tornado` instead of `dask` to be more lightweight and reliable. Furthermore, a worker client is persistent across blocks, allowing it to request and receive multiple blocks to process on. This change is heavily motivated by the long queuing delay of `lsf`/`slurm` and bring-up delay of `tensorflow`. Furthermore, the user now has an API for acquiring and releasing block, allowing them to write their own Python module implementation of workers. By **:user:'Tri Nguyen <trivoldus28>'**, **:user:'Jan Junke <funkey>'**
- Introduce `Task` and `Parameter`, and `daisy.distribute()` to execute `Task` chain.
- Changed `SharedGraphProvider` and `SharedSubGraph` APIs, and added many new features including ability to have directed and undirected graphs. For the mongo backend, also added the ability to store node/edge features in separate collections, and filter by node/edge feature values.

2.1.2 Notable features

- `Task` sub-ROI request. A sub-region of a task's available `total_roi` can be restricted/requested explicitly using the `daisy.distribute()` interface.

  Example:

```
task_spec = {'task': mytask, 'request': [daisy.Roi((3,), (2,))]}
daisy.distribute([task_spec])
```

  The `request` will be expanded to align with `write_roi`.

- Multiple `Task` targets. A single `daisy.distribute()` can execute multiple target tasks simultaneous.

  Example:

```
task_spec0 = {'task': mytask0}
task_spec1 = {'task': mytask1}
daisy.distribute([task_spec0, task_spec1])
```

Tasks' dependencies (shared or not) will be processed correctly.

- Periodic status report. Daisy gives a status report of running/finished tasks, blocks running/finished status, and an ETA based on the completion rate of the last 2 minutes.
- Z-order block-wise scheduling.

2.1.3 Maintenance

- Drop support for Python 3.4.x and 3.5.x. We have moved to using Python's `asyncio` capability as the sole backend for Tornado. Python 3.4.x does not have asyncio. While Python 3.5.x does have asyncio built-in, its implementation is buggy.
- search

Python Module Index

d

Index