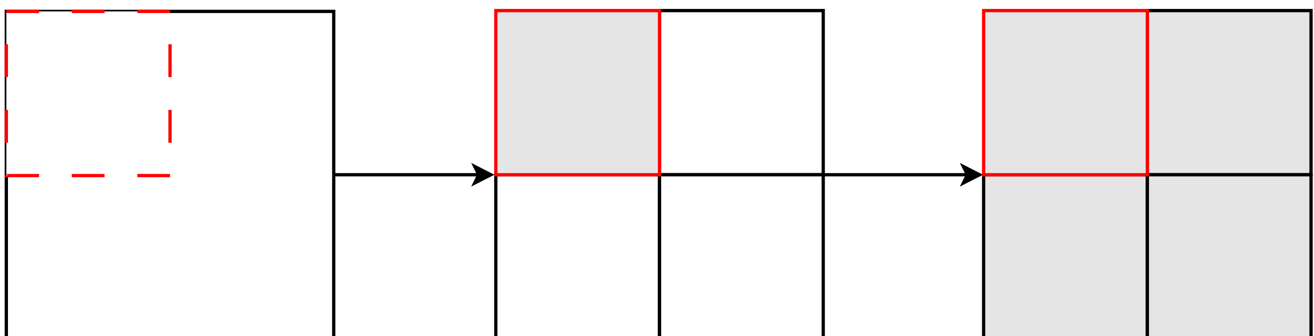
Decomposition and parallelism

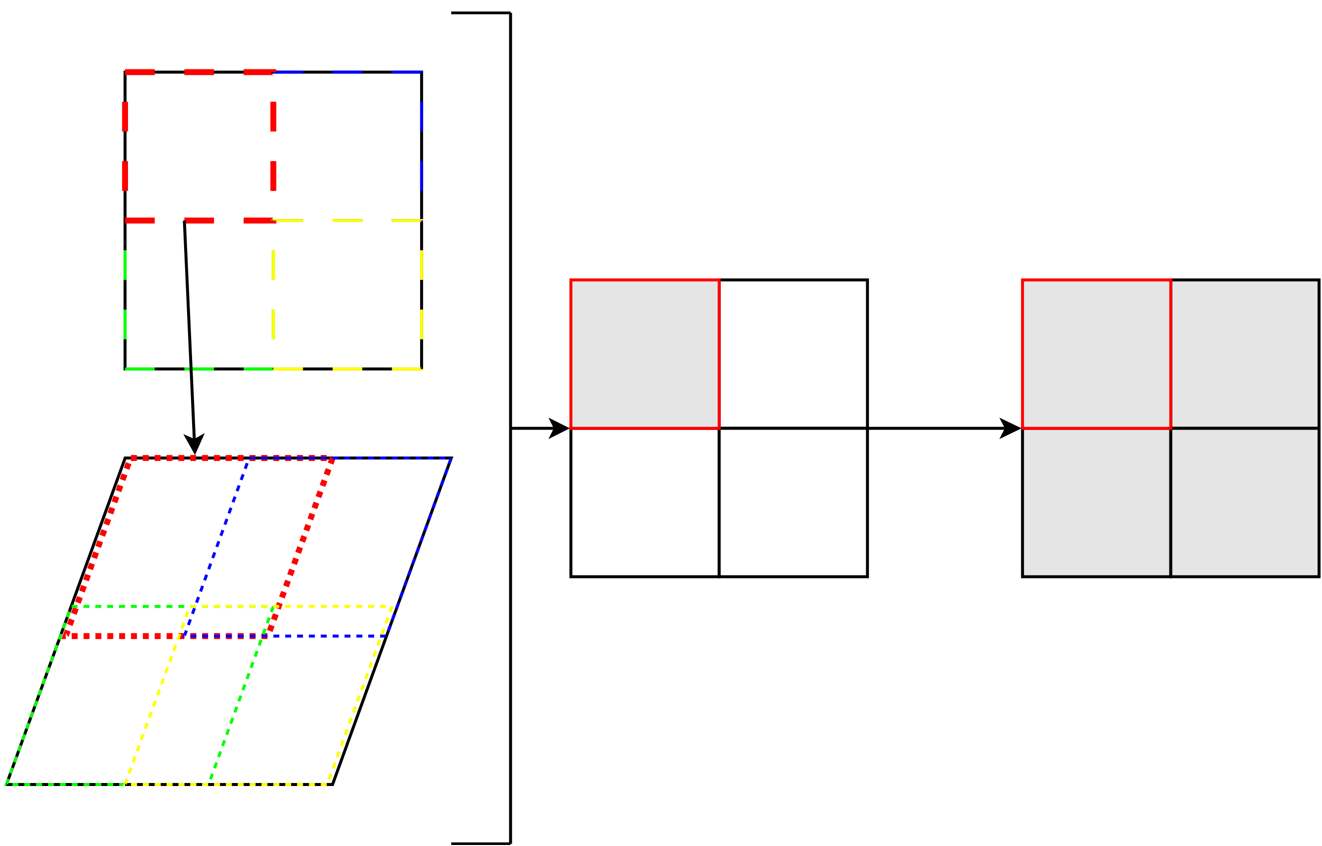
The decomposition module provides code to help you divide your processing into smaller pieces which can fit into memory, or be run in parallel. The decomposition framework will deal with splitting your cubes into sub cubes, running the processing, and combining the results back into a single cube. You need to define a function that can work on a portion of a cube to do your processing, the decomposition framework will do the rest. For the full set of reference documentation, the reader is referred to `ants.decomposition`.

Decomposition is supported for both *unary* and *binary* operations. The former is where we operate on a single field, processing its data in some way. The latter is where we operate on two fields, one the source and the other some target grid.

In the unary case, the source is simply turned into a mosaic via a class generator factory and dealt with piecemeal.



In the binary case, the target grid is decomposed to make a mosaic in the same way that the source is in the unary case. Each piece of these target mosaic pieces are then used to retrieve the corresponding overlapping source. The binary operation is then performed with each source-target pair.



Basic Usage

Usage of the decomposition framework can be achieved via a simple interface (`ants.decomposition.decompose()`):

No decomposition:

```
res = operation(source, target)
```

Decomposition:

```
res = decomp.decompose(operation, source, target)
```

This interface gives control over how the problem is to be decomposed, which framework is to be utilised, whether serial, single machine or multi-machine parallelism.

`ants.decomposition.decompose`(*operation, sources, targets=None, split=None, framework=None*) [source]

Breaks a source cube into smaller slices to perform a parallel calculation.

Parameters:

- **operation** (*callable*) – Operation to be computed on each decomposed piece within the decomposition framework, whether unary or binary.
- **sources** (One or more `iris.cube.Cube`) – Cube(s) upon which the operation will be performed.
- **targets** (One of more `iris.cube.Cube`, optional) – Target grid cube(s), utilised in binary operations. See the note below on providing suitable targets.
- **split** (*tuple, optional*) – Defines how each dimension should be split. If not specified, a suitable value is derived (can also be specified via the configuration file).
- **framework** (`ants.decomposition.DomainDecompose` object, optional) – Chosen decomposition framework, whether serial via `ants.decomposition.DomainDecompose` (default), multiprocessing via `ants.decomposition.MultiprocessingDomainDecompose` or multi-machine parallelism via `ants.decomposition.IPythonParallelDomainDecompose`. The framework can also be specified via the configuration files via section 'decomposition', option 'framework'.

Returns: Result from applying source and optional targets to the specified operation via the decomposition framework.

Return type: One or more `iris.cube.Cube`

Notes

When providing multiple sources, one of the following rules must hold true for the specified targets variable: * len(targets) == 0 * len(targets) == 1 * len(targets) == len(sources)

When these conditions are not met, the relationship between source and target is ambiguous and an exception is thrown.

Advanced Usage

The following section provides a means to gain greater control over the decomposition framework as well as provide a greater depth of understanding to how the components which make up the framework function.

Mosaics

There are numerous ways in which one could define how to generate a mosaic. The following abstract class defines the interface which all mosaic generator factory classes must implement (i.e. all custom mosaic class generator factories must inherit from this abstract base class `CallableMosaic`):

class `ants.decomposition.CallableMosaic` [source]

Abstract mosaic generator factory.

`__call__()` [source]

Mosaic generator.

Called each time we require a slice through our sliceable object. Particularly beneficial where we are restricted by memory.

Returns: Mosaic generator of sliceable pieces.

`__init__`

x.__init__(...) initializes x; see help(type(x)) for signature

The following approach generates a mosaic by specifying the resulting shape of the mosaic pieces.

class `ants.decomposition.MosaicByShape`(*sliceable*, *shape*) [source]

Mosaic generator factory where mosaic piece size is determine by the specified shape.

For example:

```
>>> arr = np.array([[1, 2], [3, 4]])
>>> splitter = MosaicByShape(arr, (1, 2))
>>> # Return our generator
>>> slices = splitter()
>>> print list(slices)
[array([[1, 2]]), array([[3, 4]])]
```

`__call__`() [source]

`__init__`(*sliceable*, *shape*) [source]

Mosaic generator factory for the given sliceable with specified target shape.

Parameters:
- **sliceable** (*sliceable object*) – Object with shape property and numpy style indexing. Mosaic pieces correspond to pieces of this given sliceable object.
- **shape** (*tuple*) – Resulting shape of each mosaic piece.

`shape`

tuple – The shape of this mosaic piece.

The alternative is to request decomposition based on the number of slices or pieces for each dimension.

class `ants.decomposition.MosaicBySplit`(*sliceable*, *split*) [source]

Mosaic generator factory where mosaic piece size is determined by the number of pieces requested for the mosaic.

For example:

```
>>> import numpy as np
>>> arr = np.array([[1, 2], [3, 4]])
>>> splitter = MosaicBySplit(arr, (2, 1))
>>> # Return our generator
>>> slices = splitter()
>>> print list(slices)
[array([[1, 2]]), array([[3, 4]])]
```

`__call__`() [source]

`__init__`(*sliceable*, *split*) [source]

Mosaic generator factory for the given sliceable with specified target shape.

Parameters:
- **sliceable** (*sliceable object*) – Object with shape property and numpy style indexing. Mosaic pieces correspond to a pieces of this given sliceable object.
- **split** (*tuple*) – Specified how each dimension should be split.

`split`

tuple – Number of split corresponding to each dimension.

Decomposition

Serial

When utilising serial decomposition, the `DomainDecompose` class gives us access to a decomposition framework with minimal effort.

class `ants.decomposition.DomainDecompose` [source]

Domain decompose an operation for a given cube for both unary and binary operations.

`__call__`(*operation*, *mosaics*, *sources=None*) [source]

Perform the provided operation over each decomposed piece.

Parameters:
- **operation** (*callable*) – Binary or unary operation on which to apply over each decomposed piece.
- **mosaic** (`CallableMosaic` object) – Callable which returns a generator of cubes.
- **sources** (One or more `iris.cube.Cube` , optional) – Source cube(s), where an extracted overlap is performed with each target piece in order to perform our binary operation.

`__init__`() [source]

Create a decomposable cube wrapper to which we can apply unary and binary operations.

`mosaic_generator`

iterator – An iterator over all the mosaic pieces.

`src_generator`

list of `Cube` – The source which overlaps each decomposed target piece.

Single-machine parallelism

Also available is a `multiprocessing` implementation of the decomposition framework, which has the same calling interface as `DomainDecompose`. Each piece within the decomposition is then operated upon on a separate process.

class `ants.decomposition.MultiprocessingDomainDecompose` [source]

Domain decompose an operation in parallel for a given cube for both unary and binary operations. Utilises `multiprocessing.Pool`.

`__call__`(*operation*, *mosaics*, *sources=None*)

Perform the provided operation over each decomposed piece.

Parameters:
- **operation** (*callable*) – Binary or unary operation on which to apply over each decomposed piece.
- **mosaic** (`CallableMosaic` object) – Callable which returns a generator of cubes.
- **sources** (One or more `iris.cube.Cube`, optional) – Source cube(s), where an extracted overlap is performed with each target piece in order to perform our binary operation.

`__init__`()

Create a decomposable cube wrapper to which we can apply unary and binary operations.

`mosaic_generator`

iterator – An iterator over all the mosaic pieces.

`src_generator`

list of `Cube` – The source which overlaps each decomposed target piece.

Multi-machine parallelism

We can extend capability further by utilising `ipython.parallel` via the `ants.decomposition.IPythonParallelDomainDecompose` interface. This allows us to utilise both single machine or multiple machine parallelism.

class `ants.decomposition.IPythonParallelDomainDecompose` [source]

Domain decompose an operation in parallel for a given cube for both unary and binary operations. Utilises `IPython.parallel`.

`__call__`(*operation*, *mosaics*, *sources=None*)

Perform the provided operation over each decomposed piece.

Parameters:
- **operation** (*callable*) – Binary or unary operation on which to apply over each decomposed piece.
- **mosaic** (`CallableMosaic` object) – Callable which returns a generator of cubes.
- **sources** (One or more `iris.cube.Cube`, optional) – Source cube(s), where an extracted overlap is performed with each target piece in order to perform our binary operation.

`__init__`()

Create a decomposable cube wrapper to which we can apply unary and binary operations.

`mosaic_generator`

iterator – An iterator over all the mosaic pieces.

`src_generator`

list of `Cube` – The source which overlaps each decomposed target piece.

Again, this has the same calling interface as `DomainDecompose`. See the IPython userguide on how to get started.

Example use-case

Unary Example

Let's begin by defining a unary function:

```
def operation_unary(a):
    return a + 1
```

Now define a sample cube:

```
import iris.tests.stock as stock
source = stock.lat_lon_cube()
```

This is how we might call our function with this `Cube` without decomposition:

```
res = operation_unary(source)
```

To repeat the process utilising decomposition, first import the relevant module:

```
import ants.decomposition as decomp
```

Now, perform the operation utilising decomposition. We define how we want to generate our pieces, here splitting by our second dimension into two pieces (see):

```
mosaic = decomp.MosaicBySplit(source, (1, 2))
```

Now we must choose our decomposition framework and then call this with our operation, using our mosaic factory generator (mosaic):

```
domain_decomposer = decomp.DomainDecompose()
res = domain_decomposer(operation_unary, mosaic)
```

Similarly we could have specified our mosaic by a specified shape, where the following specifies that the shape of each piece is (1, 2):

```
mosaic = decomp.MosaicByShape(source, (1, 2))
```

Similarly, to utilise parallel decomposition, is as simple as choosing the alternative framework and then calling it in the same way:

```
domain_decomposer = decomp.MultiprocessingDomainDecompose()
res = domain_decomposer(operation_unary, mosaic)
```

Binary Example

Now let's define a binary function:

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

This time we will need to define a target grid:

```
target = stock.lat_lon_cube()
```

This is how we would call this without decomposition:

```
res = operation_binary(source, target)
```

Now let's utilise decomposition in exactly the same way as we did in the unary case, passing this time both our target mosaic and the source:

```
import ants.decomposition as decomp

domain_decomposer = decomp.DomainDecompose()
mosaic = decomp.MosaicBySplit(target, (1, 2))
res = domain_decomposer(operation_binary, mosaic, source)
```

Extended Usage

In some cases your processing function may have arguments other than the source and/or target cubes. The python function `functools.partial()` can be used to help you turn these processing functions into a form usable by the decomposition module.

A simple example of this is shown here:

```
from functools import partial
import ants.decomposition as decomp

def operation_unary(a, base=0):
    return a + 1 + base

mosaic = decomp.MosaicByShape(source, (1, 2))
domain_decomposer = decomp.DomainDecompose()

# set the values of the keyword argument for decomposition framework
process_func = partial(operation_unary, base=10)
res = domain_decomposer(process_func, mosaic)
```

Quick-start interface

The following shows a comparison between the interfaces as a means to providing a quick reference to getting started.

No decomposition:

```
res = operation_unary(source)
```

Decomposition (serial):

```
import ants.decomposition as decomp

mosaic = decomp.MosaicByShape(source, (1, 2))
domain_decomposer = decomp.DomainDecompose()
res = domain_decomposer(mosaic)
```

Decomposition (parallel - single host):

```
import ants.decomposition as decomp

mosaic = decomp.MosaicByShape(source, (1, 2))
domain_decomposer = decomp.MultiprocessingDomainDecompose()
res = domain_decomposer(mosaic)
```