A dive into the heart of S1Tiling

a python orchestrator dedicated to OTB

OTB User Days, 21 November 2024

Luc Hermitte (CS Group FRANCE)

Agenda

1 | The original design

2 | Goals turned into features

3 | Zoom into the kernel

4 | What's next?

01

The original design

THE ORIGINAL DESIGN

WORKFLOW

0. Downloads S1 products on PEPS
1. Scans for all S1 products
2. Calibrates all S1 products
3. Cuts margins on all calibrated images
4. Orthorectifies calibrated and cut images to target S2 MGRS tiles
5. Assembles up-to 2 orthorectified images per target tile

THE ORIGINAL DESIGN

ISSUES

- High I/O usage
  - Many `glob`s, at each step
  - Numerous undeleted files
  - Reliance on GPFS if not cautious
  - A file is produced after each OTB application
- Complex stop/start-over
  - Incomplete image files from previous executions
  - Reliance on `glob`...

02

Goals turned into features

GOALS

- Industrialize
  - Reduce I/O
  - Handle start-over
  - Be efficient
  - Handle GeoTIFF metadata
  - Support any S1 data provider
  - Simplify installation
  - Document
- Easy to evolve, and reuse

GOALS TURNED INTO FEATURES

IMPROVE I/O

- Reduce I/O usage
  - Cache known filenames instead of globbing them
  - Use *in-memory* pipelines from OTB Python API
- Automate removal of old files
- Work on local SSD instead of GPFS
  - Optional copy of static files (DEM, Geoid...)
  - But jobs on HPC clusters need to correctly set working directories

GOALS TURNED INTO FEATURES

START-OVER

- Never regenerate what we don't need anymore
  - Yet a `glob` on input products is required at start-up
- Distinguish incomplete `.tiff` images from complete ones.

GOALS TURNED INTO FEATURES

EFFICIENCY

- All the above, plus
- Parallelise processings
- Factorise common steps

GOALS TURNED INTO FEATURES

OTHER FEATURES

- Automated setting of GeoTIFF metadata
- Support of any data provider thanks to EODAG
- Documented
- S1Tiling packet on pypi, and module on TREX
- Ready-to-use dockers

03

Zoom into the kernel

ZOOM INTO THE KERNEL

THE CORE IDEAS

- *In-memory* pipelines of OTB Applications
- Processing chains *à la make*

ZOOM INTO THE KERNEL

THE CORE IDEAS: TASK DEPENDENCY ANALYSIS

Given all existing inputs,

Two passes:

1. build graph of all possible flows
2. trim unrequired tasks

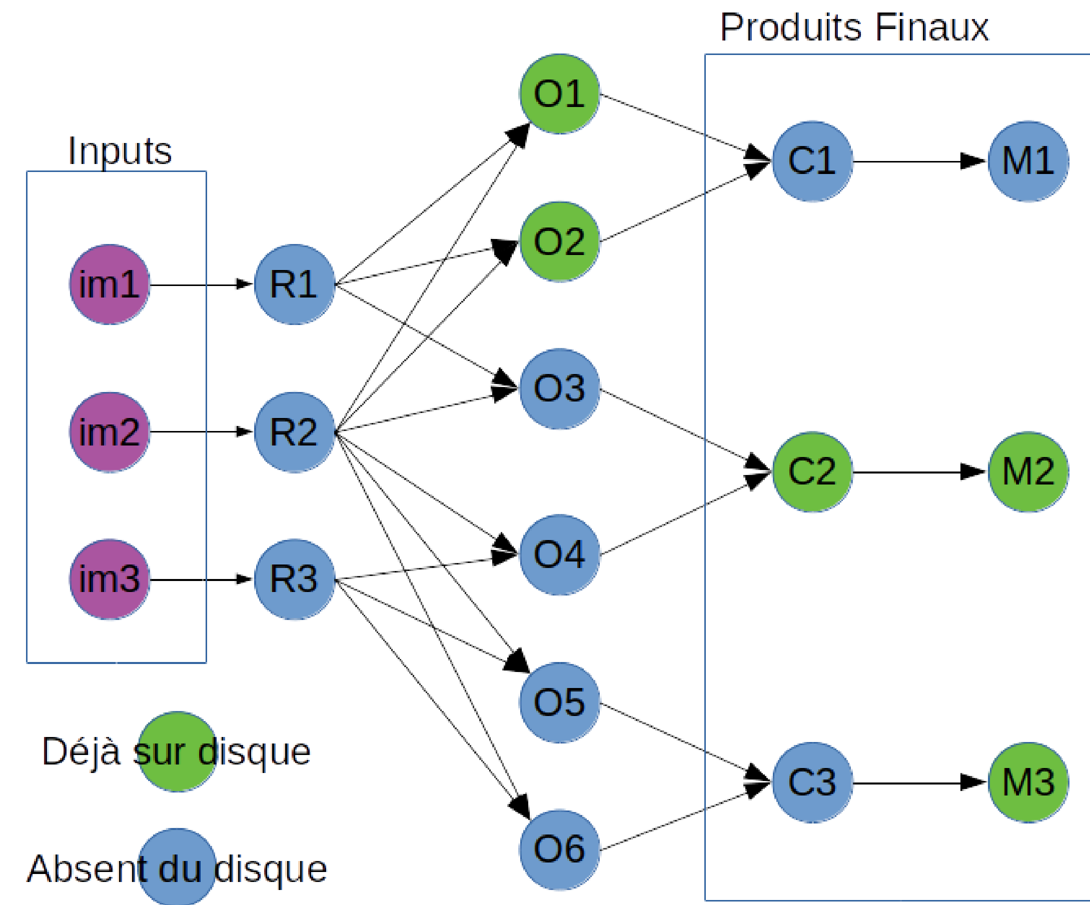
ZOOM INTO THE KERNEL

THE CORE IDEAS: TASK DEPENDENCY ANALYSIS

Given all existing inputs,

Two passes:

1. build graph of all possible flows
2. trim unrequired tasks



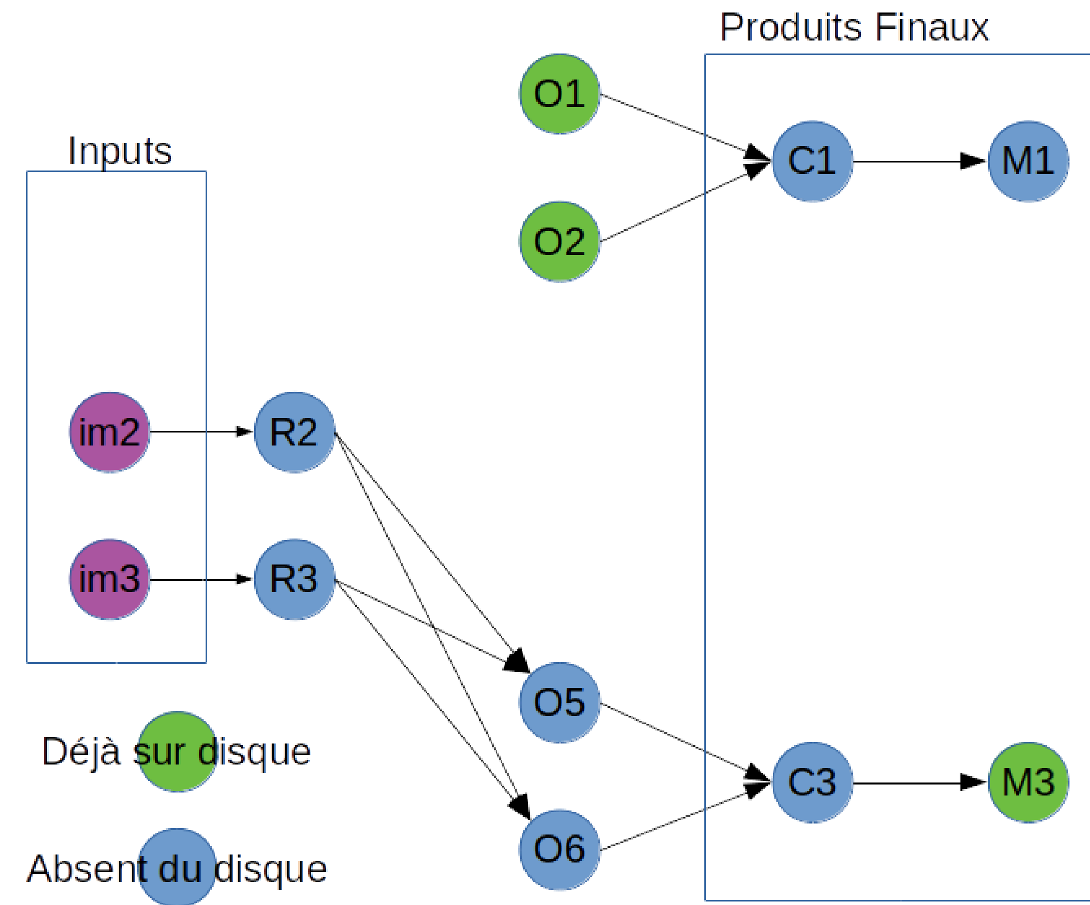
ZOOM INTO THE KERNEL

THE CORE IDEAS: TASK DEPENDENCY ANALYSIS

Given all existing inputs,

Two passes:

1. build graph of all possible flows
2. trim unrequired tasks



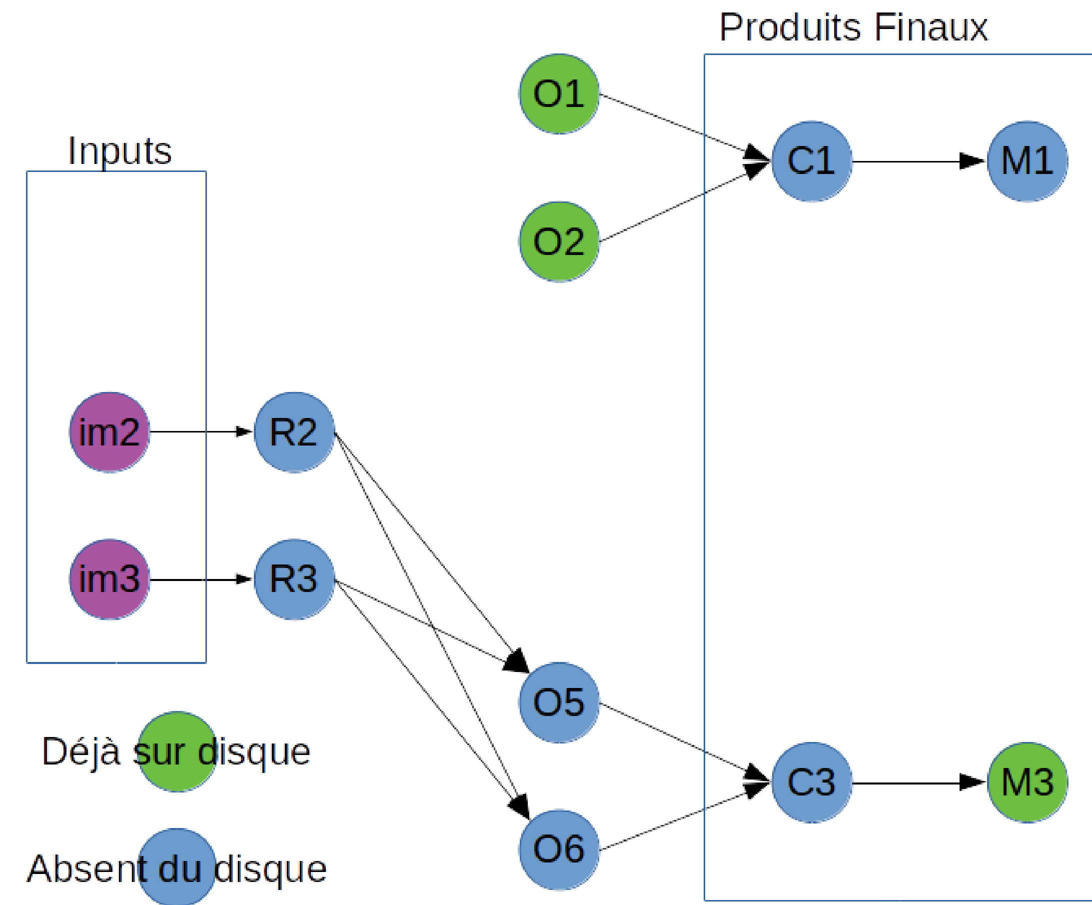
ZOOM INTO THE KERNEL

THE CORE IDEAS: TASK DEPENDENCY ANALYSIS

Given all existing inputs,

Two passes:

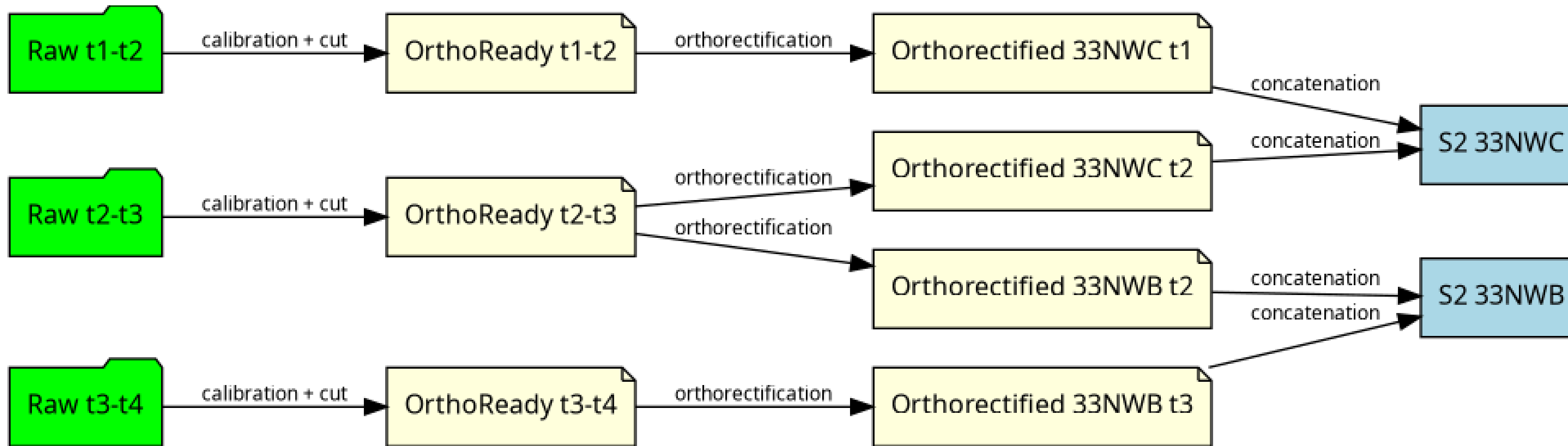
1. build graph of all possible flows
2. trim unrequired tasks



Eventually we have a DAG that we can pass to Dask

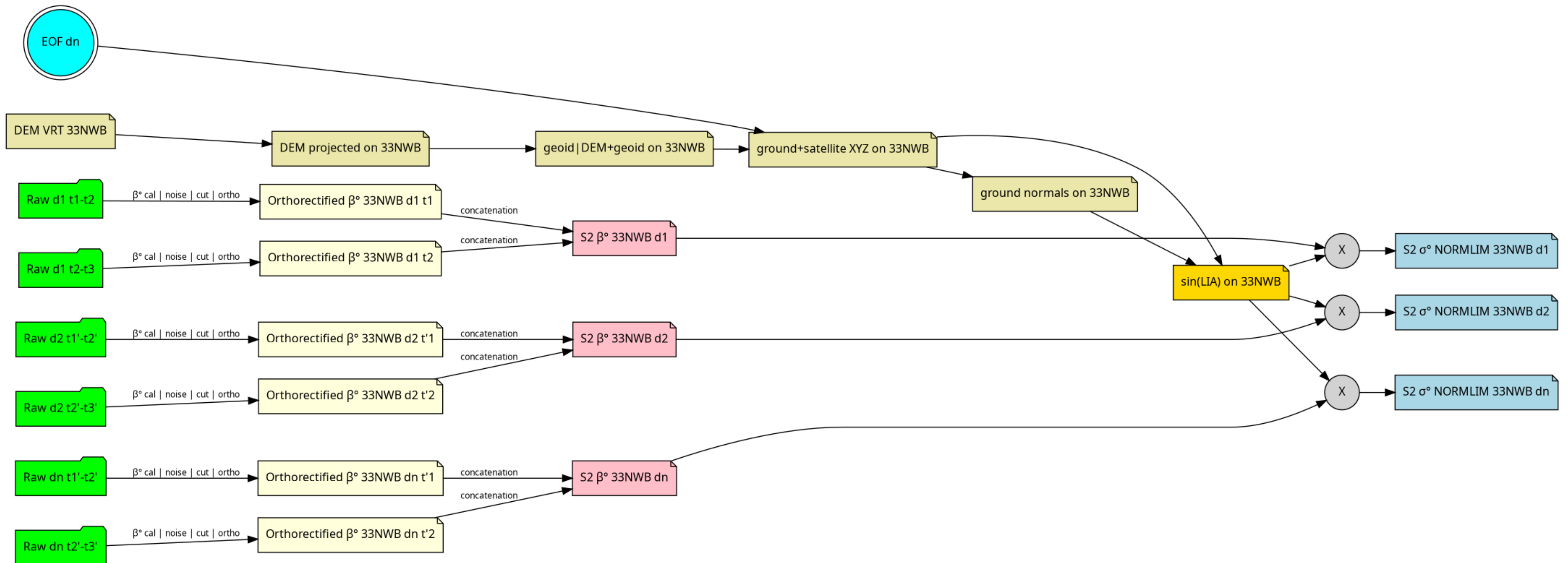
ZOOM INTO THE KERNEL

PIPELINE EXAMPLE: THE STRAIGHTFORWARD ONE



ZOOM INTO THE KERNEL

PIPELINE EXAMPLE: A BIT MORE COMPLEX ONE



ZOOM INTO THE KERNEL

HIGH LEVEL API

- Processings are done at each `Step`
  - Exact `Steps` are instantiated by `StepFactories`
    - Cut, Calibrate, FixThermalNoiseRemoval,
    - Orthorectification, Concatenation...
  - They are assembled into *pipelines*
- Exact starting points are instantiated by `FirstStepFactories`
  - S1 Products
  - Precise orbit files
  - S2 MGRS information...

ZOOM INTO THE KERNEL

HIGH LEVEL API

- Processings are done at each `Step`
  - Exact `Step`s are instantiated by `StepFactories`
    - Cut, Calibrate, FixThermalNoiseRemoval,
    - Orthorectification, Concatenation...
  - They are assembled into *pipelines*
- Exact starting points are instantiated by `FirstStepFactories`
  - S1 Products
  - Precise orbit files
  - S2 MGRS information...

Kernel is (now) independent of any Sentinel-1/S1Tiling specificities

ZOOM INTO THE KERNEL

REGISTRATION EXAMPLES

THE STRAIGHTFORWARD CASE

First we have to manually define the various factories, then their sequencing

```
1  pipelines = PipelineDescriptionSequence(config)
2
3  # Register input factory
4  pipelines.register_inputs('basename', s1_raster_first_inputs_factory)
5
6  # Register Step Factories: Calibratation ... Concatenation
7  calib_seq = [ExtractSentinel1Metadata, AnalyseBorders, Calibrate,
8               CorrectDenoising, CutBorders]
9
10 pipelines.register_pipeline(calib_seq,      'PrepareForOrtho',
11                             product_required=False, is_name_incremental=True)
12 pipelines.register_pipeline([OrthoRectify], 'OrthoRectify',
13                             product_required=False)
14
15 pipelines.register_pipeline([Concatenate], product_required=True, is_name_incremental=Tru
```

A *pipeline* is a sequence of steps, and pipelines are also sequenced between themselves

ZOOM INTO THE KERNEL

REGISTRATION EXAMPLES

THE STRAIGHTFORWARD CASE

First we have to manually define the various factories, then their sequencing

```
 1 pipelines = PipelineDescriptionSequence(config)
 2
 3 # Register input factory
 4 pipelines.register_inputs('basename', s1_raster_first_inputs_factory)
 5
 6 # Register Step Factories: Calibratation ... Concatenation
 7 calib_seq = [ExtractSentinel1Metadata, AnalyseBorders, Calibrate,
 8              CorrectDenoising, CutBorders]
 9
10 pipelines.register_pipeline(calib_seq,      'PrepareForOrtho',
11                             product_required=False, is_name_incremental=True)
12 pipelines.register_pipeline([OrthoRectify], 'OrthoRectify',
13                             product_required=False)
14
15 pipelines.register_pipeline([Concatenate], product_required=True, is_name_incremental=Tru
```

A *pipeline* is a sequence of steps, and pipelines are also sequenced between themselves

ZOOM INTO THE KERNEL

REGISTRATION EXAMPLES

THE STRAIGHTFORWARD CASE

First we have to manually define the various factories, then their sequencing

```
1  pipelines = PipelineDescriptionSequence(config)
2
3  # Register input factory
4  pipelines.register_inputs('basename', s1_raster_first_inputs_factory)
5
6  # Register Step Factories: Calibratation ... Concatenation
7  calib_seq = [ExtractSentinel1Metadata, AnalyseBorders, Calibrate,
8               CorrectDenoising, CutBorders]
9
10 pipelines.register_pipeline(calib_seq,      'PrepareForOrtho',
11                             product_required=False, is_name_incremental=True)
12 pipelines.register_pipeline([OrthoRectify], 'OrthoRectify',
13                             product_required=False)
14
15 pipelines.register_pipeline([Concatenate], product_required=True, is_name_incremental=Tru
```

A *pipeline* is a sequence of steps, and pipelines are also sequenced between themselves

ZOOM INTO THE KERNEL

REGISTRATION EXAMPLES

OR THE BIT MORE COMPLEX CASE

```
1 pipelines = PipelineDescriptionSequence(config)
2
3 pipelines.register_inputs('tilename', tilename_first_inputs_factory)
4 pipelines.register_inputs('eof', eof_first_inputs_factory)
5
6 dem_vrt = pipelines.register_pipeline([AgglomerateDEMOnS2], 'AgglomerateDEM',
7                                       inputs={'tilename': 'tilename'})
8
9 s2_dem = pipelines.register_pipeline([ProjectDEMToS2Tile], "ProjectDEMToS2Tile",
10                                     inputs={"indem": dem_vrt})
11
12 s2_height = pipelines.register_pipeline([ProjectGeoidToS2Tile, SumAllHeights], "GenerateH
13                                        inputs={"in_s2_dem": s2_dem})
14
15 xyz = pipelines.register_pipeline([ComputeGroundAndSatPositionsOnDEMFromEOF], "ComputeGro
16                                   inputs={'ineof': 'eof', 'inheight': s2_height})
17
18 lia = pipelines.register_pipeline([ComputeNormalsOnS2, ComputeLIAOnS2], 'ComputeLIAOnS2',
```

ZOOM INTO THE KERNEL

REGISTRATION EXAMPLES

OR THE BIT MORE COMPLEX CASE

```
1  pipelines = PipelineDescriptionSequence(config)
2
3  pipelines.register_inputs('tilename', tilename_first_inputs_factory)
4  pipelines.register_inputs('eof', eof_first_inputs_factory)
5
6  dem_vrt = pipelines.register_pipeline([AgglomerateDEMOnS2], 'AgglomerateDEM',
7                                        inputs={'tilename': 'tilename'})
8
9  s2_dem = pipelines.register_pipeline([ProjectDEMToS2Tile], "ProjectDEMToS2Tile",
10                                      inputs={"indem": dem_vrt})
11
12 s2_height = pipelines.register_pipeline([ProjectGeoidToS2Tile, SumAllHeights], "GenerateH
13                                         inputs={"in_s2_dem": s2_dem})
14
15 xyz = pipelines.register_pipeline([ComputeGroundAndSatPositionsOnDEMFromEOF], "ComputeGro
16                                   inputs={'ineof': 'eof', 'inheight': s2_height})
17
18 lia = pipelines.register_pipeline([ComputeNormalsOnS2, ComputeLIAOnS2], 'ComputeLIAOnS2',
```

ZOOM INTO THE KERNEL

REGISTRATION EXAMPLES

OR THE BIT MORE COMPLEX CASE

```python

pipelines.register_inputs('tilename', tilename_first_inputs_factory)
pipelines.register_inputs('eof', eof_first_inputs_factory)

dem_vrt = pipelines.register_pipeline([AgglomerateDEMOnS2], 'AgglomerateDEM',
                                      inputs={'tilename': 'tilename'})

s2_dem = pipelines.register_pipeline([ProjectDEMToS2Tile], "ProjectDEMToS2Tile",
                                     inputs={"indem": dem_vrt})

s2_height = pipelines.register_pipeline([ProjectGeoidToS2Tile, SumAllHeights], "GenerateH
                                        inputs={"in_s2_dem": s2_dem})

xyz = pipelines.register_pipeline([ComputeGroundAndSatPositionsOnDEMFromEOF], "ComputeGro
                                  inputs={'ineof': 'eof', 'inheight': s2_height})

lia = pipelines.register_pipeline([ComputeNormalsOnS2, ComputeLIAOnS2], 'ComputeLIAOnS2',
                                  inputs={'xyz': xyz}, product_required=True)
```

ZOOM INTO THE KERNEL

THE BINDING CODE

```
1  # The preparation seen earlier
2  config = ...
3  pipelines = ...
4
5  # Extra preparation step: the input factories need some data
6  pipelines.register_extra_parameters_for_input_factories(
7      dag=dag,                            # EODAG object used by s1_raster_first_inputs_factor
8      s1_file_manager=s1_file_manager, # Main object in charge of DL S1 products by the fac
9  )
10
11 # Now let's execute with Dask!
12 results : List[Outcome] = []
13 with s1tiling.libs.utils.dask.DaskContext(config) as dask_client:
14     for idx, tile_it in enumerate(tiles_to_process):
15         res = process_one_tile(  # Very S1Tiling specific: handle one S2 tile
16             tile_it, idx, nb_tiles,
17             config,
18             pipelines,
```

ZOOM INTO THE KERNEL

THE BINDING CODE

```
10
11 # Now let's execute with Dask!
12 results : List[Outcome] = []
13 with s1tiling.libs.utils.dask.DaskContext(config) as dask_client:
14     for idx, tile_it in enumerate(tiles_to_process):
15         res = process_one_tile(  # Very S1Tiling specific: handle one S2 tile
16             tile_it, idx, nb_tiles,
17             config,
18             pipelines,
19             dask_client.client,
20         )
21             # ensure_tiled_workspaces_exist(cfg, tile_name, required_workspaces)
22             # pipelines.register_extra_parameters_for_input_factories(tile_name=tile_name
23             # dsk, required_products, errors = pipelines.generate_tasks()
24             # if errors:
25             #     return errors
26             # return client.get(dsk, required_products)
27         results.extend(res)
28
```

ZOOM INTO THE KERNEL

THE BINDING CODE

```
19                dask_client.client,
20            )
21                # ensure_tiled_workspaces_exist(cfg, tile_name, required_workspaces)
22                # pipelines.register_extra_parameters_for_input_factories(tile_name=tile_name
23                # dsk, required_products, errors = pipelines.generate_tasks()
24                # if errors:
25                #     return errors
26                # return client.get(dsk, required_products)
27            results.extend(res)
28
29 # Reporting results
30 nb_issues = sum(not bool(res) for res in results)
31 if nb_errors_detected > 0:
32     logger.warning('Execution report: %s errors detected', nb_issues)
33 else:
34     logger.info('Execution report: no error detected')
35 for res in results:
36     logger.log(log_level(res), ' - %s', res)
```

ZOOM INTO THE KERNEL

HIGH LEVEL API USAGE

We define:

- a domain aware `StepFactory` per processing (OTB app, Python function, External process)
- a `PipelineDescription` for a sequence of one or several steps.
  - Multiple OTB steps will be chained in memory automatically
  - All the `PipelineDescription` instances are registered in a `PipelineDescriptionSequence`

Then the kernel will take care of:

- Searching all possible inputs thanks to the `FirstStepFactories`.
- ...that'll be used to build the reduced DAG of Dask Tasks

ZOOM INTO THE KERNEL

HIGH LEVEL API USAGE: `StepFactories`

A concrete and specialized `StepFactory`

- is configured from the execution configuration object
- relies on input products *metadatas* to
  - define what the output filename would be
  - determine the execution parameters -- according to the kind of `StepFactory`
  - determine what GeoTIFF metadata should be written in the product

ZOOM INTO THE KERNEL

HIGH LEVEL API USAGE

A SIMPLE StepFactory EXAMPLE

```
 1 class Calibrate(s1tiling.libs.otbpipeline.OTBStepFactory):
 2     def __init__(self, cfg: Configuration) -> None:
 3         fname_fmt = cfg.fname_fmt.get('calibration', '{rootname}_{calibration_type}_calOk
 4         super().__init__(cfg,
 5                 appname='SARCalibration',
 6                 name='Calibration',
 7                 gen_tmp_dir=os.path.join(cfg.tmpdir, 'S1'),
 8                 gen_output_filename=TemplateOutputFilenameGenerator(fname_fmt),
 9                 image_description='{calibration_type} calibrated Sentinel-{flying_unit_co
10         )
11         self.__calibration_type   = cfg.calibration_type
12         self.__removethermalnoise = cfg.removethermalnoise
13 
14     def _update_filename_meta_pre_hook(self, meta: Meta) -> Meta:
15         meta['calibration_type'] = self.__calibration_type
16         return meta
17 
18     def update_image_metadata(self, meta: Meta, all_inputs: InputList) -> None:
```

ZOOM INTO THE KERNEL

HIGH LEVEL API USAGE

A SIMPLE StepFactory EXAMPLE

```
1  class Calibrate(s1tiling.libs.otbpipeline.OTBStepFactory):
2      def __init__(self, cfg: Configuration) -> None:
3          fname_fmt = cfg.fname_fmt.get('calibration', '{rootname}_{calibration_type}_calOk
4          super().__init__(cfg,
5                  appname='SARCalibration',
6                  name='Calibration',
7                  gen_tmp_dir=os.path.join(cfg.tmpdir, 'S1'),
8                  gen_output_filename=TemplateOutputFilenameGenerator(fname_fmt),
9                  image_description='{calibration_type} calibrated Sentinel-{flying_unit_co
10         )
11         self.__calibration_type   = cfg.calibration_type
12         self.__removethermalnoise = cfg.removethermalnoise
13
14     def _update_filename_meta_pre_hook(self, meta: Meta) -> Meta:
15         meta['calibration_type'] = self.__calibration_type
16         return meta
17
18     def update_image_metadata(self, meta: Meta, all_inputs: InputList) -> None:
```

ZOOM INTO THE KERNEL

HIGH LEVEL API USAGE

A SIMPLE `StepFactory` EXAMPLE

```
1  class Calibrate(s1tiling.libs.otbpipeline.OTBStepFactory):
2      def __init__(self, cfg: Configuration) -> None:
3          fname_fmt = cfg.fname_fmt.get('calibration', '{rootname}_{calibration_type}_calOk
4          super().__init__(cfg,
5                  appname='SARCalibration',
6                  name='Calibration',
7                  gen_tmp_dir=os.path.join(cfg.tmpdir, 'S1'),
8                  gen_output_filename=TemplateOutputFilenameGenerator(fname_fmt),
9                  image_description='{calibration_type} calibrated Sentinel-{flying_unit_co
10         )
11         self.__calibration_type    = cfg.calibration_type
12         self.__removethermalnoise = cfg.removethermalnoise
13
14     def _update_filename_meta_pre_hook(self, meta: Meta) -> Meta:
15         meta['calibration_type'] = self.__calibration_type
16         return meta
17
18     def update_image_metadata(self, meta: Meta, all_inputs: InputList) -> None:
```

ZOOM INTO THE KERNEL

HIGH LEVEL API USAGE

A SIMPLE `StepFactory` EXAMPLE

```
12          self.__removethermalnoise = cfg.removethermalnoise
13
14      def _update_filename_meta_pre_hook(self, meta: Meta) -> Meta:
15          meta['calibration_type'] = self.__calibration_type
16          return meta
17
18      def update_image_metadata(self, meta: Meta, all_inputs: InputList) -> None:
19          super().update_image_metadata(meta, all_inputs)
20          assert 'image_metadata' in meta
21          imd = meta['image_metadata']
22          imd['CALIBRATION']    = str(self.__calibration_type)
23          imd['NOISE_REMOVED'] = str(self.__removethermalnoise)
24
25      def parameters(self, meta: Meta) -> OTBParameters:
26          params : OTBParameters = {
27                  'ram'          : ram(self.ram_per_process),
28                  self.param_in : s1tiling.libs.meta.in_filename(meta),
29                  'lut'          : self.__calibration_type,
```

ZOOM INTO THE KERNEL

HIGH LEVEL API USAGE

A SIMPLE StepFactory EXAMPLE

```
15          meta['calibration_type'] = self.__calibration_type
16          return meta
17
18      def update_image_metadata(self, meta: Meta, all_inputs: InputList) -> None:
19          super().update_image_metadata(meta, all_inputs)
20          assert 'image_metadata' in meta
21          imd = meta['image_metadata']
22          imd['CALIBRATION']   = str(self.__calibration_type)
23          imd['NOISE_REMOVED'] = str(self.__removethermalnoise)
24
25      def parameters(self, meta: Meta) -> OTBParameters:
26          params : OTBParameters = {
27                  'ram'           : ram(self.ram_per_process),
28                  self.param_in : s1tiling.libs.meta.in_filename(meta),
29                  'lut'           : self.__calibration_type,
30                  'removenoise' : self.__removethermalnoise,
31          }
32          return params
```

ZOOM INTO THE KERNEL

LET'S DIVE A BIT MORE

- Dask task nodes are actually filenames (not image rasters)
- and the edges are `Pipeline` instantiated from:
  - `PipelineDescriptionSequence`
  - plus input metadata
- When Dask executes a `Pipeline`, it instanciates generic `Step` instances
- The `Steps` will generate their output as per specified in the related `StepFactory`, and also update GeoTIFF metadata.

ZOOM INTO THE KERNEL

LET'S DIVE A BIT MORE

- Dask task nodes are actually filenames (not image rasters)
- and the edges are `Pipeline` instantiated from:
  - `PipelineDescriptionSequence`
  - plus input metadata
- When Dask executes a `Pipeline`, it instanciates generic `Step` instances
- The `Steps` will generate their output as per specified in the related `StepFactory`, and also update GeoTIFF metadata.

But none of it is your concern

04

What's next?

WHAT'S NEXT?

- New chains in S1Tiling (RTC, IA...)
- A new processing chain: SLC Time Series
  - => Extract S1Tiling kernel into its own packet
- YAML description of pipelines

Q&A