Dask-based libRadtran Server

CJ Willers

nwillers@csir.co.za
neliswillers@gmail.com

April 2020

Contents

List of Figures

ABBREVIATIONS

BPDF Bidirectional Polarization Distribution Functions

GB Gigabyte

GIL Global Interpreter Lock

PC Personal Computer

URI Uniform Resource Identifier

VPN Virtual Private Network

Chapter 1

Introduction

1.1 Overview

libRadtran [2, 3] is a free C- and Fortran-based collection of programs and libraries for the calculation of radiative transfer and solar and thermal radiation in the earth's atmosphere. The concept of a network-based libRadtran server allows the distributed deployment of the libRadtran functionality. The server approach allows easy access for any number of users[1] and also supports deployment on clusters with many computational nodes.

Dask [1] is a library for parallel computing in Python that works on a local computer, a remote computer and cluster computers. Dask has two parts: (1) dynamic task scheduling for executing, and (2) fast processing of Python data structures (e.g., numpy, pandas, etc.). The main interest in Dask for this application is for distributed processing, with less interest in fast processing of Python structures. Using Dask requires only Python familiarity and can deploy tasks to large clusters with thousands of cores.

The libraddask is a Python module that integrates libRadtran with Dask with the aim to provide a remote and network-distributed libRadtran. This functionality is only available via Python, but this is not considered a major issue with most of the work flow is in Python.

For rapid shortwave spectrum work, the band parametrisation by Kato is typically used. The Kato model has only 32 spectral bands across the whole available spectrum and all computations are reduced to a vector of values representative of the Kato bands.

For more accurate work, such as simulations of satellite views of water targets, the REPTRAN model is preferred. REPTRAN provides a full spectral calculation or rapid band model calculations for a variety of satellite sensors.

libRadtran is able to calculate radiances and irradiances when the sun is below the horizon by up to 9 degrees or more. This is well into "nautical" twilight. The MYSTIC monte carlo solver is required to perform twilight computations.

[1] libRadtran is easiest to build on Linux and, when deployed on a Linux server, provide easy access to Windows users.

1.2 Historical Background

The libraddask library was originally written by Derek Griffith [4] as part of a larger project. The files pertaining to using Dask for distributed libRadtran were extracted as a stand-alone project. The work was originally done in Python 2 against the libRadtran 2.0.0 Python files.

This version ported all the code to Python 3 and uses the the libRadtran 2.0.3 Python files. No new substantial functionality is added. This present document was added by its author as a means to document the new project.

1.3 Code Availability

The libraddask package is available here [5]:
`https://github.com/NelisW/libraddask`

1.4 Example Case Study

For the practical example in this document, assume the following (will be different in your installation):

1. The Dask client will be activated in code run on a local computer using Windows. The Dask scheduler and workers will be running on a remote server using Linux.
2. Server (Linux):
   a) The server is called `nimbus` with **IP!** (**IP!**) address `146.64.246.94`
   b) The work is done from user account `dgriffith` on the server.
   c) The Python and dask packages are set up in the `mordevpy37` conda environment on the server.
3. Local Personal Computer (PC) (Windows):
   a) The Python and dask packages are set up in the `mort` conda environment on the local computer.

1.5 Document Structure

Chapter 2 provides an overview of Dask, on a conceptual and practical basis. Understanding of these concepts are important for effectively using the server.

Chapter 3 provides and overview of libRadtran, including obtaining, installation and simple example usage.

Chapter 4 provides instructions on how to set up libraddask.

Chapter 5 contains and extract of an example Jupyter notebook on how to use libraddask.

Chapter 6 concludes the document.

Appendix A contains additional Dask examples.

Appendix B contains a description of the porting of the libRadtran Python files in `scr_py` to Python 3.

Appendix C contains additional information on tools that may be useful when using libraddask.

Chapter 2

Dask

2.1 Overview

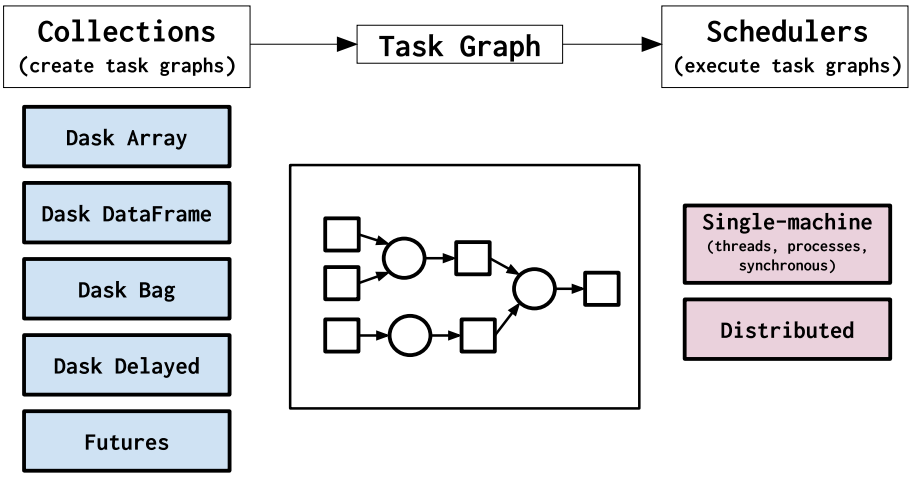


Figure 2.1: Dask functionality [1]

Dask [1] is a flexible library for parallel computing in Python. It is composed of these parts:

1. Scheduler: Dynamic task scheduling optimized for computation. This is similar to Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
2. Graphs: Dask represents parallel computations with task graphs. Working directly with dask graphs is rare, unless you intend to develop new modules with Dask. Dask graphs are created internally and most users don't ever see a graph. See Section A.1 for an example of a Dask graph.
3. Collections: "Big Data" collections like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

The graph and collections functionalities are very powerful but not used in the libraddask module. These topics are not further considered here.

Dask installs conveniently on a laptop, a desktop or a cluster computer. Dask can scale to a cluster of 100s of machines. This ease of transition between single-machine to moderate cluster enables users to both start simple and grow when necessary. The libraddask module uses the Dask scheduler to run libRadtran on a Linux computer, serving locally or to remote computers.

2.2 Dask as Server

The client Client-Server design pattern is generally used in the context where any number of clients, running on many different (weaker) computers can access the centralised server that is highly optimised to perform a specific set of functions very efficiently. A common instance of this pattern is in enterprise database systems such as SAP or PeopleSoft. The clients have a small footprint (called 'thin', perhaps even just a browser), with most of the functionality in the main server (often with big databases and high processing capability).

A cloud service is any service made available to users on demand via the Internet from a cloud computing provider's servers as opposed to being provided from a company's own on-premises servers. Cloud services are designed to provide easy, scalable access to applications, resources and services, and are fully managed by a cloud services provider. In the context of this document, consider the cloud service as a massively parallel computer providing computing and/or storage via the internet.

Dask is neither of the above, it is a technology that enables parallel computation, either on the local computer, on other computers on a private network or even in a cloud service. Using the Dask distributed module a server-like functionality can be implemented for doing remote processing in a specialist area — this is what the current application is doing.

Setting up a distributed computation server with Dask requires that work on the local PC and work on the server be done in Python. Even more strictly, exactly the same versions of Python and all required modules must be installed on all computers. This requirement stems from Dask's packaging (using pickle or dill) of the code to be executed as well as the data to be used. The combined package of code and data is sent to the remote computer, which returns the resulting data after computation. This model is fundamentally different from conventional servers.

2.3 Dask `Distributed`

The Dask distributed concept has three major roles:

1. Clients: The client resides on a local PC (of any form) which puts together a package of code and data. This package is pickled[1] and sent to a scheduler. The client also receives results back from the scheduler. So the client sends task packages and retrieves results. There can be any number of clients, on any number of computers. One client can communicate with any number of schedulers.

2. Schedulers: The scheduler listens on a specific port for any client requests. At the same time, the scheduler also polls the workers to keep record of 'idle' workers that are available to take on new tasks. Task packages from clients are routed to available workers for execution. When the worker has completed the task, the scheduler can make it available

[1] Pickle is a Python way to serialise complex objects into a single package.

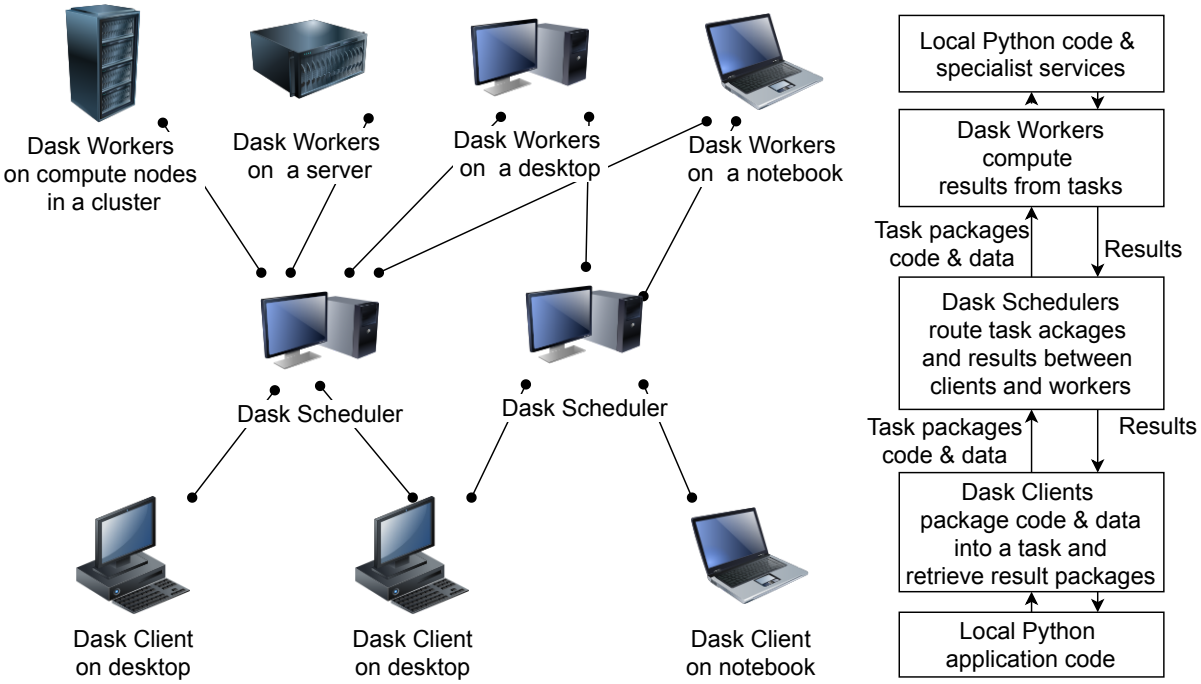


Figure 2.2: Dask distributed concept

when the client requests the results. There can be any number of schedulers, on any number of computers. One scheduler can communicate with any number of clients and any number of workers.

3. Workers: The workers wait for task assignments from the scheduler. When the task package is received it is executed on the worker node CPU. The results are packaged and stored until retrieved by the scheduler. There can be any number of workers, on any number of computing nodes or computers. One worker can communicate with any number of schedulers.

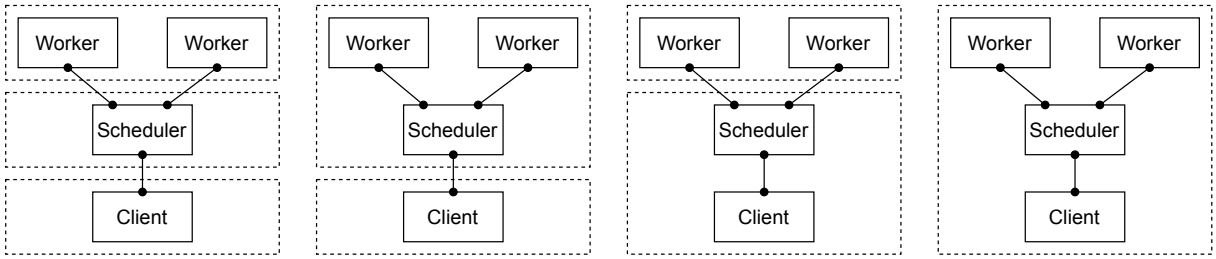


Figure 2.3: Dask distributed in different computers configurations. The dotted lines represent different computers

The Client, the Scheduler and the Worker are all independently running code instances, which could be running on one or more computers. Figure 2.3 shows the options for running the processes. The figure shows the computer hardware boundaries in dashed lines. It is evident that the different processes can run on the same computer, different computers or any mix of computers. Each computer is identified by an IP address on the network and the network location (IP addresses) of the scheduler and worker computers are specified in the code. Hence, the IP addresses must be known before the code is executed.

2.4 Executing User Code

How does this Dask Distributed fit into the user code? Imagine the scenario where a Windows user requires a libRadtran run, but does not have it compiled for his Windows PC. Suppose libRadtran is available on a Linux computer somewhere on the network. The Linux computer can be single computer in the office next door, or it can be a server on the local network or even in a cloud service.

The user Python application sets up a task package containing some Python code to run libRadtran on the remote computer and the input files for the run. The package and execution proceeds from user application code to the Dask client on the local PC, to the scheduler, to the worker (the worker then does some work on the data), the results are then sent back to the scheduler, back to the client, and finally back to the user code, where the results are processed. This process can be followed for a single libRadtran run or it can be for thousands of runs, where each run is executed on one of many different workers. The workers run concurrently on a cluster or on multiple cores in a local CPU.

Once the Dask distributed channel is set up, the user has no further action than to set up the task package. The distribution and execution is all taken care of by Dask. The price to be paid for this convenience is setting up the scheduler and worked environments, which is relatively small. Of course, setting up libRadtran must be done anyway, either on the local or remote computer, so there is no penalty here. In fact, building libRadtran on a Linux PC is relatively simple and quick.

How does the worker know what processing is required? The package delivered to the worker contains Python code and the input data. The worker simply executes the Python code in the package, on the data in the package. The package code can be only Python code, such as a Python machine learning algorithm, but the Python code can also execute other executable codes (such as libRadtran) on the worker node. The process to execute other codes will be described later in this report.

2.5 Serialisation

Passing data between computers requires conversion of the data into a sequence of bytes that can be communicated across the network. The code and data flowing from the user PC to the scheduler and worker must be 'zipped-up'[2] into a package. This is called serialisation. Choices made in serialization can affect performance and security. Dask serialisation is described here: `https://distributed.dask.org/en/latest/serialization.html`.

The standard Python solution to this, Pickle, is often but not always the right solution. Dask uses a number of different serialization schemes in different situations. These are extensible to allow users to control in sensitive situations and also to enable library developers to plug in higher performing serialization solutions. Figure 2.4 shows the serialisation process in Dask.

There are three kinds of messages passed through the Dask network:

1. Small administrative messages like "Worker A has finished task X" or "I'm running out of memory". These are always serialized with `msgpack`.

[2]The term zip up is used figuratively, it is not implying that the pkzip technology is used.

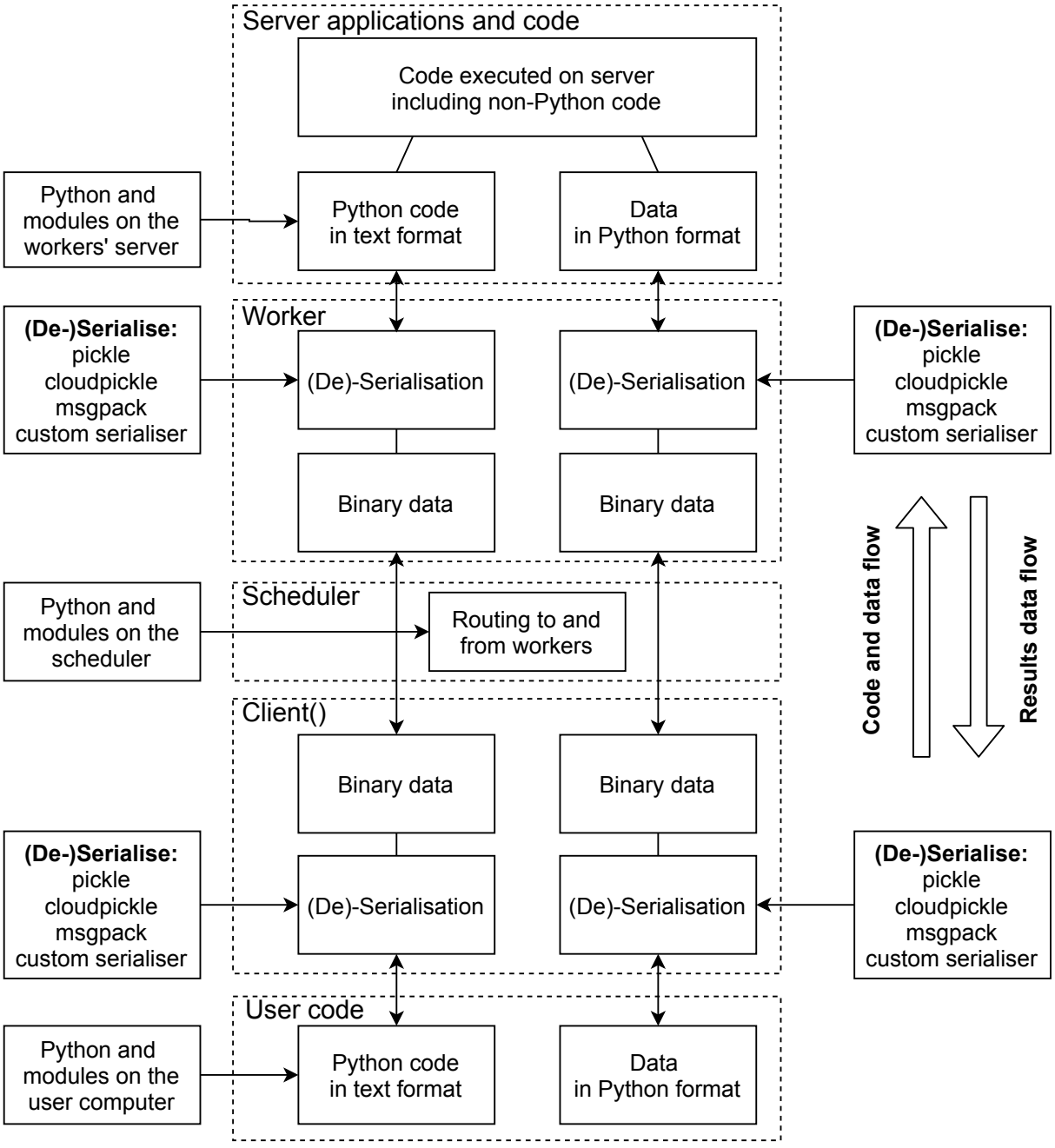


Figure 2.4: Dask serialisation

2. Movement of program data, such as Numpy arrays and Pandas dataframes. This uses a combination of pickle, cloudpickle, msgpack, and custom per-type serialisers that come with Dask.

3. Computational tasks (Python code) like `f(x)` that are defined and serialized on client processes and deserialized and run on worker processes. These are serialized using a fixed scheme decided on by those libraries. Today this is a combination of pickle and cloudpickle.

You can choose which families you want to use to serialize data and to deserialize data when you create a client.

The various serialisation modules implement compact binary protocols for serializing and deserializing a Python object structure. These protocols are unique and version specific, which may lead to incompatibilities between the serialised data and the module versions used to deserialise the data. Most of these serialisation module formats are not transportable between versions.

For more information on the pickle module see
`https://docs.python.org/3/library/pickle.html`, and
`https://docs.python.org/2/library/pickle.html`.

2.6 Versions

Careful review of Figure 2.4 indicates that there are potentially three different Python instances: one each in the user computer, the scheduler computer and one in the worker computer[3]. The way Dask works is to pass the Python code for execution to the other computer. If these versions differ, there could be complications. The idea is that the code should execute exactly the same irrespective of which computer it is executed on. Hence the software versions of Python and all the modules used, must be exactly the same on all computers — even to the bug fix version (the last number in the three-number version number).

Dask should check the versions as default action when the packages are sent around. If the versions are different the user is warned and the processes is stopped. If the versions do not match an error report similar to the following may be displayed. The Dask software is continuously updated so the format may differ from as it is shown here.

```
distributed.worker - WARNING - Mismatched versions found

distributed
+---------------------------+---------------------------+
|                           | version                   |
+---------------------------+---------------------------+
| This Worker               | 2.11.0+21.g05ca3fb1.dirty |
| scheduler                 | 2.12.0                    |
| tcp://192.168.0.102:64357 | 2.11.0+21.g05ca3fb1.dirty |
+---------------------------+---------------------------+

msgpack
+---------------------------+---------+
|                           | version |
+---------------------------+---------+
```

[3] If all three instances are on the same computer then the same Python is used.

```
| This Worker               | 0.6.1   |
| scheduler                 | 1.0.0   |
| tcp://192.168.0.102:64357 | 0.6.1   |
+---------------------------+---------+
```

or

```
ValueError: Mismatched versions found

cloudpickle
+------------------------+---------+
|                        | version |
+------------------------+---------+
| client                 | 0.5.3   |
| scheduler              | 0.5.2   |
| tcp://100.96.1.5:36964 | 0.5.2   |
| tcp://100.96.1.6:38415 | 0.5.2   |
| tcp://100.96.2.6:44389 | 0.5.2   |
| tcp://100.96.2.7:39214 | 0.5.2   |
| tcp://100.96.2.8:40317 | 0.5.2   |
+------------------------+---------+

dask
+------------------------+---------+
|                        | version |
+------------------------+---------+
| client                 | 0.18.2  |
| scheduler              | 0.17.4  |
| tcp://100.96.1.5:36964 | 0.17.4  |
| tcp://100.96.1.6:38415 | 0.17.4  |
| tcp://100.96.2.6:44389 | 0.17.4  |
| tcp://100.96.2.7:39214 | 0.17.4  |
| tcp://100.96.2.8:40317 | 0.17.4  |
+------------------------+---------+
```

If the process does not work correctly, check for the versions by using `client.get_versions(check=True)`, that returns the current versions in a json file.

```
{'scheduler': {'host': (('python', '3.7.6.final.0'),
   ('python-bits', 64),
   ('OS', 'Windows'),
   ('OS-release', '7'),
   ('machine', 'AMD64'),
   ('processor', 'Intel64 Family 6 Model 94 Stepping 3, GenuineIntel'),
   ('byteorder', 'little'),
   ('LC_ALL', 'None'),
   ('LANG', 'None')),
  'packages': {'dask': '2.12.0',
   'distributed': '2.12.0',
   'msgpack': '0.6.1',
   'cloudpickle': '1.3.0',
   'tornado': '6.0.4',
   'toolz': '0.10.0',
   'numpy': '1.18.1',
   'lz4': None,
   'blosc': None}},
 'workers': {'tcp://127.0.0.1:63694': {'host': (('python', '3.7.6.final↵
    .0'),
```

```
    ('python-bits', 64),
    ('OS', 'Windows'),
    ('OS-release', '7'),
    ('machine', 'AMD64'),
    ('processor', 'Intel64 Family 6 Model 94 Stepping 3, GenuineIntel'),
    ('byteorder', 'little'),
    ('LC_ALL', 'None'),
    ('LANG', 'None')),
   'packages': {'dask': '2.12.0',
    'distributed': '2.12.0',
    'msgpack': '0.6.1',
    'cloudpickle': '1.3.0',
    'tornado': '6.0.4',
    'toolz': '0.10.0',
    'numpy': '1.18.1',
    'lz4': None,
    'blosc': None}}},
 'client': {'host': [('python', '3.7.6.final.0'),
   ('python-bits', 64),
   ('OS', 'Windows'),
   ('OS-release', '7'),
   ('machine', 'AMD64'),
   ('processor', 'Intel64 Family 6 Model 94 Stepping 3, GenuineIntel'),
   ('byteorder', 'little'),
   ('LC_ALL', 'None'),
   ('LANG', 'None')],
  'packages': {'dask': '2.12.0',
   'distributed': '2.12.0',
   'msgpack': '0.6.1',
   'cloudpickle': '1.3.0',
   'tornado': '6.0.4',
   'toolz': '0.10.0',
   'numpy': '1.18.1',
   'lz4': None,
   'blosc': None}}}
```

The solution to this problem is to ensure that the same versions are present on all computers. See Section 4.1.2 on installing specific package versions.

2.7 Dask Experiments

The rest of this chapter is an extract from this notebook:
`https://github.com/NelisW/miscellania/blob/master/dask/Dask-Experiments.ipynb`↵

2.7.1 Setting Up

The following page explain in more detail how to set up Dask on a variety of local and distributed hardware. `https://docs.dask.org/en/latest/setup.html`

There are two ways to set up Dask on a local computer. Both of these ways are explored in the notebook. Setting up on a local computer is important to grasp the concepts even if the end objective is to use a multi PC configuration.

There are several multi-computer set up techniques. This document explores two techniques: manual set up and SSH set up.

```
from dask.distributed import Client, LocalCluster
```

2.7.2 Single Machine (local PC)

The `dask.distributed` scheduler works well on a single machine. It is sometimes preferred over the default scheduler. You can create a `dask.distributed` scheduler by importing and creating a Client with no arguments. This overrides whatever default was previously set. In the context of this section, the word distributed is understood to be distributed programmatically on the local physical computer (not distributed in the physical sense).

The `Client()` call used here is shorthand for creating a `LocalCluster` and then passing that to your client. You may want to look at the wider range keyword arguments available on `LocalCluster` to understand the options available to you on handling the mixture of threads and processes, like specifying explicit ports, and so on. For example, if you use the `localCluster` approach you can set the number of workers.

Note that `Client()` and `LocalCluster()` take many optional arguments, to configure the server.

You can navigate to `http://localhost:8787/status` to see the diagnostic dashboard if you have Bokeh installed.

Once the local client is started it will start a local cluster, but all working on the local computer.

The `client.scheduler\_info()` command provides information about the client, server and worker setup.

The `client.shutdown()` command can be used to shut down the client on the server

`https://docs.dask.org/en/latest/setup/single-distributed.html`

`https://distributed.dask.org/en/latest/api.html#distributed.Client`

Dask is not the only means to do parallel processing. See also this tutorial on multiprocessing.

`https://sebastianraschka.com/Articles/2014\_multiprocessing.html`

```
# to use use local host
useSimpleClient = True

if useSimpleClient:
    # simplified setup, less control
    client = Client()
else:
    # Setup a local cluster, more control
    # By default this sets up 1 worker per core
    cluster = LocalCluster(n_workers=1)
    client = Client(cluster)

client.get_versions(check=True)

client
```

Client Scheduler: tcp://127.0.0.1:64865 Dashboard: http://127.0.0.1:8787/status	Cluster Workers: 4 Co

```
client.scheduler_info()
```


```
{'type': 'Scheduler',
 'id': 'Scheduler-e40d7349-ee4d-481f-b416-bc9b75b87b92',
 'address': 'tcp://127.0.0.1:64865',
 'services': {'dashboard': 8787},
 'workers': {'tcp://127.0.0.1:64887': {'type': 'Worker',
   'id': 3,
   'host': '127.0.0.1',
   'resources': {},
   'local_directory': 'V:\\work\\WorkN\\miscellania\\dask\\dask-worker-↵
      space\\worker-sz1aoteq',
   'name': 3,
   'nthreads': 2,
   'memory_limit': 8500740096,
   'last_seen': 1586869485.3058014,
   'services': {},
   'metrics': {'cpu': 0.0,
    'memory': 52654080,
    'time': 1586869485.2592325,
    'read_bytes': 0.0,
    'write_bytes': 0.0,
    'executing': 0,
    'in_memory': 0,
    'ready': 0,
    'in_flight': 0,
    'bandwidth': {'total': 100000000, 'workers': {}, 'types': {}}},
   'nanny': 'tcp://127.0.0.1:64868'},
  'tcp://127.0.0.1:64889': {'type': 'Worker',
   'id': 2,
   'host': '127.0.0.1',
   'resources': {},
   'local_directory': 'V:\\work\\WorkN\\miscellania\\dask\\dask-worker-↵
      space\\worker-a3868zvm',
   'name': 2,
   'nthreads': 2,
   'memory_limit': 8500740096,
   'last_seen': 1586869485.3589983,
   'services': {},
   'metrics': {'cpu': 0.0,
    'memory': 52600832,
    'time': 1586869485.3125384,
    'read_bytes': 0.0,
    'write_bytes': 0.0,
    'executing': 0,
    'in_memory': 0,
    'ready': 0,
    'in_flight': 0,
    'bandwidth': {'total': 100000000, 'workers': {}, 'types': {}}},
   'nanny': 'tcp://127.0.0.1:64870'},
  'tcp://127.0.0.1:64890': {'type': 'Worker',
   'id': 1,
   'host': '127.0.0.1',
   'resources': {},
```

```
   'local_directory': 'V:\\work\\WorkN\\miscellania\\dask\\dask-worker-↵
      space\\worker-jou7s9bk',
   'name': 1,
   'nthreads': 2,
   'memory_limit': 8500740096,
   'last_seen': 1586869485.3625963,
   'services': {},
   'metrics': {'cpu': 0.0,
    'memory': 52740096,
    'time': 1586869485.3174865,
    'read_bytes': 0.0,
    'write_bytes': 0.0,
    'executing': 0,
    'in_memory': 0,
    'ready': 0,
    'in_flight': 0,
    'bandwidth': {'total': 100000000, 'workers': {}, 'types': {}}},
   'nanny': 'tcp://127.0.0.1:64869'},
  'tcp://127.0.0.1:64893': {'type': 'Worker',
   'id': 0,
   'host': '127.0.0.1',
   'resources': {},
   'local_directory': 'V:\\work\\WorkN\\miscellania\\dask\\dask-worker-↵
      space\\worker-djqrx1np',
   'name': 0,
   'nthreads': 2,
   'memory_limit': 8500740096,
   'last_seen': 1586869485.4007897,
   'services': {},
   'metrics': {'cpu': 0.0,
    'memory': 52621312,
    'time': 1586869485.3620787,
    'read_bytes': 0.0,
    'write_bytes': 0.0,
    'executing': 0,
    'in_memory': 0,
    'ready': 0,
    'in_flight': 0,
    'bandwidth': {'total': 100000000, 'workers': {}, 'types': {}}},
   'nanny': 'tcp://127.0.0.1:64867'}}}
```

```
# client.shutdown()
```

```
#client.get_versions(check=True)
```

There are a few different ways to interact with the cluster through the client:

1. The Client satisfies most of the standard concurrent.futures - PEP-3148 interface with `.submit`, `.map` functions and `Future` objects, allowing the immediate and direct submission of tasks `https://docs.python.org/3/library/concurrent.futures.↵html`.
2. The Client registers itself as the default Dask scheduler, and so runs all dask collections like `dask.array`, `dask.bag`, `dask.dataframe` and `dask.delayed`
3. The Client has additional methods for manipulating data remotely. See the full API for a thorough list `https://distributed.dask.org/en/latest/api.html`.

Dask Dashboard

The Dask dashboard is a bokeh-based display of what is taking place in the server and its workers. For this to work the Python bokeh package must be installed.

Workers capture durations associated to tasks. For each task that passes through a worker we record start and stop times for each of the following:

- Serialization (gray)
- Dependency gathering from peers (red)
- Disk I/O to collect local data (orange)
- Execution times (coloured by task)

The main way to observe these times is with the task stream plot on the scheduler's /status page where the colours of the bars correspond to the colours listed above.

`https://docs.dask.org/en/latest/diagnostics-distributed.html`

`https://distributed.dask.org/en/latest/diagnosing-performance.html`

`https://medium.com/@kartikbhanot/dask-scheduler-dashboard-understanding-resource-and-task-allocation-in-local-machines-bc5aa60eca6e`

`https://www.youtube.com/watch?v=N\_GqzcuGLCY`

There is also a Dask extension for Jupyter notebooks `dask-labextension`.

`https://github.com/dask/dask-labextension`

```
# to obtain the dashboard for the client
def clientDashboardURI(client):
    """Extract the bokeh dashboard URI from the client's scheduler info
    """

    dashboardURI = client.scheduler_info()['address'].rsplit(':',1)[0]+':'
    dashboardURI += str(client.scheduler_info()['services']['dashboard'])

    return dashboardURI.replace('tcp:','http:')

print(clientDashboardURI(client))
```

```
http://127.0.0.1:8787
```

Open a browser window with the above Uniform Resource Identifier (URI) (for the currently running client). Set the `if` condition to `True` and execute the code below to observe the dashboard in action. The code below is not meaningful it is just meant to keep Dask busy for a while.

While the code is executing, click on the different tabs in the dashboard to learn about the different displays.

```
if False:
    import dask.array as da
    x = da.random.random((10000, 10000,10), chunks=(1000,1000,5))
```

```
    y = da.random.random((10000, 10000,10), chunks=(1000,1000,5))
    z = (da.arcsin(x)+da.arccos(y)).sum(axis=(1,2))
    z.compute()
```

Preparing functions

Create a few functions that will be used in the examples.

```
# to create simple task to experiment with
def inc(x):
    return x + 1

def add(x, y):
    return x + y
```

Single function calls

Using the client created above, a single function and a single data (datum?) item will be dispatched to the scheduler and worker. All of the scheduling work is transparent.

We can submit individual function calls with the `client.submit` method.

The simple example here will execute much faster in normal in-line Python code, the idea here is to show the Dask method.

The `submit` function returns a `Future`, which refers to a (future) remote result. This result may not yet be completed. Eventually it will complete. The result stays in the remote thread-/process/worker until you ask for it back explicitly by calling the `result` method on the future itself (not a client method as for the map case below).

`https://distributed.dask.org/en/latest/client.html`

```
# to create a single future
x1 = client.submit(inc, 10)
print(x1)
```

```
<Future: pending, key: inc-083b5e2ba45c380966bcb963d4544e5e>
```

```
# to see what a future looks like
print(x1)
```

```
<Future: finished, type: builtins.int, key: inc-083↵
   b5e2ba45c380966bcb963d4544e5e>
```

```
# to retrieve a result from a future
x1r = x1.result()
print(x1r)
```

```
11
```

```
# to check if a future resets or disappears after its initial retrieval
print(x1)
x1r = x1.result()
print(x1r)
```

```
<Future: finished, type: builtins.int, key: inc-083↵
   b5e2ba45c380966bcb963d4544e5e>
11
```

You can pass futures as inputs to submit. Dask automatically handles dependency tracking; once all input futures have completed, they will be moved onto a single worker (if necessary), and then the computation that depends on them will be started. You do not need to wait for inputs to finish before submitting a new task; Dask will handle this automatically:

```
# to create a graph of futures
x1 = client.submit(inc, 10)
x2 = client.submit(inc, 10)
print(x1)
print(x2)
xs = client.submit(add,x1,x2)
print(xs)
```

```
<Future: finished, type: builtins.int, key: inc-083↵
   b5e2ba45c380966bcb963d4544e5e>
<Future: finished, type: builtins.int, key: inc-083↵
   b5e2ba45c380966bcb963d4544e5e>
<Future: pending, key: add-6e6cf98db4b0523bbdc831467bd5720b>
```

```
# to print the result of the graph final output
xsr = xs.result()
print(xsr)
```

```
22
```

Multiple function calls

Using the client created above, a single function and multiple data items will be dispatched to the scheduler and worker. All of the scheduling work is transparent.

Similar to Python's `map`, you can use `Client.map` to call the same function and many inputs.

The returns a list of futures. These results live on the distributed workers.

We can submit tasks on futures. The function will go to the machine where the futures are stored and run on the result once it has completed.In the example below the list of futures is sent to the built-in Python `sum()` function, which adds the elements of an iterable and returns the sum.

```
# to create a map of input values for a function
futures  = client.map(inc, [2, 4, 6])
for future in futures:
    print(future)
```

```
<Future: pending, key: inc-423e89f2660795305fd6c03c55d1de5d>
<Future: pending, key: inc-98feef5b8b4598158d0e47a6d4acde9d>
<Future: pending, key: inc-d016ddc28876119154a07cfbe98a9ed7>
```

The results stay in the remote thread/process/worker until you ask for it back explicitly by calling the client `gather` method (not the future's result method) because here a list must be processed.

```
# to gather the list of results
futuresr = client.gather(futures)
print(futuresr)
```

```
[3, 5, 7]
```

```
# to graph a list of futures into another submit
total = client.submit(sum, futures)
print(total)

totalr = total.result()
print(totalr)
```

```
<Future: pending, key: sum-c34d59da0efacac3f60627d03084e274>
15
```

Serialising Code and Data

In the examples shown above the code and data are serialised in the client before passing on to the scheduler. This can be visualised as shown in Figure 2.5.

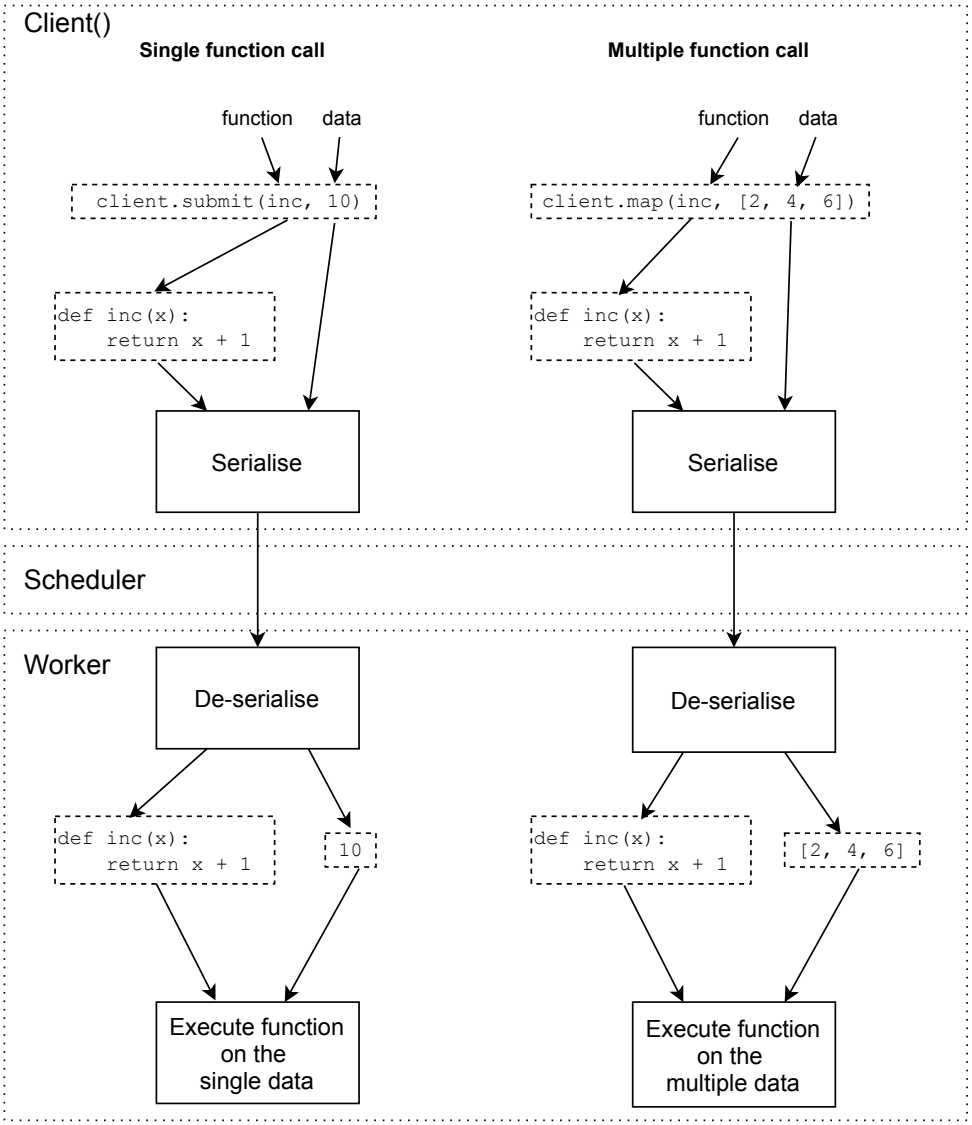


Figure 2.5: Dask serialisation and worker execution visualised

2.7.3 Multiple Machines (local PC and Remote Server(s))

Setting up

For the sake of these discussions suppose the computers have the following IP addresses (replace with the IP addresses on your system):

1. Local computer running the `dask.distributed.Client`: `146.64.202.163`
2. Computer running the scheduler: `146.64.246.94`
3. Computer running the workers: `146.64.246.94`

In this case the scheduler and worker are run on the same server, but could be ran on different computers.

All three computers must have exactly the same versions of Python, dask, pickle, etc., otherwise the following will not work. If required versions are available in a Python environment, the environment must be activated.

The client, scheduler and workers must be able to communicate with each other. This is achieved by using selected ports and IP addresses. The main IP address of interest here is the scheduler IP address which must be made known to the client and the workers, see the example below.

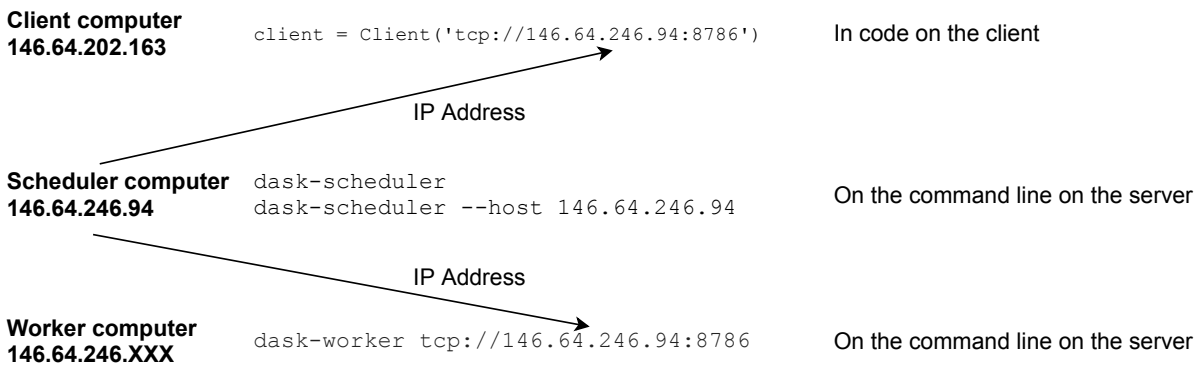


Figure 2.6: IP port notification in Dask

Start the scheduler and workers:

1. **Start the scheduler on the server**. Log in to the server (`146.64.246.94`), then activate the Python environment and start the scheduler:

   ```
   conda activate mordevpy37
   dask-scheduler
   ```

   If the server already has the necessary Python, dask and pickle versions in the root Python environment there is no need to first activate the Python environment. In this case the scheduler can be started from any computer by specifying the hostname. So on the local computer the following command will start the scheduler on the server (if versions are correct in the root Python):

   ```
   dask-scheduler --host 146.64.246.94
   ```

Either of the above should start the scheduler:

```
(mordevpy37) dgriffith@nimbus:~$ dask-scheduler
 distributed.scheduler - INFO - -----------------------------------------------
 distributed.scheduler - INFO - Local Directory:    /tmp/scheduler-mxiy5sli
 distributed.scheduler - INFO - -----------------------------------------------
 distributed.scheduler - INFO - Clear task state
 distributed.scheduler - INFO -   Scheduler at:  tcp://146.64.246.94:8786
 distributed.scheduler - INFO -   dashboard at:                     :8787
 distributed.scheduler - INFO - Register worker <Worker 'tcp://146.64.246.94:45347', ↵
     name: tcp://146.64.246.94:45347, memory: 0, processing: 0>
 distributed.scheduler - INFO - Starting worker compute stream, tcp↵
     ://146.64.246.94:45347
 distributed.core - INFO - Starting established connection
```

2. **Start the worker on the server** Log in to the server (`146.64.246.94`), then activate the Python environment and start the worker:

```
dask-worker tcp://146.64.246.94:8786
```

where the IP address and port number must correspond to the scheduler IP and port address. This should start the worker task. Note that the dashboard IP address is also given when the workers start. Without explicitly defining the number of processors, this server has 32 cores and 32 Gigabyte (GB) memory.

```
(mordevpy37) dgriffith@nimbus:~$ dask-worker tcp://146.64.246.94:8786
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:35459'
distributed.worker - INFO -       Start worker at:  tcp://146.64.246.94:45347
distributed.worker - INFO -          Listening to:  tcp://146.64.246.94:45347
distributed.worker - INFO -          dashboard at:        146.64.246.94:42791
distributed.worker - INFO - Waiting to connect to:   tcp://146.64.246.94:8786
distributed.worker - INFO - -------------------------------------------------
distributed.worker - INFO -               Threads:                         32
distributed.worker - INFO -                Memory:                  33.71 GB
distributed.worker - INFO -       Local Directory: /home/dgriffith/dask-worker-space/↵
    worker-lz08iq0s
distributed.worker - INFO - -------------------------------------------------
distributed.worker - INFO -         Registered to:   tcp://146.64.246.94:8786
distributed.worker - INFO - -------------------------------------------------
distributed.core - INFO - Starting established connection
```

You can also define the number of processors to be used and the threads per processor, see https://stackoverflow.com/questions/49406987/how-do-we-choose-nthreads-and-nprocs-per-worker-in-dask-distributed

By default Dask creates a single process with as many threads as you have logical cores on your machine (as determined by `multiprocessing.cpu_count()`).

```
dask-worker ... --nprocs 1 --nthreads 8  # assuming you have eight ↵
    cores
dask-worker ...                          # this is actually the ↵
    default setting
```

Using few processes and many threads per process is good if you are doing mostly numeric workloads, such as are common in Numpy, Pandas, and Scikit-Learn code, which is not affected by Python's Global Interpreter Lock (GIL). However, if you are spending most of your compute time manipulating Pure Python objects like strings or dictionaries then you may want to avoid GIL issues by having more processes with fewer threads each

```
dask-worker ... --nprocs 8 --nthreads 1
```

Based on benchmarking you may find that a more balanced split is better

```
dask-worker ... --nprocs 4 --nthreads 2
```

Using more processes avoids GIL issues, but adds costs due to inter-process communication. You would want to avoid many processes if your computations require a lot of inter-worker communication..

So specifying eight processors:

```
dask-worker tcp://146.64.246.94:8786 --nprocs 8
```

Starts the workers as follows, each with a different port number (not all the detail shown here):

```
(mordevpy37) dgriffith@nimbus:~/libRadtran/libRadtran-2.0.3/bin$ dask↵
   -worker tcp://146.64.246.94:8786 --nprocs 8
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:33671'
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:46725'
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:42509'
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:42983'
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:43735'
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:36947'
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:40075'
distributed.nanny - INFO -         Start Nanny at: 'tcp↵
   ://146.64.246.94:40495'
...
distributed.worker - INFO -       Start worker at:  tcp↵
   ://146.64.246.94:36533
distributed.worker - INFO -          Listening to:  tcp↵
   ://146.64.246.94:36533
distributed.worker - INFO -          dashboard at:        ↵
   146.64.246.94:43069
distributed.worker - INFO - Waiting to connect to:   tcp↵
   ://146.64.246.94:8786
distributed.worker - INFO - ↵
   -------------------------------------------------
distributed.worker - INFO -               Threads: ↵
                           4
distributed.worker - INFO -                Memory: ↵
                     4.21 GB
distributed.worker - INFO -       Local Directory: /home/dgriffith/↵
   libRadtran/libRadtran-2.0.3/bin/dask-worker-space/worker-_0o0qta1
distributed.worker - INFO - ↵
   -------------------------------------------------
distributed.worker - INFO -         Registered to:   tcp↵
   ://146.64.246.94:8786
distributed.worker - INFO - ↵
   -------------------------------------------------
distributed.core - INFO - Starting established connection
...
distributed.worker - INFO -       Start worker at:  tcp↵
   ://146.64.246.94:46597
```

```
distributed.worker - INFO -          Listening to:  tcp↵
   ://146.64.246.94:46597
distributed.worker - INFO -          dashboard at:        ↵
   146.64.246.94:33737
distributed.worker - INFO - Waiting to connect to:   tcp↵
   ://146.64.246.94:8786
distributed.worker - INFO - ↵
   -------------------------------------------------
distributed.worker - INFO -               Threads: ↵
                           4
distributed.worker - INFO -                Memory: ↵
                     4.21 GB
distributed.worker - INFO -       Local Directory: /home/dgriffith/↵
   libRadtran/libRadtran-2.0.3/bin/dask-worker-space/worker-n0anjq00
distributed.worker - INFO - ↵
   -------------------------------------------------
distributed.core - INFO - Starting established connection
distributed.worker - INFO -         Registered to:   tcp↵
   ://146.64.246.94:8786
distributed.worker - INFO - ↵
   -------------------------------------------------
distributed.core - INFO - Starting established connection
```

3. **Start the client** When the client is initiated, pass the scheduler IP:port address:

```
# to start a local client with a remote scheduler
client = Client('146.64.246.94:8786')
```

To see the Python module versions:

```
client.get_versions(check=True)
```

```
{'scheduler': {'host': (('python', '3.7.6.final.0'),
   ('python-bits', 64),
   ('OS', 'Linux'),
   ('OS-release', '4.9.0-8-amd64'),
   ('machine', 'x86_64'),
   ('processor', ''),
   ('byteorder', 'little'),
   ('LC_ALL', 'None'),
   ('LANG', 'en_ZA.UTF-8')),
  'packages': {'dask': '2.12.0',
   'distributed': '2.12.0',
   'msgpack': '0.6.1',
   'cloudpickle': '1.3.0',
   'tornado': '6.0.4',
   'toolz': '0.10.0',
   'numpy': '1.18.1',
   'lz4': None,
   'blosc': None}},
 'workers': {'tcp://146.64.246.94:45347': {'host': (('python',
     '3.7.6.final.0'),
    ('python-bits', 64),
    ('OS', 'Linux'),
    ('OS-release', '4.9.0-8-amd64'),
    ('machine', 'x86_64'),
    ('processor', ''),
    ('byteorder', 'little'),
    ('LC_ALL', 'None'),
    ('LANG', 'en_ZA.UTF-8')),
```

```
   'packages': {'dask': '2.12.0',
    'distributed': '2.12.0',
    'msgpack': '0.6.1',
    'cloudpickle': '1.3.0',
    'tornado': '6.0.4',
    'toolz': '0.10.0',
    'numpy': '1.18.1',
    'lz4': None,
    'blosc': None}}},
 'client': {'host': [('python', '3.7.6.final.0'),
   ('python-bits', 64),
   ('OS', 'Windows'),
   ('OS-release', '7'),
   ('machine', 'AMD64'),
   ('processor', 'Intel64 Family 6 Model 94 Stepping 3, GenuineIntel↵
      '),
   ('byteorder', 'little'),
   ('LC_ALL', 'None'),
   ('LANG', 'None')],
  'packages': {'dask': '2.12.0',
   'distributed': '2.12.0',
   'msgpack': '0.6.1',
   'cloudpickle': '1.3.0',
   'tornado': '6.0.4',
   'toolz': '0.10.0',
   'numpy': '1.18.1',
   'lz4': None,
   'blosc': None}}}
```

To see the Dask scheduler information:

```
client.scheduler_info()
```

To determine the number of workers:

```
len(client.scheduler_info()['workers'])
```

```
4
```

```
{'type': 'Scheduler',
 'id': 'Scheduler-c18c6181-421b-4826-aea7-b1bba21efeb0',
 'address': 'tcp://146.64.246.94:8786',
 'services': {'dashboard': 8787},
 'workers': {'tcp://146.64.246.94:45347': {'type': 'Worker',
   'id': 'tcp://146.64.246.94:45347',
   'host': '146.64.246.94',
   'resources': {},
   'local_directory': '/home/dgriffith/dask-worker-space/worker-↵
      lz08iq0s',
   'name': 'tcp://146.64.246.94:45347',
   'nthreads': 32,
   'memory_limit': 33712070656,
   'last_seen': 1586869487.8488212,
   'services': {'dashboard': 42791},
   'metrics': {'cpu': 2.0,
    'memory': 107896832,
    'time': 1586869487.3504004,
    'read_bytes': 16650.21565100753,
    'write_bytes': 1014.9159528091943,
    'num_fds': 25,
```

```
    'executing': 0,
    'in_memory': 0,
    'ready': 0,
    'in_flight': 0,
    'bandwidth': {'total': 100000000, 'workers': {}, 'types': {}}},
   'nanny': 'tcp://146.64.246.94:35459'}}}
```

Running workers remotely

Now execute the same graph as on the single computer before, but now on the remote server.

To see if the server is running, put the `client()` call in a Python exception.

```
# to run the graph on a remove server

try:
    client = Client('tcp://146.64.246.94:8786', timeout='2s')
    x1 = client.submit(inc, 10)
    x2 = client.submit(inc, 10)
    print(x1)
    print(x2)
    xs = client.submit(add,x1,x2)
    print(xs)
    # to print the result of the graph final output
    xsr = xs.result()
    print(xsr)
except TimeoutError:
    print('dask scheduler server is not responding, probably not running.'↵
        )
```

```
<Future: pending, key: inc-083b5e2ba45c380966bcb963d4544e5e>
<Future: pending, key: inc-083b5e2ba45c380966bcb963d4544e5e>
<Future: pending, key: add-6e6cf98db4b0523bbdc831467bd5720b>
22
```

Debugging

If the scheduler fails to start with messages such as shown below, check to see

1. If starting the scheduler remotely with `dask-scheduler --host 146.64.246.94`: Does the server's root Python have the correct software and versions? If the scheduler is started remotely the server's root Python is used.
2. If the server is started locally on the server with `dask-scheduler`: Does the currently active Python environment (root or other) have the correct software and versions?

```
dask-scheduler --host 146.64.202.118
distributed.scheduler - INFO - -----------------------------------------------
distributed.dashboard.proxy - INFO - To route to workers diagnostics web server please ↵
    install jupyter-server-proxy: python -m pip install jupyter-server-proxy
distributed.scheduler - INFO - Local Directory: C:\Users\NWillers\AppData\Local\Temp\↵
    scheduler-b856sir6
distributed.scheduler - INFO - -----------------------------------------------
distributed.scheduler - INFO - Clear task state
Traceback (most recent call last):
  File "C:\ProgramData\Anaconda3\lib\site-packages\distributed\cli\dask\_scheduler.py", line ↵
     237, in main
```

```
    loop.run\_sync(run)
  File "C:\ProgramData\Anaconda3\lib\site-packages\tornado\ioloop.py", line 532, in run\_sync
    return future\_cell[0].result()
  File "C:\ProgramData\Anaconda3\lib\site-packages\distributed\cli\dask\_scheduler.py", line ↲
      233, in run
    await scheduler
  File "C:\ProgramData\Anaconda3\lib\site-packages\distributed\scheduler.py", line 1424, in ↲
      start
    await self.listen(addr, listen\_args=self.listen\_args)
  File "C:\ProgramData\Anaconda3\lib\site-packages\distributed\core.py", line 319, in listen
    connection\_args=listen\_args,
  File "C:\ProgramData\Anaconda3\lib\site-packages\distributed\comm\core.py", line 170, in \_
    await self.start()
  File "C:\ProgramData\Anaconda3\lib\site-packages\distributed\comm\tcp.py", line 411, in ↲
      start
    self.port, address=self.ip, backlog=backlog
  File "C:\ProgramData\Anaconda3\lib\site-packages\tornado\netutil.py", line 174, in bind\↲
      _sockets
    sock.bind(sockaddr)
OSError: [WinError 10049] The requested address is not valid in its context
```

Chapter 3

libRadtran

3.1 libRadtran Overview

From [2]: "libRadtran is a widely used software package for radiative transfer calculations. It allows one to compute (polarized) radiances, irradiance, and actinic fluxes in the solar and thermal spectral regions. libRadtran has been used for various applications, including remote sensing of clouds, aerosols and trace gases in the Earth's atmosphere, climate studies, e.g., for the calculation of radiative forcing due to different atmospheric components, for UV forecasting, the calculation of photolysis frequencies, and for remote sensing of other planets in our solar system. The package has been described in Mayer and Kylling (2005) [3]. Since then several new features have been included, for example polarization, Raman scattering, a new molecular gas absorption parameterisation, and several new parameterizations of cloud and aerosol optical properties. Furthermore, a graphical user interface is now available, which greatly simplifies the usage of the model, especially for new users. This paper gives an overview of libRadtran version 2.0.1 with a focus on new features. Applications including these new features are provided as examples of use. A complete description of libRadtran and all its input options is given in the user manual included in the libRadtran software package, which is freely available at `http://www.libradtran.org`."

See also [3, 6, 7, 8].

3.2 Obtaining and Building libRadtran

3.3 Reference Systems

For reference purposes this installation was done on the following Linux systems.

Nimbus Server

This is an existing server that had libRadtran 2.0.0 running on it before, so all the pre-requisites were installed.

```
(mordevpy27) dgriffith@nimbus:~/libRadtran/libRadtran-2.0.3/examples$ uname -a
Linux nimbus 4.9.0-8-amd64 #1 SMP Debian 4.9.110-3+deb9u6 (2018-10-08) x86_64 GNU/Linux
```

Fresh Install

This was a newly-set-up Ubuntu 20.04 installation, in an Oracle VirtualBox (Ver 6.1.6). This means that none of the prequisites were already installed on the system.

```
(base) nwillers@nwillers-VirtualBox:~$ uname -a
Linux nwillers-VirtualBox 5.4.0-26-generic #30-Ubuntu SMP Mon Apr 20 16:58:30 UTC 2020 x86_64↵
     x86_64 x86_64 GNU/Linux
```

The initial installation (for other general development work) was as follows. The packages were installed using synaptic, my preferred Ubuntu installer.

1. CMake.
2. Make (4.2.1-1.2).
3. GCC (4:9.3.0-1ubuntu2).
4. Perl (might be installed already).
5. gparted.
6. Konsole.
7. Kdiff.
8. git.

libRadtran requires additional tools: `http://www.libradtran.org/doku.php?id=download`. Don't take shortcuts, it does not work. Take the trouble to install these (in the Ubuntu package name convention):

1. flex.
2. gfortran.
3. NetCDF for C: libnetcdf-dev.
4. NetCDF for Fortran: libnetcdff-dev.
5. The GNU Scientific Library: gsl-bin, libgsl-dev, libgsl23 libglscblas0.

3.3.1 Downloading libRadtran

Download libRadtran and the optional additional files/modules from
`http://www.libradtran.org/doku.php?id=download`
For our application the REPTRAN absorption parameterization data was required, but it may differ between different sites.

3.3.2 Create the server account and software

1. Create a user account on the server from which libRadtran is served. This is a standard Linux admin task. For the present case assume the account name is `dgriffith`.

2. Download libRadtran from `http://www.libradtran.org/doku.php?id=download` and follow the installation instructions. The instructions are repeated below. In this case libRadtran version 2.0.3 is downloaded and installed.

3. Copy the downloaded file to the folder where it must be installed. Unzipping the compressed tar file with the following command will create a folder `libRadtran-2.0.3`

   ```
   tar -xvf  libradtran-2.0.3.tar.gz
   ```

4. The libRadtran installation, building, and testing requires Python 2.7 (installation will fail if Python 3 is used). If necessary, install Python 2.7 in a conda environment in order to proceed with the libRadtran installation. For the purpose of this discussion we assume that the Python 2.7 environment is named `devpy27`.

   It seems that you do not need a full Anaconda Python 2.7 (including all the scientific libraries). It is sufficient to install only a conda:

   ```
   conda create --name devpy27 python=2.7
   ```

5. Compile the distribution by doing the following:

   ```
   conda activate devpy27
   cd libRadtran-2.0.3
   ./configure
   make
   ```

6. Test the program, (make sure to use GNU make):

   ```
   conda activate devpy27
   make check
   ```

   As the test progresses the test results will display in the terminal (the display shown here has been shortened):

```
(base) dgriffith@nimbus:~/libRadtran/libRadtran-2.0.3$ conda activate devpy27
(devpy27) dgriffith@snimbus:~/libRadtran/libRadtran-2.0.3$ make check
for dir in examples src_py libsrc_c libsrc_f src GUI lib; do make -C $dir all || exit $↵
    ?; done
make[1]: Entering directory '/home/dgriffith/libRadtran/libRadtran-2.0.3/examples'

...

make[1]: Entering directory '/home/dgriffith/libRadtran/libRadtran-2.0.3/test'
/usr/bin/perl test.pl
Running various libRadtran tests. This may take some time....

The numbers in the parenthesis behind the name of the tests are:
 1st number:  The lower absolute limit of values included in the test.
              Values in the output less than limit are ignored.
 2nd number:  The maximum difference allowed between local test
              results and the standard results (in percentage).

If this is still unclear, check the source in test/test.pl.in.

make_slitfunction test
make_slitfunction (0.00001,   0.1)..................................... ok.
All make_slitfunction tests succeeded.
make_angresfunc test
make_angresfunc (0.00001,   0.1)....................................... ok.
All make_angresfunc tests succeeded.
angres test
angres (0.00001,   0.1)................................................ ok.
All angres tests succeeded.
Some \lstinline{uvspec} tests
uvspec simple (0.00001,   0.1)......................................... ok.
disort clear sky (0.00001,   0.1)...................................... ok.
disort aerosol (0.00001,   0.1)........................................ ok.
disort aerosol moments (0.00001,   0.1)................................ ok.
disort aerosol refractive index (0.00001,   0.1)....................... ok.
disort BRDF Ross-Li (0.00050,   0.1)................................... ok.
disort water cloud (0.00001,   0.1).................................... ok.
disort SO2 (0.00001,   0.1)............................................ ok.
disort radiances (0.00001,   0.3)...................................... ok.
disort SCIAMACHY HG approximation (0.00001,   0.3)..................... ok.
disort aerosol (0.00001,   0.1)........................................ ok.
disort wc Legendre moments (0.00001,   0.1)............................ ok.
disort water and ice clouds (0.00001,   0.2)........................... ok.
```

```
disort cloud overlap random (0.00001,   0.2)................................ ok.
disort cloud overlap maximum-random (0.00001,   0.2)........................ ok.
disort cloud overlap maximum (0.00001,   0.2)............................... ok.
disort, albedo and altitude map (0.00001,   0.7)............................ ok.
disort BRDF (0.00050,   0.1)................................................ ok.
disort BRDF Hapke (0.00050,   0.1).......................................... ok.
disort BRDF AMBRALS (0.00050,   0.1)........................................ ok.
disort BRDF AMBRALS file (0.00050,   0.1)................................... ok.
twostr aerosol and water cloud (0.00001,   0.1)............................. ok.
twostrpp aerosol and water cloud (0.00001,   0.1)........................... ok.
c_twostr aerosol and water cloud (0.00001,   0.1)........................... ok.
rodents aerosol and water cloud (0.00001,   0.1)............................ ok.
rodents aerosol and water cloud, zout (0.00001,   0.1)...................... ok.
rodents aerosol and water cloud, zout, thermal (0.00001,   0.1).............. ok.
single scattering lidar, water cloud (0.00001,   0.1)....................... ok.
twostrebe aerosol and water cloud (0.00001,   0.1)........................... ok.
twomaxrand water cloud (0.00001,   0.1)..................................... ok.
twomaxrand wc/ic cloud (0.00001,   0.1)..................................... ok.
polradtran (0.00001,   0.1)................................................. ok.
profiles 1 (0.00001,   0.2)................................................. ok.
profiles 2 (0.00001,   0.1)................................................. ok.
profiles 3 (0.00001,   0.1)................................................. ok.
profiles 4 (0.00001,   0.1)................................................. ok.
radiosonde (0.00001,   0.4)................................................. ok.
redistribution (0.00001,   0.1)............................................. ok.
AVHRR [Kratz, 1995], channel 1, solar (0.00001,   0.1)...................... ok.
AVHRR [Kratz, 1995], channel 2, solar (0.00100,   0.1)...................... ok.
AVHRR [Kratz, 1995], channel 3, solar (0.00001,   0.1)...................... ok.
AVHRR [Kratz, 1995], channel 3, thermal (0.00001,   0.1).................... ok.
AVHRR [Kratz, 1995], channel 4, thermal (0.00001,   0.1).................... ok.
AVHRR [Kratz, 1995], channel 5, thermal (0.00001,   0.1).................... ok.
correlated-k [Kato et al., 1999], twostr (0.00100,   0.1).................... ok.
correlated-k, new Kato tables, twostr (0.00100,   0.1)...................... ok.
correlated-k [Fu and Liou, 1992/93], disort (0.01000,   1.5)................. ok.
correlated-k [Fu and Liou, 1992/93], ice clouds (0.01000,   1.5)............. ok.
Fu and Liou, thermal irradiance, disort (0.00001,   0.9)..................... ok.
LOWTRAN absorption parameterization (0.00001,   2.2)........................ ok.
LOWTRAN absorption parameterization, thermal (0.00001,   1.0)............... ok.
SSRadar Test (0.01000,   0.0)............................................... ok.
atmospheric reflectivity (0.00001,   0.1)................................... ok.
IPA and correlated_k (0.00100,   0.2)....................................... ok.
cloudcover and correlated_k (0.00100,   1.8)................................ ok.
cloudcover, correlated_k and redistribution (0.00100,   1.8)................ ok.
wc_ipa_files and correlated_k (0.00100,   0.1)............................... ok.
wc_ipa_files, correlated_k and redistribution (0.00100,   0.1).............. ok.
wc_ipa_files, ic_ipa_files and redistribution (0.00001,   0.1).............. ok.
heating rates and wc_ipa_files (0.00100,   0.2)............................. ok.
cooling rates and wc_ipa_files (0.00100,   1.9)............................. ok.
transmittance_wl_file (0.00100,   0.1)...................................... ok.
raman (0.10000,   0.1)...................................................... ok.
fluorescence (0.10000,   0.1)............................................... ok.
reptran_thermal (0.01000,   0.1)............................................ ok.
reptran_solar (0.01000,   0.2).............................................. ok.
tzs (0.01000,   0.1)........................................................ ok.
disort_spherical_albedo (0.01000,   0.1).................................... ok.
MYSTIC polarisation (0.00001,   3.0)........................................ ok.
MYSTIC polarized surface reflection (BPDF) (0.00001,   1.0).....................    1 ↵
    serious differences.  Maximum difference:   1.030000%. Absolute values (test,ans): ↵
    (-7.743240e-03, -7.663602e-03).
MYSTIC backward polarized surface reflection (BPDF) (0.00001,   1.0)........... ok.
MYSTIC spectral importance sampling (REPTRAN) (0.00001,   5.0)................. ok.
MYSTIC boxairmass factor (0.00001,   1.0)................................... ok.
MYSTIC spherical (0.00001,   2.0)........................................... ok.
NCA 1.0 (0.00001,   0.1).................................................... ok.
NCA 2.0 cuboid (0.00001,   0.1)............................................. ok.
NCA 2.0 triangle (0.00001,   0.1)........................................... ok.
1 of the \lstinline{uvspec} tests failed.
make[1]: Leaving directory '/home/dgriffith/libRadtran/libRadtran-2.0.3/test'
(devpy27) dgriffith@nimbus:~/libRadtran/libRadtran-2.0.3$
```

From the above test report it is evident that the MYSTIC polarized surface reflection Bidirectional Polarization Distribution Functions (BPDF) test had a result error difference

of 1.03%. All the other tests passed.

On the matter of tests failing in the check phase, see
`http://www.libradtran.org/doku.php?id=faq`
where it states:
How serious are serious differences reported by "make check"?
They are usually not as serious as it sounds. We haven't had a case where libRadtran worked on one computer and produced really wrong results on another. However, radiative transfer equation solvers like disort are numerical methods which are affected by the limited numerical precisions of processors, and these differ from processor to processor and sometimes even from compiler to compiler. Usually, large differences occur for small numbers. As an example, in the near-infrared the diffuse downward irradiance is very small while the direct beam source is large - hence one may expect some uncertainty in the small diffuse radiation.

3.4 Introductory Example Use

The following is taken (with some adaptation) from
`http://www.libradtran.org/doku.php?id=basic_usage`

Those users who are not familiar with the predecessor of libRadtran(), `uvspec`, please note the following: The central program of the package is an executable called `uvspec` which can be found in the bin directory. If you are interested in a user-friendly program for radiative transfer calculations, `uvspec` is the software you want to become familiar with. A description of `uvspec` is provided in the first part of the manual. Examples of its use, including various input files and corresponding output files for different atmospheric conditions, are provided in the examples directory. For a quick try of `uvspec` go to the examples directory and run `uvspec`:

```
cd libRadtran-2.0.3/examples
../bin/uvspec < UVSPEC_CLEAR.INP > test.out
```

The input file for this example run is given in `UVSPEC_CLEAR.INP`

```
                        # Location of atmospheric profile file.
atmosphere_file ../data/atmmod/afglus.dat
                        # Location of the extraterrestrial spectrum
source solar ../data/solar_flux/atlas_plus_modtran
mol_modify O3 300. DU    # Set ozone column
day_of_year 170          # Correct for Earth-Sun distance
albedo 0.2               # Surface albedo
sza 32.0                 # Solar zenith angle
rte_solver disort        # Radiative transfer equation solver
number_of_streams  6     # Number of streams
wavelength 299.0 341.0   # Wavelength range [nm]
slit_function_file ../examples/TRI_SLIT.DAT
                        # Location of slit function
spline 300 340 1         # Interpolate from first to last in step

quiet
```

This input file yields the following output file (the details of which you can decipher from the libRadtran User Guide [9]):

```
  300.000  2.763049e+00  3.087059e+00  1.170022e+00  2.592736e-01  4.777781e-01  1.862147e-01
  301.000  5.223602e+00  5.888303e+00  2.222381e+00  4.901621e-01  9.168083e-01  3.537029e-01
  302.000  6.212306e+00  7.024819e+00  2.647425e+00  5.829381e-01  1.098937e+00  4.213508e-01
  303.000  1.499798e+01  1.709326e+01  6.418247e+00  1.407351e+00  2.687637e+00  1.021496e+00
  304.000  1.731212e+01  1.968946e+01  7.400318e+00  1.624501e+00  3.108258e+00  1.177797e+00
  305.000  2.666450e+01  3.052658e+01  1.143822e+01  2.502091e+00  4.841135e+00  1.820449e+00
```

```
306.000  2.714423e+01  3.094949e+01  1.161874e+01  2.547107e+00  4.925832e+00  1.849181e+00
307.000  3.892420e+01  4.444759e+01  1.667436e+01  3.652493e+00  7.100925e+00  2.653807e+00
308.000  4.928456e+01  5.616349e+01  2.108961e+01  4.624668e+00  9.003217e+00  3.356516e+00
309.000  4.968203e+01  5.629452e+01  2.119531e+01  4.661965e+00  9.051084e+00  3.373338e+00
310.000  5.282494e+01  5.981413e+01  2.252781e+01  4.956883e+00  9.651869e+00  3.585413e+00
311.000  9.116537e+01  1.021252e+02  3.865812e+01  8.554597e+00  1.652179e+01  6.152631e+00
312.000  8.539021e+01  9.519187e+01  3.611642e+01  8.012679e+00  1.544426e+01  5.748106e+00
313.000  1.008859e+02  1.116138e+02  4.249995e+01  9.466736e+00  1.815837e+01  6.764078e+00
314.000  1.140307e+02  1.247804e+02  4.776222e+01  1.070019e+01  2.035162e+01  7.601594e+00
315.000  1.217260e+02  1.323822e+02  5.082165e+01  1.142229e+01  2.164887e+01  8.088517e+00
316.000  1.001420e+02  1.074433e+02  4.151704e+01  9.396925e+00  1.761202e+01  6.607643e+00
317.000  1.555369e+02  1.652335e+02  6.415407e+01  1.459496e+01  2.715088e+01  1.021044e+01
318.000  1.362324e+02  1.430189e+02  5.585025e+01  1.278350e+01  2.355464e+01  8.888844e+00
319.000  1.611656e+02  1.681876e+02  6.587064e+01  1.512314e+01  2.776797e+01  1.048364e+01
320.000  1.788111e+02  1.816904e+02  7.210032e+01  1.677893e+01  3.006627e+01  1.147512e+01
321.000  1.785449e+02  1.818189e+02  7.207275e+01  1.675394e+01  3.015396e+01  1.147073e+01
322.000  1.869200e+02  1.855010e+02  7.448420e+01  1.753983e+01  3.083632e+01  1.185453e+01
323.000  1.653979e+02  1.624324e+02  6.556606e+01  1.552028e+01  2.706200e+01  1.043516e+01
324.000  2.089290e+02  2.038225e+02  8.255030e+01  1.960507e+01  3.403920e+01  1.313829e+01
325.000  2.134198e+02  2.024978e+02  8.318353e+01  2.002647e+01  3.389131e+01  1.323907e+01
326.000  2.853074e+02  2.686987e+02  1.108012e+02  2.677211e+01  4.507038e+01  1.763456e+01
327.000  2.888047e+02  2.682587e+02  1.114127e+02  2.710029e+01  4.510012e+01  1.773188e+01
328.000  2.730194e+02  2.472749e+02  1.040589e+02  2.561906e+01  4.166240e+01  1.656148e+01
329.000  3.073699e+02  2.764369e+02  1.167614e+02  2.884237e+01  4.668488e+01  1.858315e+01
330.000  3.460204e+02  3.062719e+02  1.304585e+02  3.246919e+01  5.183652e+01  2.076311e+01
331.000  2.989884e+02  2.580901e+02  1.114157e+02  2.805589e+01  4.377979e+01  1.773236e+01
332.000  3.145912e+02  2.690056e+02  1.167194e+02  2.951999e+01  4.573355e+01  1.857646e+01
333.000  3.134522e+02  2.635762e+02  1.154057e+02  2.941311e+01  4.491315e+01  1.836738e+01
334.000  3.062941e+02  2.523394e+02  1.117267e+02  2.874142e+01  4.309492e+01  1.778186e+01
335.000  3.283018e+02  2.671566e+02  1.190917e+02  3.080654e+01  4.572950e+01  1.895403e+01
336.000  2.771408e+02  2.219277e+02  9.981371e+01  2.600579e+01  3.807083e+01  1.588585e+01
337.000  2.736382e+02  2.146446e+02  9.765657e+01  2.567713e+01  3.690510e+01  1.554253e+01
338.000  3.155172e+02  2.437741e+02  1.118583e+02  2.960688e+01  4.200866e+01  1.780280e+01
339.000  3.408671e+02  2.595133e+02  1.200761e+02  3.198561e+01  4.481992e+01  1.911070e+01
340.000  3.823216e+02  2.855006e+02  1.335645e+02  3.587555e+01  4.941755e+01  2.125744e+01
```

This newly-calculated output file differs somewhat from the reference output file `UVSPEC_CLEAR.OUT` present in the `examples` folder. The differences are small (fifth decimal), in the third column. This is within the allowable variance limit for testing the installation.

3.5 Status Review

After the above installation there is a running libRadtran version 2.0.3 running on the server. The next step is to set up Dask to use this server.

Chapter 4

libraddask Setup

4.1 Set up Python for Dask

4.1.1 Creating the Python 3 environment

Create an environment with Anaconda with the appropriate packages and versions to use for Dask. In this case we are using Python 3.7 to run the Dask scheduler and workers on the server. Exactly the same versions must be installed on the server and local PC.

On the local Windows computer:

```
conda create --name mort python=3.7 anaconda
```

On the remote Linux server:

```
conda create --name mordevpy37 python=3.7 anaconda
```

4.1.2 Additional packages

Ensure that the following packages of the same version numbers are installed on all the computers. Some or all of these packages may already be present as part of the full Anaconda installation. The package numbers installed at the current time on our computers are also listed. Note that this list only covers the minimum Dask requirements, your local application may require other packages as well.

```
'dask': '2.12.0'
'distributed': '2.12.0'
'msgpack': '0.6.1'
'cloudpickle': '1.3.0'
'tornado': '6.0.4'
'toolz': '0.10.0'
'numpy': '1.18.1'
```

Search for packages here: `https://anaconda.org/anaconda/` for standard Anaconda packages and `https://anaconda.org/conda-forge/` for community-supported packages.

Note that `msgpack` is known in conda as `msgpack-python`.

To see a list of the currently installed packages on a computer type one of the following, to see versions for all package or a specific package:

```
conda list
conda list dask
```

To install a specific version of a package use the command, with appropriate replacement of the package name and version number:

```
conda install package=version

conda install dask=2.12.0
conda install distributed=2.12.0
conda install msgpack-python=0.6.1
```

4.1.3 Install libraddask

The libraddask library is normally cloned from the GitHub repository and not installed via PyPi or conda. Create a folder into which libraddask can be cloned. In this case assume the folder on a Windows computer is `V:\work\WorkN`. Change the current working directory to this newly-created folder and clone the libraddask repository from GitHub. In a command window do:

```
v:
cd \work\WorkN
git clone https://github.com/NelisW/libraddask.git
```

This should clone libraddask into the `V:\work\WorkN\libraddask` folder.

To tell Python where to find this library create a file with the filename `\libraddask.pth` in the `site-packages` folder of the same environment where dask present. When Python is searching for the location of the libraddask module, it will find this file, which will indicate the location where the module is installed. To see where the `site-packages` folder for the dask conda environment is, activate the environment and execute this code:

```
    from distutils.sysconfig import get_python_lib
    print(get_python_lib())
```

On my **Windows** computer the file is saved in the environment here:

```
    C:/ProgramData/Anaconda3/envs/mort/lib/site-packages\libraddask.pth
```

The `\libraddask.pth` file must have only one line, and this line must be the libraddask library location. In my case the library was cloned from the repository into the following folder, and this must be the contents of the file:

```
    V:\work\WorkN
```

On the **Linux** computer (the nimbus server in this case) the file is saved in the environment here:

```
/home/dgriffith//anaconda2/envs/mordevpy37/lib/python3.7/site-packages/↵
   libraddask.pth
```

The file contents must be a full path, the '∼/libraddask' form does not seem to work:

```
/home/dgriffith/libraddask
```

4.1.4 Prepare for Jupyter

If the jupyter notebook will be used in your work, install the Jupyter notebook in Python.

```
conda install notebook ipykernel
ipython kernel install
```

See here for more detail:
https://github.com/NelisW/ComputationalRadiometry/blob/master/00-Installing-Python-pyradi.ipynb.

Chapter 5

Server User Manual

5.1 libraddask Template for Radiative Transfer

The User Manual is by way of example in using the libraddask server.

This notebook tests the main functionality and demonstrates the use of the libraddask library. The notebook is available in the `/libraddask/doc` folder as (`libraddaskTemplate.ipynb`).

It calculates the radiative transfer calculations for the Sentinel 3 overpass over Roodeplaat (25°37′29″S, 28°21′39″E) on Sunday 2016-06-05.

5.2 Running libRadtran

1. If working off-site, use a Virtual Private Network (VPN) client to connect to the same network as used by the server. Dask requires that your local PC must have an IP number in the same subnetwork as the server. Some VPNs do not provide an IP number on the same subnetwork and such VPN clients will not work with Dask.
2. Using a remote terminal client on your local computer, open three terminal windows on the server:
   - a terminal for general use,
   - a terminal for the Dask scheduler, and
   - a terminal for the Dask workers.
3. In the scheduler terminal activate the conda environment with the Dask packages and then start the scheduler.

   ```
   conda activate mordevpy37
   dask-scheduler
   ```

   The scheduler responds with something like this:

   ```
   (base) dgriffith@nimbus:~$ conda activate mordevpy37
   (mordevpy37) dgriffith@nimbus:~$ dask-scheduler
   distributed.scheduler - INFO - -----------------------------------------------
   distributed.scheduler - INFO - Local Directory:    /tmp/scheduler-ay08oqkc
   distributed.scheduler - INFO - -----------------------------------------------
   distributed.scheduler - INFO - Clear task state
   distributed.scheduler - INFO -   Scheduler at:  tcp://146.64.246.94:8786
   distributed.scheduler - INFO -   dashboard at:                     :8787
   ```

Observe the scheduler URI: `tcp://146.64.246.94:8786`, this must be used in the next step to set up the workers.

4. In the worker terminal change to the `bin` folder in the libRadtran installation folder and activate the conda environment with the Dask packages and then start the worker, using the scheduler URI and setting the required number of processors on the cluster :

```
cd libRadtran/libRadtran-2.0.3/bin
conda activate mordevpy37
dask-worker tcp://146.64.246.94:8786 --nprocs 8
```

This should start eight workers, each with a different port number (not all detail shown below):

```
(base) dgriffith@nimbus:~$ cd libRadtran/libRadtran\-2\.0\.3/bin
(base) dgriffith@nimbus:~/libRadtran/libRadtran\-2\.0\.3/bin$ conda activate mordevpy37
(mordevpy37) dgriffith@nimbus:~/libRadtran/libRadtran-2.0.3/bin$ dask-worker tcp↵
    ://146.64.246.94:8786 --nprocs 8
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:33671'
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:46725'
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:42509'
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:42983'
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:43735'
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:36947'
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:40075'
distributed.nanny - INFO -         Start Nanny at: 'tcp://146.64.246.94:40495'
...
distributed.worker - INFO -       Start worker at:  tcp://146.64.246.94:36533
distributed.worker - INFO -          Listening to:  tcp://146.64.246.94:36533
distributed.worker - INFO -          dashboard at:        146.64.246.94:43069
distributed.worker - INFO - Waiting to connect to:   tcp://146.64.246.94:8786
distributed.worker - INFO - -------------------------------------------------
distributed.worker - INFO -               Threads:                          4
distributed.worker - INFO -                Memory:                   4.21 GB
distributed.worker - INFO -       Local Directory: /home/dgriffith/libRadtran/↵
    libRadtran-2.0.3/bin/dask-worker-space/worker-_0o0qta1
distributed.worker - INFO - -------------------------------------------------
distributed.worker - INFO -         Registered to:   tcp://146.64.246.94:8786
distributed.worker - INFO - -------------------------------------------------
distributed.core - INFO - Starting established connection
...
distributed.worker - INFO -       Start worker at:  tcp://146.64.246.94:46597
distributed.worker - INFO -          Listening to:  tcp://146.64.246.94:46597
distributed.worker - INFO -          dashboard at:        146.64.246.94:33737
distributed.worker - INFO - Waiting to connect to:   tcp://146.64.246.94:8786
distributed.worker - INFO - -------------------------------------------------
distributed.worker - INFO -               Threads:                          4
distributed.worker - INFO -                Memory:                   4.21 GB
distributed.worker - INFO -       Local Directory: /home/dgriffith/libRadtran/↵
    libRadtran-2.0.3/bin/dask-worker-space/worker-n0anjq00
distributed.worker - INFO - -------------------------------------------------
distributed.core - INFO - Starting established connection
distributed.worker - INFO -         Registered to:   tcp://146.64.246.94:8786
distributed.worker - INFO - -------------------------------------------------
distributed.core - INFO - Starting established connection
```

5. Run the code that activates the client in the local PC (see the code further down below). This should send the serialised data to the scheduler:

```
distributed.scheduler - INFO - Register worker <Worker 'tcp://146.64.246.94:33437', ↵
    name: tcp://146.64.246.94:33437, memory: 0, processing: 0>
distributed.scheduler - INFO - Starting worker compute stream, tcp↵
    ://146.64.246.94:33437
distributed.core - INFO - Starting established connection
distributed.scheduler - INFO - Receive client connection: Client-1ed95fc0-8076-11ea↵
    -8700-cfd8862f98eb
distributed.core - INFO - Starting established connection
distributed.scheduler - INFO - Receive client connection: Client-27b067b0-8076-11ea↵
    -8700-cfd8862f98eb
distributed.core - INFO - Starting established connection
```

The scheduler will send the serialised data to the workers, which will execute the tasks:

```
Running AerAng600to900nm
Running AerAng650to800nm
```

If the libRadtran execution was successful the tasks should complete and the data returned to the client.

5.3 Prepare for Python

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import datetime
import pytz
import copy # required for deepcopy

from dask.distributed import Client # For contacting the dask scheduler
import libraddask.rad.librad as librad

import pprint
pp = pprint.PrettyPrinter(indent=4)

%matplotlib inline
```

5.4 Set up and Execute a Scenario or Case

Populate an instance of the `librad.Case` class. Start with an empty instance and then add options to obtain the required input file for libRadtran.

Note that this code will be executed in the server's filesystem in
`/home/dgriffith/libRadtran/libRadtran-2.0.3/bin/`
folder, so the paths must be relative to this executing location. The paths defined here has no bearing to any folder on the local computer.

```
# Create a blank libRadtran case
# with a name for this case
S3 = librad.Case(casename='AerAng600to900nm')

# Set revision
revision = '00A'

# Choose basic atmospheric profile
atmos_profile = '../data/atmmod/afglmw.dat'
# mid-latitude winter standard atmosphere
S3.set_option('atmosphere_file', atmos_profile)

# Change to the Thuillier spectrum and set wavelength range appropriately
# these files must be present on the server, relative to the bin directory
solar_toa_file = '../data/solar_flux/Solar_irradiance_Thuillier_2002.txt'
solar_toa_file = '../data/solar_flux/kurudz_1.0nm.dat'

# Choose start and stop wavelengths and minimum edge margin in nm
wv_minimum_range = [[385.0, -2.0], [955.0, 2.0]]
S3.set_option('source solar', solar_toa_file)
S3.set_option('wavelength', 600.0, 900.0)

# Set up dates and times
# Overpass date and time down to second
overpass_datetime = datetime.datetime(2016, 6, 5, 7, 42, 31, tzinfo=pytz.utc)
overpass_datestr = overpass_datetime.strftime('%Y%m%d')
# Get the day of year
day_of_year = int(overpass_datetime.strftime('%j'))
```

```
results_folder = 'ResultsS3on' + overpass_datestr + 'Rev' + revision

# Choose solver
S3.set_option('rte_solver disort')

# Set ground altitude
S3.set_option('altitude', 1.225)  # ground altitude in km above sea level

# Set ground albedo
S3.set_option('albedo 0.5')

# Set up aerosol model using the Angstrom law
S3.set_option('aerosol_default')
aot_wv = np.array([440, 500, 675, 870], dtype=np.float)  # MicroTOPS measurement wavelengths
aot = np.array([0.703, 0.615, 0.362, 0.206])  # MicroTOPS measurements
# Fit Angstrom law to data
alpha, beta = librad.angstrom_law_fit(aot_wv, aot)
# Fit King Byrne formula
alpha_0, alpha_1, alpha_2 = librad.king_byrne_formula_fit(aot_wv, aot)
S3.set_option('aerosol_angstrom', alpha, beta)

# Set up viewing and solar geometry. Note that these angles are taken from the S3 product and↵
     special
# care has to be taken when putting geometry information into libRadtran
OAA = 104.01066  # deg. Observation azimuth angle (presumably relative to north through east,↵
     satellite from dam)
OZA = 14.151356  # deg. Observation zenith angle (satellite zenith angle as seen from the dam↵
    )
SAA = 38.719933  # deg. Solar azimuth angle (presumably relative to north through east)
SZA = 59.316036  # deg. Solar zenith angle

S3.set_option('sza', SZA)  # deg. This one is straightforward
# Now when entering solar and observation zenith angles, it is necessary to provide the ↵
    azimuth of light propagation
# rather than the azimuth of the view direction, which is 180 deg different
#S3.set_option('phi0', 180.0 - SAA)  # solar radiation propagation azimuth from north through↵
     east
#S3.set_option('phi', OAA)  # This is the azimuth of the satellite as seen from the target - ↵
    also azimuth of light propgation
#S3.set_option('umu', np.cos(np.deg2rad(OZA))) # For downward-looking (upward propagating), ↵
    check that umu is positive

S3.set_option('verbose')  # Will produce a lot of diagnostic output on stderr

#S3.set_option('zout boa')  # Set altitude of output data
S3.purge = False  # Prevent purging of output files
```

```
# make a new case as a copy of existing
# If you use S3b = S3, both variables point to the same copy
# deepcopy operation required to create a new independent object

S3b = copy.deepcopy(S3)
# define a new spectral band, otherwise the same
S3b.name = 'AerAng650to800nm'
S3b.set_option('wavelength', 650.0, 800.0)
# S3b.set_option('aerosol_visibility', 5.0)
```

```
#print out the current file contents
# Case().__repr__() provides the same output as written to the file.
print(f'S3:\n{S3}\n')
print(f'S3b:\n{S3b}\n')
```

```
S3:
atmosphere_file ../data/atmmod/afglmw.dat
source solar ../data/solar_flux/kurudz_1.0nm.dat
wavelength 600.0 900.0
rte_solver disort
altitude 1.225
albedo 0.5
aerosol_default
aerosol_angstrom 1.6848887536371142 0.18175118840581214
```

```
sza 59.316036
verbose
```

```
atmosphere_file ../data/atmmod/afglmw.dat
source solar ../data/solar_flux/kurudz_1.0nm.dat
wavelength 650.0 800.0
rte_solver disort
altitude 1.225
albedo 0.5
aerosol_default
aerosol_angstrom 1.6848887536371142 0.18175118840581214
sza 59.316036
verbose
```

```
# create dask Client method with scheduler serverURI (IP and port)
serverURI = '146.64.246.94:8786'
client = Client(serverURI)

# count number of workers, obtained from scheduler
number_of_processes = len(client.scheduler_info()['workers'])
print(f'Number of workers: {number_of_processes}')
print(f'Client cores: {client.ncores()} ')

# uncomment to see the detail
# pp.pprint(client.get_versions(check=True))
```

```
Number of workers: 8
Client cores: {'tcp://146.64.246.94:34397': 4, 'tcp↵
    ://146.64.246.94:38009': 4, 'tcp://146.64.246.94:39663': 4, 'tcp↵
    ://146.64.246.94:41169': 4, 'tcp://146.64.246.94:41399': 4, 'tcp↵
    ://146.64.246.94:41797': 4, 'tcp://146.64.246.94:43815': 4, 'tcp↵
    ://146.64.246.94:45191': 4}
```

```
# do a run on the server using two cases
futuresRad = client.map(librad.Case.run, [S3, S3b])
# Gather results. This will wait for completion of all tasks.
S3List = client.gather(futuresRad)

# check the return values
librad.check_uvspec_run_success(S3List)
```

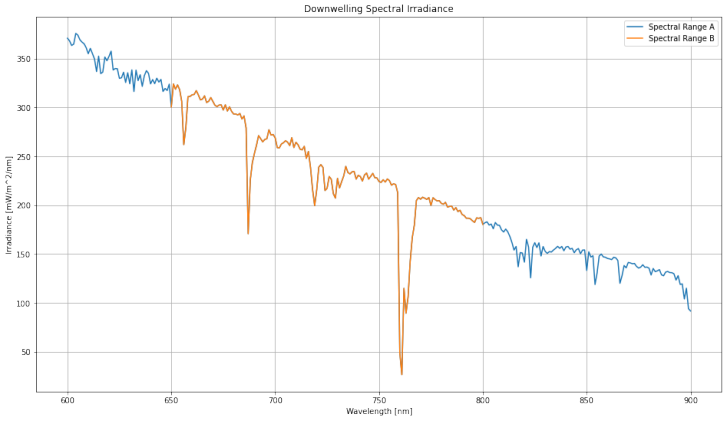
```
All runs successful.
```

```
# extract to
S3 = S3List[0]
S3b = S3List[1]
```

```
# Check for any errors by printing the return code and anything written to stderr
print(f'Return Codes : {S3.run_return_code} {S3b.run_return_code} ')
# print(f'Run({S3.name}) error (if any):\n{S3.stderr}')
# print(f'Run({S3b.name}) error (if any):\n{S3b.stderr}')
```

```
Return Codes : 0 0
```

```
# now plot on the same graph
plt.figure(figsize=(16,9))
plt.plot(S3.wvl, S3.edn, S3b.wvl, S3b.edn)
plt.xlabel('Wavelength [nm]')
plt.ylabel('Irradiance [mW/m^2/nm]')
plt.title('Downwelling Spectral Irradiance')
plt.legend(['Spectral Range A', 'Spectral Range B'])
plt.grid()
# plt.savefig('ednAerAng.pdf')
```



5.5 Demonstrate King-Byrne and Angstrom Law

The run above was done using the Angstrom law parameters fitted to MicroTOPS measurements.

```
# Check the quality of the Angstrom Law and King-Byrne formula fits
the_wv = np.arange(350.0, 950.0, 10.0)  # Pick a wavelength range
the_aot = librad.king_byrne_formula(the_wv, alpha_0, alpha_1, alpha_2)  # Calculate King-↵
    Byrne
aot_ang = librad.angstrom_law(the_wv, alpha, beta)  # Calculate Angstrom Law
print(f'Angstrom alpha={alpha} beta={beta}')
print(f'king_byrne alpha0={alpha_0} alpha1={alpha_1} alpha2={alpha_2}')
```

```
Angstrom alpha=1.6848887536371142 beta=0.18175118840581214
king_byrne alpha0=-1.9950667229251264 alpha1=-3.0186651652429806 alpha2↵
    =-1.2338155309188534
```

```
# Set up atmospheric model and base case
atm = librad.Case(casename='RoodeplaatTransmittance')

# mid-latitude winter
atm.set_option('atmosphere_file', '../data/atmmod/afglmw.dat')

# solar_toa_file = '../data/solar_flux/Solar_irradiance_Thuillier_2002.txt'
# solar_toa_file = '../data/solar_flux/kurudz_1.0nm.dat'
# atm.set_option('source solar', solar_toa_file)

# Choose solver
atm.set_option('rte_solver', 'disort')

# Set ground altitude (BOA) above sea-level
atm.set_option('altitude', 1.225)  # km AMSL

# Set up aerosol model
atm.set_option('aerosol_default')
# Aerosol type above 2km
# atm.set_option('aerosol_vulcan', 1)
# Angstrom aerosol
atm.set_option('aerosol_angstrom', alpha, beta)

atm.set_option('wavelength', 400.0, 900.0)

SZA = 59.316036  # deg. Solar zenith angle
atm.set_option('sza', SZA)  # deg. This one is straightforward
atm.set_option('output_quantity transmittance')
atm.purge = False  # Prevent purging of output files
```

```
print(f'atm:\n{atm}')
```

```
atm:
atmosphere_file ../data/atmmod/afglmw.dat
rte_solver disort
```

```
altitude 1.225
aerosol_default
aerosol_angstrom 1.6848887536371142 0.18175118840581214
wavelength 400.0 900.0
sza 59.316036
output_quantity transmittance
```

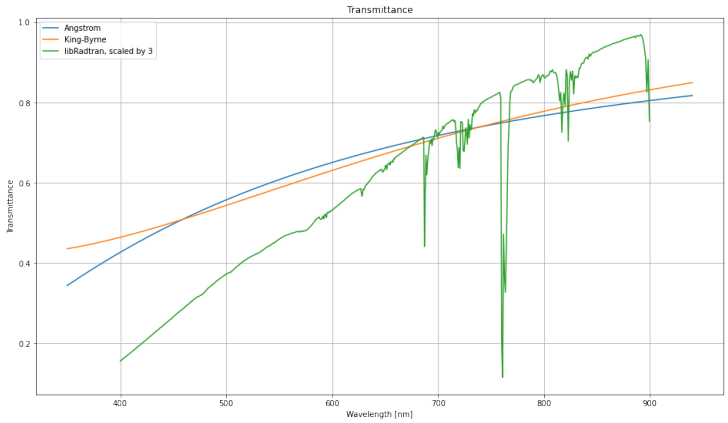
```
# do a run on the server using two cases
futuresRad = client.map(librad.Case.run, [atm])

# Gather results. This will wait for completion of all tasks.
T3List = client.gather(futuresRad)

# check the return values
librad.check_uvspec_run_success(T3List)
```

```
All runs successful.
```

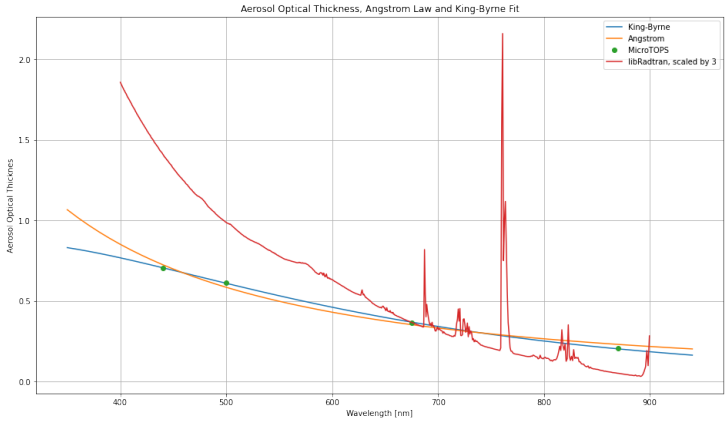
For some unknown reason the transmittance from libRadtran comes out significantly **smaller** than what the Angstrom equation predicts.

```
T3a = T3List[0]
transm = T3a.edir[0:,0]
scaling = 3
txm = transm * scaling
```

```
# Plot transmittance
plt.figure(figsize=(16,9))
plt.plot(the_wv, np.exp(- aot_ang))
plt.plot(the_wv, np.exp(- the_aot))
plt.plot(T3a.wvl, txm)
plt.xlabel('Wavelength [nm]')
plt.ylabel('Transmittance')
plt.title('Transmittance')
plt.legend(['Angstrom','King-Byrne',f'libRadtran, scaled by {scaling}'])
plt.grid()
```




```
# Plot Angstrom Law and King-Byrne fitted curves with MicroTOPS measurements
plt.figure(figsize=(16,9))
plt.plot(the_wv, the_aot, the_wv, aot_ang, aot_wv, aot, 'o')
plt.plot(T3a.wvl, -np.log(txm) )
plt.xlabel('Wavelength [nm]')
plt.ylabel('Aerosol Optical Thicknes')
plt.title('Aerosol Optical Thickness, Angstrom Law and King-Byrne Fit')
plt.legend(['King-Byrne', 'Angstrom', 'MicroTOPS',f'libRadtran, scaled by {scaling}'])
plt.grid()
```



5.6 Split Case

`librad.Case()` has the option to split a wavelength range into smaller sections. These smaller sections can be executed on the server in parallel and the results merged afterwards.

```
# Split up into number of sub-cases depending on the number of cores and
# other factors on the compute cluster
# use wavelength overlap of 1.0 nm
atm_list = atm.split_case_by_wavelength(number_of_processes, overlap=1.0)

# Take a look at the splits
print('file contents:')
print(atm_list[0],end='\n\n')

print(f'Spectral ranges for {number_of_processes} processes:')
for i,item in enumerate(atm_list):
    print(f"process {i} range={item.tokens[item.options.index('wavelength')]}")
```

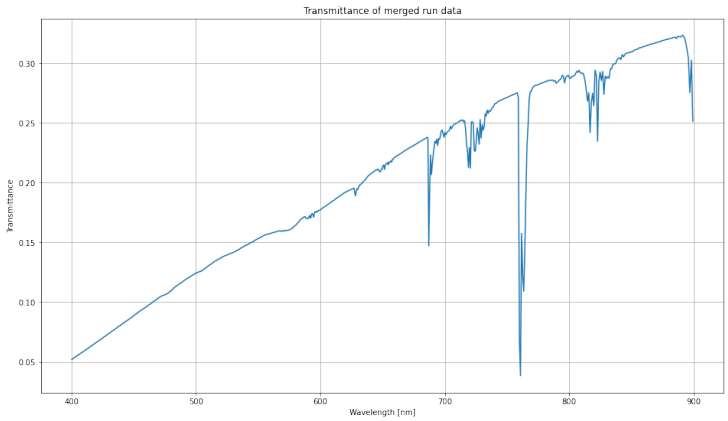
```
file contents:
atmosphere_file ../data/atmmod/afglmw.dat
rte_solver disort
altitude 1.225
aerosol_default
aerosol_angstrom 1.6848887536371142 0.18175118840581214
wavelength 400.0 462.5
sza 59.316036
output_quantity transmittance

Spectral ranges for 8 processes:
process 0 range=['400.0', '462.5']
process 1 range=['461.5', '525.0']
process 2 range=['524.0', '587.5']
process 3 range=['586.5', '650.0']
process 4 range=['649.0', '712.5']
process 5 range=['711.5', '775.0']
process 6 range=['774.0', '837.5']
process 7 range=['836.5', '900.0']
```

```
# run the list of spectral ranges
futureRadBatch = client.map(librad.Case.run, atm_list)
atm_listrtn = client.gather(futureRadBatch)
# check the return values
librad.check_uvspec_run_success(atm_listrtn)
```

```
All runs successful.
```

```
# Merge results for edir, which will be direct (specular) transmittance
# in 'output transmittance' mode
wvl_atm_reptran, atm_reptran = \
    librad.Case.merge_caselist_by_wavelength(atm_listrtn, 'edir')
```

```
# Plot transmittance
plt.figure(figsize=(16,9))
plt.plot(wvl_atm_reptran, atm_reptran)
plt.xlabel('Wavelength [nm]')
plt.ylabel('Transmittance')
plt.title('Transmittance of merged run data')
plt.grid()
```



5.7 Shutting down the Client

When the client is no longer required, shut it down.

BUT: this also kills the scheduler and/or workers!

```
# client.shutdown()
```

```
# to get software versions
# https://github.com/rasbt/watermark
# An IPython magic extension for printing date and time stamps, version numbers, and hardware↵
    information.
# you only need to do this once
# !pip install watermark

%load_ext watermark
%watermark -v -m -p numpy,matplotlib,datetime,pytz,dask,dask.distributed,libraddask -g
```

```
CPython 3.7.6
IPython 7.13.0

numpy 1.18.1
matplotlib 3.2.1
datetime unknown
pytz 2019.3
dask 2.12.0
dask.distributed 2.12.0
libraddask 0.1.1

compiler   : MSC v.1916 64 bit (AMD64)
system     : Windows
release    : 7
```

```
machine    : AMD64
processor  : Intel64 Family 6 Model 94 Stepping 3, GenuineIntel
CPU cores  : 8
interpreter: 64bit
Git hash   : 678c42400523b7d730d6646132e271a7cacd8ab4
```

Chapter 6

Conclusion

The libraddask Python package provides a libRadtran server functionality to Python scripts and Jupyter notebooks. The package is available on GitHub [5].

Bibliography

[1] dask.org, *Dask*, https://docs.dask.org/en/latest/.

[2] Emde, C., Buras-Schnell, R., Kylling, A., Mayer, B., Gasteiger, J., Hamann, U., Kylling, J., Richter, B., Pause, C., Dowling, T., , and Bugliaro, L., *The libRadtran software package for radiative transfer calculations (version 2.0.1)*, *Geoscientific Model Development* 9, 1647–1672 (2016) [doi: DOI:10.5194/gmd-9-1647-2016].

[3] Mayer, B. and Kylling, A., *Technical Note: The libRadtran Software Package for Radiative Transfer Calculations – Description and Examples of Use*, *Atmospheric Chemistry And Physics* 5, 1855–1877 (2005) [doi: DOI:10.5194/acp-5-1855-2005].

[4] Griffith, D. J., *MORTICIA*, https://github.com/derekjgriffith/MORTICIA.

[5] Willers, C. J., *libraddask*, https://github.com/NelisW/libraddask.

[6] Mayer, B., Emde, C., Buras, R., and Kylling, A., *libRadtran — library for radiative transfer*, http://www.libradtran.org.

[7] Mayer, B., Emde, C., Buras, R., and Kylling, A., *libRadtran download*, http://www.libradtran.org/doku.php?id=download.

[8] www.meteo.physik.uni muenchen.de, *libRadtran*, https://www.meteo.physik.uni-muenchen.de/~libradtran/doku.php.

[9] Mayer, B., Kylling, A., Emde, C., Buras, R., Hamann, U., Gasteiger, J., and Richter, B., *libRadtran user's guide (version 2.0.3)*, http://www.libradtran.org.

Appendix A

Simple Dask Examples

A.1 Dask Compute Graphs

A simple Dask compute graph is demonstrated here, taken from `https://towardsdatascience.com/why-every-data-scientist-should-use-dask-81b2b850e15b`

A common pattern I encounter regularly involves looping over a list of items and executing a python method for each item with different input arguments. Common data processing scenarios include, calculating feature aggregates for each customer or performing log event aggregation for each student. Instead of executing a function for each item in the loop in a sequential manner, Dask Delayed allows multiple items to be processed in parallel. With Dask Delayed each function call is queued, added to an execution graph and scheduled.

Writing custom thread handling or using `asyncio` has always looked a bit tedious to me, so I'm not even going to bother with showing you comparative examples. With Dask, you don't need to change your programming style or syntax! You just need to annotate or wrap the method that will be executed in parallel with `@dask.delayed` and call the compute method after the loop code.

In the example below, two methods have been decorated with `@dask.delayed`. Three numbers are stored in a list which must be squared and then collectively summed. Dask constructs a computation graph which ensures that the `square` method is run in parallel and that the output is collated as a list and then passed to the `sum_list` method. The computation graph can be printed out by calling `.visualize()`. Calling `.compute()` executes the computation graph. As you can see in the output, the list items are not processing in order and are run in parallel.

The number of threads can be set (i.e., `dask.set_options( pool=ThreadPool(10))`) and it is also easy to swap to use processes on your laptop or personal desktop (i.e., `dask.config.set( scheduler='processes')`.

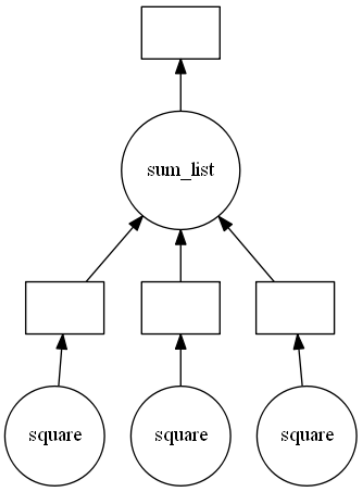
The graphic visualisation of the graph requires:

1. Install graphviz from https://graphviz.gitlab.io/download/
2. Put the path to the graphviz in the PATH environmental variable. In my case it was `C:/Program Files (x86)/Graphviz2.38/bin`

3. Install graphviz in Python conda install graphviz
4. Install python-graphviz in Python conda install python-graphviz

```
from dask import delayed, compute
import dask
```

```
@delayed
def square(num):
    print("square fn:", num)
    print()
    return num * num

@delayed
def sum_list(args):
    print("sum_list fn:", args)
    return sum(args)

items = [1, 2, 3]

computation_graph = sum_list([square(i) for i in items])

computation_graph.visualize()
```




```
print("Result", computation_graph.compute())
```

```
square fn:square fn:square fn: 3

 1

 2

sum_list fn: [1, 4, 9]
Result 14
```

Appendix B

Porting `scr_py` to Python 3

B.1 Original Work

Derek Griffith's original work was done with the Python files for libRadtran 2.0.0. This work will therefore update from 2.0.0 to 2.0.3 as well as Python 3 porting. Griffith's additions were as follows:

1. `aerosol_options.py`: a number of changes:

```
--- W:\temp\libradold\200\aerosol_options.py 2020-04-21  10:34:12
+++ W:\temp\libradold\DerekGitHubMORTICIAmorticiarad\aerosol_options.↵
   py 2020-04-21  11:01:58
@@ -23,28 +23,30 @@
  * Boston, MA 02111-1307, USA.
  *--------------------------------------------------------------------"""↵


+# Modified by Derek Griffith to include aerosol_sizedist_file and ↵
   aerosol_refrac_index
+

 from option_definition import *

@@ -205,216 +207,238 @@
      extra_dependencies=['aerosol_haze','aerosol_vulcan','↵
         aerosol_visibility','aerosol_season'],
    )

+        aerosol_sizedist_file = option(
+            name='aerosol_sizedist_file',
+            group='aerosol',
+            helpstr='Aerosol size distribution file.',
+            documentation=documentation['aerosol_sizedist_file'],
+            tokens=addToken(name='Input.aer.filename[Id]', datatype=↵
   file),
+            childs=['aerosol_option_specification'],
+            extra_dependencies=[],
+        )
+        aerosol_refrac_index = option(
+            name='aerosol_refrac_index',
```

```
+            group='aerosol',
+            helpstr='Set the aerosol particle refractive index.',
+            documentation=documentation['aerosol_refrac_index'],
+            tokens=[addToken(name='Input.aer.nreal', datatype=float)↵
   ,
+                    addToken(name='Input.aer.nimag', datatype=float,↵
    default='NOT_DEFINED_FLOAT', valid_range=[0, 1e6]),
+                    addSetting(name='Input.aer.spec', setting=1)],
+            parents=['aerosol_default'],
+        )
    self.options = [ aerosol_default,
      aerosol_file, aerosol_species_library, aerosol_species_file,
      aerosol_haze, aerosol_season, aerosol_vulcan, ↵
         aerosol_visibility,
      aerosol_profile_modtran,
      aerosol_angstrom,aerosol_king_byrne,
-     aerosol_modify, aerosol_set_tau_at_wvl ]
+                        aerosol_modify, aerosol_set_tau_at_wvl, ↵
   aerosol_sizedist_file,
+                        aerosol_refrac_index]

  def __iter__(self):
    return iter(self.options)
@@ -493,498 +515,536 @@
  \fcode{
     aerosol\_set\_tau\_at\_wvl lambda tau
  }
+   ''',
+
+        'aerosol_sizedist_file': r'''
+        Calculate optical properties from size distribution and ↵
   index of refraction using Mie
+        theory. Here is an exception from the rule that ALL values ↵
   defined above are overwritten
+        because the optical thickness profile is re-scaled so that ↵
   the optical thickness
+        at the first internal wavelength is unchanged. It is done ↵
   that way to give the user an
+        easy means of specifying the optical thickness at a given ↵
   wavelength.
+        ''',
+
+        'aerosol_refrac_index': r'''
+        Calculate optical properties from size distribution and ↵
   index of refraction using Mie
+        theory. Here is an exception from the rule that ALL values ↵
   defined above are overwritten
+        because the optical thickness profile is re-scaled so that ↵
   the optical thickness
+        at the first internal wavelength is unchanged. It is done ↵
   that way to give the user an
+        easy means of specifying the optical thickness at a given ↵
   wavelength.
    ''',
 }
```

2. `spectral_options.py`: added `wavelength_step` option:

```
--- W:\temp\libradold\200\spectral_options.py 2020-04-21  10:34:14
```

```
+++ W:\temp\libradold\DerekGitHubMORTICIAmorticiarad\spectral_options↵
    .py 2016-09-23  11:33:02
@@ -46,51 +46,62 @@
      non_parents=['wavelength_index'],
    )

+   wavelength_step = option(
+     name='wavelength_step',
+     group='spectral',
+     helpstr='Set the wavelength step (in nm) in conjunction with ↵
    the wavelength range.',
+     documentation=['wavelength_step'],
+     gui_inputs=(),
+     tokens=[],
+     parents=['uvspec'],
+     non_parents=['wavelength_index'],
+   )
+
    wavelength_index = option(
      name='wavelength_index',
      group='spectral',
```

B.2 Porting

The following work was done:

1. Change all print statement lines to print functions.
2. Python 3 changed the local import syntax. The libRadtran files are in a separate folder, which requires am awkward means to specify the folder. All local imports must be of the form from `libraddask.libradtran import option_definition`.

   I also opted to not do a star import, which means that you must provide 'a path' such as `option_definition.option` instead of the starred import version `option`. This change required a large number of changes throughout the code.
3. Updated Python 2 `file` type tests to Python 3 `io.IOBase` tests. For more detail see: `https://docs.python.org/3/library/io.html#io.IOBase`,
   "the abstract base class for all I/O classes, acting on streams of bytes."
4. Changed exceptions: `except Exception, e:` to `except Exception as e:`
5. Changed data type from `double` to `float`.
   Changed data type from `long` to `int`.
6. The following files were changed with the original additions by Derek Griffith (see above): `aerosol_options.py`, `spectral_options.py`. These changes will be applied after all the other Python conversion modifications.

B.3 New Code Testing

The following tests were done on revision `678c42400523b7d730d6646132e271a7cacd8ab4`. These tests are not exhaustive or very deep. The focus was more on determining if the porting to Python 3 did not cause breaking changes.

1. `libraddask_test.py`. The libRadtran files were scanned and all possible options that can be set. The test entails creating an `empty libraddask.rad.librad.Case` instance and then setting the options available by `set_option()` observing the resulting `case` output. Some options coded in the files were not available and such `set-option()` lines were commented out. The data used in this test are not representative of values that have any physical significance, do this test must not be seen as a tutorial on how to use.

2. The notebook shown in Section 5 was executed and checked for correct operation. It executed all required functionality.

   One issue arose: the libRadtran results different from transmittance results calculated for an atmosphere with a given set of Angstrom coefficients. This is not a package issue, it is a libRadtran issue, still to be resolved.

3. A number of internal (proprietary) notebooks were executed against this version of the package and all notebooks completed successfully.

Appendix C

Tools

C.1 Remote Server Access From Windows

With the libraddask server running on a remote Linux computer, some tools would be required to access the server from a local Windows computer. There are several options available, from simple terminals to complex X-server applications. Consider one of the following:

`https://mobaxterm.mobatek.net/` (free for personal user)

`http://www.straightrunning.com/XmingNotes/` (free)

`https://www.netsarang.com/en/xshell/` (free for personal user)

C.2 Remote Editing

If you feel adventurous, try so use Visual Studio Code to edit files on the server
`https://www.petri.com/how-to-edit-linux-files-remotely-in-windows-using-visual-studio-code`

C.3 Shell Scripts

On the nimbus server create the following shell scripts and mark them as executable. Then run the scripts to speed up the process of starting the scheduler and worker tasks. The scripts should be run by the `source` mechanism otherwise the `conda` execution does not work.

`dasksched.sh`

```
conda activate mordevpy37
dask-scheduler
```

`daskwork.sh` (assuming that the scheduler is running on `tcp://146.64.246.94:8786`)

```
cd libRadtran/libRadtran-2.0.3/bin
conda activate mordevpy37
dask-worker tcp://146.64.246.94:8786 --nprocs 8
```

Then, in two different terminals, run them by

```
source scriptname.sh
```