Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Ausarbeitung

Python Dask

vorgelegt von

Alena Pils

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Meteorologie
Matrikelnummer: 6773951

Betreuer: Jannek Squar

Hamburg, 2020-08-18

Inhaltsverzeichnis

1. Einleitung

In manchen Fällen ist die Analyse von Daten nötig, die den Arbeitsspeicher des eigenen Laptops übersteigen. Ein Beispiel für solche Datenmengen ist die ERA5 Reanalyse. Dieser Datensatz umfasst eine Vielzahl an Klimavariablen mit einer stündlichen Auflösung von 1979 bis heute [fMRWF]. Die räumliche Auflösung liegt global bei ca. 30 km und die Atmosphäre umfasst 137 vertikale Schichten. Diese Auflösung führt dazu, dass auch bei der Auswahl einzelner Klimavariablen mehrere Gigabyte an Daten zusammenkommen. Insgesamt umfasst der Datensatz ungefähr neun Petabyte.

Die Analyse weniger Klimavariablen auf dem eigenen Laptop führt daher schnell dazu, dass die Datenmenge den Arbeitsspeicher übersteigt. In Python kommt es in diesem Fall zu einem Memory Error. Eine Möglichkeit die Analyse trotzdem mit Python durchzuführen liefert Dask.

Dask ist eine Bibliothek zur Parallelen Programmierung in Python [Das19]. Dask ist dann sinnvoll, wenn die Datenmenge den Arbeitsspeicher übersteigt. Tut sie das nicht, sind meistens Tools wie beispielsweise Pandas schneller [AcRc18]. Pandas ist eine Bibliothek zur Datenanalyse. Dabei ist es sowohl möglich Dask auf dem eigenen Laptop zu nutzen als auch auf einem Cluster.

Das Grundprinzip von Dask ist das Folgende [AcRa18a]: Data Collections, wie z.B. Arrays, erzeugen einen sogenannten Task Graphen (Abb. 1.1). Dieser Task Graph zeigt die für das Ergebnis aufzurufenden Funktionen auf. Die Ausführung des Task Graphen, also die Berechnung des Ergebnisses, geschieht dann durch den Scheduler. Als Collections möglich sind Dask Array, Dask DataFrame, Dask Bag, Dask Delayed und Futures. Ein DataFrame ist eine tabellenähnliche Collection (siehe Ausarbeitung von Larissa Lach) und ein Bag ist eine ungeordnete Collection. Dask Delayed ist zur Parallelisierung von Funktionen hilfreich. Bei Futures wird im Gegensatz zu Dask Delayed das Ergebnis direkt im Hintergrund berechnet, wodurch die Zelle sofort zurückgegeben wird. Dies ist nur über verteilte Cluster (distributed) möglich.

In dieser Ausarbeitung wird der Fokus auf Dask Array, DataFrame und Delayed gelegt. Im Anschluss werden die Scheduler vorgestellt. Zu jedem Thema befindet sich im Anhang ein kurzes Beispielprogramm, in dem ERA5-Daten analysiert werden.

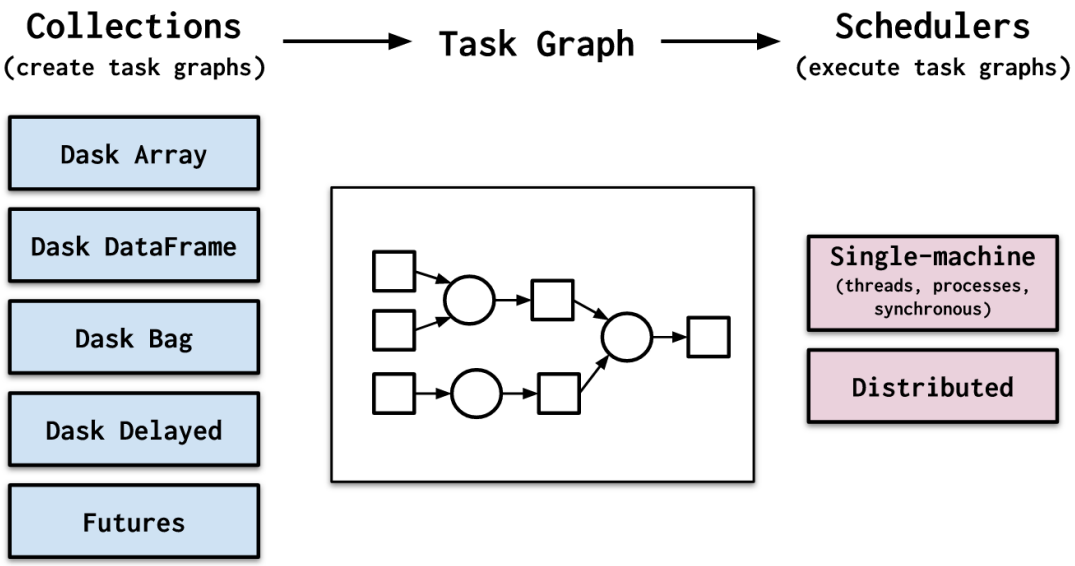


Abbildung 1.1.: Grundprinzip von Dask. Collections erzeugen sogenannte Task Graphen, die zur Berechnung des Ergebnisses mithilfe von Scheduler ausgeführt werden [AcRa18a].

2. Data Collections

2.1. Dask Array

Ein Dask Array ist dem NumPy Array ähnlich [Dev18a]. NumPy ist eine Python Bibliothek, die große, mehrdimensionale Arrays unterstützt und eine Sammlung vieler mathematischer Funktionen für diese Arrays beinhaltet. Genaueres über NumPy ist der Ausarbeitung von Larissa Lach zu entnehmen. Im Gegensatz zu NumPy Arrays ermöglichen Dask Arrays parallele Berechnungen sowie Arrays, die größer als der Arbeitsspeicher sind [Dev18a]. Dabei nutzen Dask Arrays die Aufteilung der Daten in Blöcke (Abb. 2.1). Ein Beispiel, in dem die Aufteilung in Blöcke sinnvoll ist, ist das Folgende [Dev18a]: Angenommen es soll die Summe über eine große Zahl an Elementen gebildet werden. Dann können die gesamten Daten auf mehrere Datenblöcke aufgeteilt werden. Es wird zunächst die Summe für jeden einzelnen Datenblock gebildet und im Anschluss die Summe über die Datenblöcke.

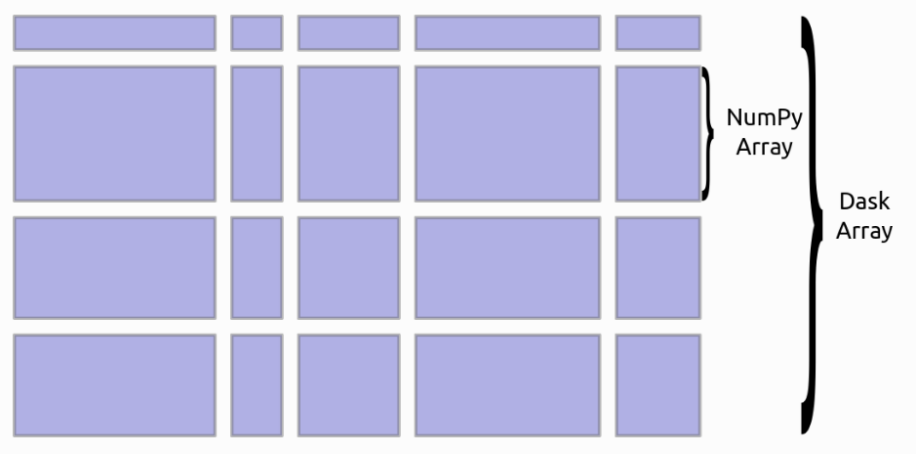


Abbildung 2.1.: Dask Array bestehend aus mehreren NumPy Arrays [AcRa18b].

Die Syntax von Dask Arrays ist der von NumPy Arrays ähnlich (Abb. 2.2). Soll ein Dask Array erstellt werden, muss zunächst *dask.array* importiert werden *import dask.array as ds*. Anschließend lässt sich, wie bei NumPy Arrays, auf verschiedene Weise ein Array erzeugen. So wird beispielsweise mit *ds.arange(10)* ein eindimensionales Array erzeugt, das Einträge mit den Zahlen von null bis neun hat. Im Gegensatz zu NumPy Arrays wird dies bei Dask Arrays nicht direkt mit der Funktion ausgeführt, sonderen erst eine Art Task Graph erstellt, der die Aufteilung der Daten aufzeigt. Erst durch

Aufruf der *compute*-Funktion ergibt sich das gleiche Ergebnis, wie bei den NumPy Arrays. Mithilfe des Parameters *chunks* lässt sich die Aufteilung des Dask Arrays in NumPy Arrays festlegen (Abb. 2.3). Übliche NumPy Operationen funktionieren mit Dask Arrays genauso (Abb. 2.2). So wird beispielsweise mit *x[::2]* jeder zweite Wert des Arrays ausgewählt und mit *x[::2].mean()* der Mittelwert dieser Werte gebildet. Das konkrete Ergebnis wird bei Dask Arrays erst mit Aufruf der *compute*-Funktion ausgegeben.

Ein Beispiel für die Anwendung von Dask Arrays ist das Darstellen und die Berechnung des zeitlichen Mittels von der 2 m Temperatur aus den bereits erwähnten ERA5-Daten. Um dies tun zu können, müssen zunächst die benötigten Bibliotheken importiert werden (Listing A.1). Das sind für die Darstellungen *matplotlib.pyplot* und zum Arbeiten mit Dask Arrays *xarray*. *xarray* ist nicht direkt Dask, verwendet aber Dask und erleichtert das Einlesen von meteorologischen Daten. Mit xarray werden dann die Daten über *xarray.open_dataset(daten, chunks='lat':-1, 'lon':-1, 'time': 50000)* eingelesen. Die Daten ist in diesem Fall die Temperatur von 1979-2019 eines Gebietes bei der Nordsee. Durch das Angeben der Chunks werden die Daten entlang der Zeit aufgeteilt (Abb. A.1). Die Temperatur ist als Dask Array ohne konkrete Werte eingelesen. Mit *ds.temperature.isel(time=[0].plot())* lassen sich die Daten des ersten Zeitschritts darstellen (Abb. A.2). Das zeitliche Mittel kann ähnlich der NumPy Funktionen mit *m = ds.temperature.mean('time')* berechnet werden. Um das konkrete Ergebnis aufzurufen muss die *compute*-Funktion aufgerufen werden. Dieses Ergebnis kann mit *.plot* dargestellt werden (Abb. A.3).

2.2. Dask DataFrame

Ein Dask DataFrame besteht aus vielen Pandas DataFrames (Abb. 2.4). Pandas ist eine Python Bibliothek zur Datenanalyse (siehe Ausarbeitung von Larissa Lach). Durch die Aufteilung in viele Pandas DataFrames ist parallele Rechnung möglich sowie Dask DataFrames, die größer als der Arbeitsspeicher sind [Dev18b]. Die Aufteilung eines Dask DataFrames in Pandas DataFrames ist nur entlang des Indexes möglich.

Mithilfe von Dask DataFrames lassen sich typische Pandas Funktionen ausführen. Als Beispiel werden dazu ERA5-Daten über dem Gebiet bei der Nordsee eingelesen (Listing B.1). In diesem Fall wird der Druck *sp*, die Windgeschwindigkeitskomponente von West nach Ost *u*, die Windgeschwindigkeitskomponente von Süd nach Nord *v*, die 2 m Temperatur *temp* und die Meeresoberflächentemperatur *sst* eingelesen. Dies geschieht mithilfe von *xarray* parallel *xr.open_mfdataset(data, parallel=True,...)*. Es entsteht ein *xarray.Dataset* bestehend aus einem *dask.array* pro Variable (Abb. B.1). Mithilfe der Funktion *to_dask_dataframe* wird das Dataset zu einem Dask DataFrame umgewandelt (Abb. B.2). Dieses besteht aus mehreren Partitions, die jeweils ein Pandas DataFrame sind. Es lassen sich einige üblich Pandas Funktionen anwenden. Mit der Funktion *head()* wird der Beginn der Daten angezeigt (Abb. B.3). Mit Pandas Funktionen können z.B. auch die Monatsmittel der Temperatur berechnet werden. Dazu werden die Daten zunächst

Importieren von numpy

```
import numpy as np
```

Erstellen eines Arrays

```
x = np.arange(10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Slicing

```
x[::2]
```

```
array([0, 2, 4, 6, 8])
```

Mittelwert bilden

```
x[::2].mean()
```

```
4.0
```

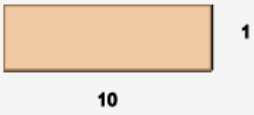
Importieren von dask.array

```
import dask.array as ds
```

Erstellen eines Arrays

```
x = ds.arange(10)
x
```

	Array	Chunk
Bytes	40 B	40 B
Shape	(10,)	(10,)
Count	1 Tasks	1 Chunks
Type	int32	numpy.ndarray



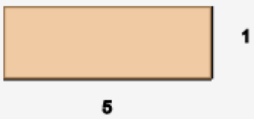

```
x.compute()
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Slicing ¶

```
x[::2]
```

	Array	Chunk
Bytes	20 B	20 B
Shape	(5,)	(5,)
Count	2 Tasks	1 Chunks
Type	int32	numpy.ndarray




```
x[::2].compute()
```

```
array([0, 2, 4, 6, 8])
```

Mittelwert bilden

```
x[::2].mean().compute()
```

```
4.0
```

Abbildung 2.2.: Vergleich der Syntax von NumPy und Dask Array

```
y = ds.arange(10,chunks=(5,))
y
```

	Array	Chunk
Bytes	40 B	20 B
Shape	(10,)	(5,)
Count	2 Tasks	2 Chunks
Type	int32	numpy.ndarray

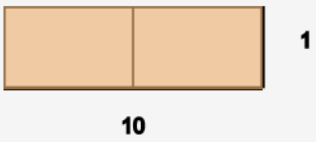


Abbildung 2.3.: Erstellen eines Dask Array mit über den Parameter *chunk* festgelegter Aufteilung des Arrays.

nach dem Monat gruppiert *groupby(by=[df.time.dt.month])*. Anschließend wird die Temperatur ausgewählt und das Mittel berechnet *df.groupby(by=[df.time.dt.month]).temp.mean()*. Damit wird in Dask jedoch nur ein Task Graph erstellt. Zur Berechnung muss die *compute*-Funktion aufgerufen werden. Mit *matplotlib.pyplot.plot()* kann das Ergebnis dargestellt werden (Abb. B.4). Durch *df.map_partitions(len)* wird die Länge der einzelnen Partitions angezeigt (Abb. B.5).

2.3. Dask Delayed

Mithilfe von Dask Delayed könne Funktionen parallelisiert werden [Dev18c]. Dabei wird vor dem eigentlichen Ausführen der Funktionen ein sogenanntes Delayed object erstellt (Abb. C.1). Dieses kann als Task Graph visualisiert werden und zeichnet die aufzurufenden Funktionen und die an sie zu übergebenden Argumente auf. Ein Delayed object wird mit *@delayed* über der Funktion oder beim Aufruf der Funktion mit *delayed(funktion)(argumente)* erstellt. Anschließend kann das Ergebnis mit *.compute()* berechnet werden. Die Parallelisierung ist nur dann möglich, wenn die Funktionen unabhängig voneineander sind, also das Ergebnis einer Funktion nicht von dem Ergebnis der anderen abhängt. Würde beispielsweise hinter der *add_del* Funktion eine weitere Funktion folgen, die vom Ergebnis von *add_del* abhängt, würde diese Funktion nicht parallel zu der *add_del* Funktion berechnet werden können (Abb. C.1).

Als Beispiel für die Parallelisierung mit Dask Delayed wird die Windgeschwindigkeit in 10 m Höhe aus den Windgeschwindigkeitskomponenten u und v berechnet. Das Ziel ist zudem die für die Berechnung benötigte Zeit zwischen der normalen Berechnung und der mit Dask Delayed zu vergleichen. Dazu werden zunächst die benötigten Bibliotheken importiert und die Daten eingelesen. Es werden die beiden Windgeschwindigkeitskomponenten u und v aus den ERA5-Daten über dem Gebiet bei der Nordsee verwendet.

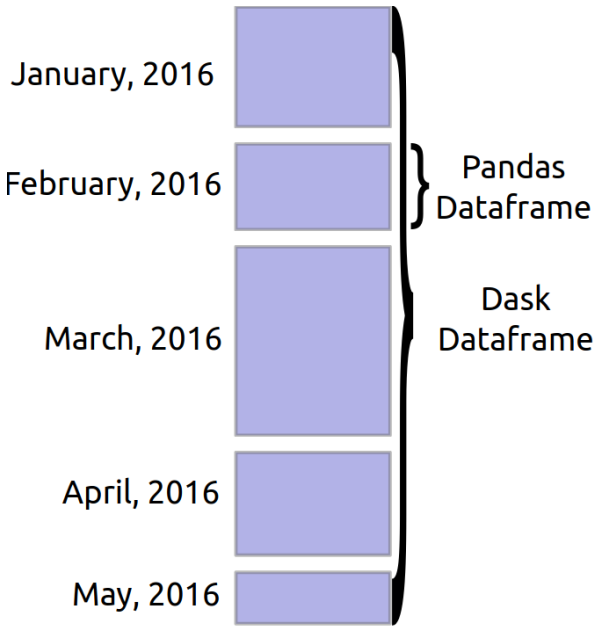


Abbildung 2.4.: Dask DataFrame bestehend aus mehreren Pandas DataFrames. Die Aufteilung in Pandas DataFrames geschieht entlang des Indexes, der hier die Zeit ist [AcRa18c].

Um die Datenmenge für dieses Beispiel geeignet zu wählen, wird der Zeitraum vom 01.01.2000 bis 31.12.2009 verwendet. Mit der Dask Funktion *load()* werden die Daten in den Arbeitsspeicher geladen. Die Windgeschwindigkeit aus den Windgeschwindigkeitskomponenten wird mit der Formel $|\vec{u}| = \sqrt{u^2 + v^2}$ berechnet. Dies wird in diesem Beispiel umgesetzt indem sie auf drei Funktionen aufgeteilt wird: das Wurzelziehen *def wurzel(x)*, das Addieren *def add(x,y)* und das Quadrieren *def quad(x)* (Listing C.1). Mit *%time speed = wurzel(add(quad(u),quad(v)))* wird bestimmt, wie lange die Berechnung dauert. Die Berechnungen der Wurzel von der u- und v-Komponente sind unabhängig voneinander und können daher parallelisiert werden. Dies geschieht mit Dask indem vor jeder der drei Funktionen *@delayed* geschrieben wird. Zudem muss zur Berechnung die *compute*-Funktion aufgerufen werden. Es ergibt sich, dass die parallele Berechnung mit Dask Delayed im Mittel etwas schneller ist als die ohne Dask Delayed (Tab. C.1). Um sich dies anzuschauen wird die Funktion *visualize* aufgerufen, die den Task Graphen aufzeigt (Abb. C.1). Es ist erkennbar, dass das Quadrieren der Funktionen parallel stattfindet. Durch das Aufteilen entsteht allerdings auch ein Overhead, der dazu führen kann, dass bei manchen Berechnungen die parallele Version langsamer als die normale Version ist.

3. Scheduler

Die parallele Ausführung des Task Graphen, der durch Collections erzeugt wurde, geschieht durch den Scheduler. Dafür gibt es verschiedene Scheduler [Dev18d]:

- **threaded**: Parallele Ausführung über Threads
- **processes**: Parallele Ausführung über Prozesse
- **single threaded**: Sequenzielle Ausführung und damit gut zum Debugging geeignet
- **distributed**: Parallele Ausführung auf einem Cluster. Dabei kann entweder ein verteiltes Cluster verwendet werden oder lokal ein Cluster auf dem Laptop erstellt werden.

Der Scheduler, z.B. der threaded-Scheduler, kann an verschiedenen Stellen ausgewählt werden [Dev18d]: Er kann direkt beim Aufruf der Funktion ausgewählt werden: wert.compute(scheduler="threaded"). Alternativ kann der Scheduler innerhalb eines Blocks mit dask.config.set(scheduler="threaded") festgelegt werden. Global wird er mit dask.config.set(scheduler="threaded") gesetzt.

Je nach Anwendung sind die einzelnen Scheduler unterschiedlich gut geeignet. Um zu verdeutlichen, dass die zur Berechnung benötigte Zeit je nach verwendeten Scheduler variiert, wird eine Zeitmessung mit verschiedenen Scheduler durchgeführt (Listing D.1). Dazu wird wie im Kapitel 2.1 das zeitliche Mittel der Temperatur berechnet (Listing A.1). Es werden Zeitmessungen mit den Scheduler *threaded*, *processes* und *single threaded* durchgeführt. Die Zeit wird mit *time.time()* jeweils vor und nach der Berechnung gemessen, um durch die Differenz beider Werte die für die Berechnung benötigte Zeit zu bestimmen. Es ergibt sich, dass im Mittel für diesen Fall der *threaded* Scheduler am besten geeignet ist (Tab. D.1). Die benötigte Zeit unterscheidet sich nicht wesentlich zwischen dem mit Prozessen parallelisiertem Code und dem sequenziellen Code. Dies liegt daran, dass es bei dem *processes* Scheduler zur Performance Einbußung durch Verschieben der Daten zu den Prozessen kommt [AcRa18d]. Im Gegensatz dazu ist der Overhead bei dem *threaded* Scheduler sehr gering. Allerdings ist eine Parallelisierung mit diesem Scheduler nur möglich, wenn die Berechnung durch nicht-Python Code (z.B. NumPy, Pandas) dominiert wird.

Der *distributed* Scheduler kann auch lokal auf dem eigenen Laptop verwendet werden (Listing D.1). Nach dem importieren der geeigneten Bibliothek *from dask.distributed*

import Client wird dazu ein Client-Objekt erstellt *client = Client*, das dann ein lokales Cluster aufsetzt *client.cluster()*. Daraufhin kann das Dashboard geöffnet werden (Abb. D.1). Dieses zeigt live an, wie viele Bytes pro Task gespeichert sind, sowie den Fortschritt der Tasks. Wird für diesen Scheduler die gleiche Zeitmessung durchgeführt, wie bei den anderen Scheduler, ergibt sich eine benötigte Zeit, die etwas geringer als für den Threaded Scheduler ist (Tab. D.1). Der Client wird mit *Client.close()* geschlossen.

4. Zusammenfassung

Übersteigt die Datenmenge den Arbeitsspeicher, kommt es in Python zu einem Memory Error. Um dennoch Python zur Datenanalyse verwenden zu können, kann Dask verwendet werden. Dask ist eine Bibliothek zur parallelen Programmierung in Python, die sowohl auf dem eigenen Laptop als auch auf einem Cluster funktioniert. Das Grundprinzip von Dask ist, dass Dask Collections Task Graphen erzeugen, die dann von Scheduler ausgeführt werden. D.h. zunächst wird ein Task Graph erstellt, der die für das Ergebnis aufzurufenden Funktionen aufzeigt. Erst durch den Aufruf der *compute*-Funktion wird das konkrete Ergebnis berechnet. Dask Collections sind Dask Arrays, Dask DataFrames, Dask Bags, Dask Delayed und Futures. Dabei gibt es eine große Überschneidung der API zwischen Dask Arrays und NumPy sowie zwischen Dask DataFrames und Pandas. Dask Delayed eignet sich zur Parallelisierung von Funktionen. Für die Ausführung existieren die folgenden Scheduler: threaded, processes, single-threaded sowie distributed. Dabei ist distributed sowohl auf einem verteilten Cluster als auch lokal auf dem Laptop möglich. Welcher Scheduler am besten geeignet ist, hängt vom konkreten Anwendungsfall ab.

Literaturverzeichnis

[AcRa18a] Inc. Anaconda and contributors Revision ab99d961, 2014-2018. https://docs.dask.org/en/latest/.

[AcRa18b] Inc. Anaconda and contributors Revision ab99d961, 2014-2018. https://docs.dask.org/en/latest/array.html.

[AcRa18c] Inc. Anaconda and contributors Revision ab99d961, 2014-2018. https://docs.dask.org/en/latest/dataframe.html.

[AcRa18d] Inc. Anaconda and contributors Revision ab99d961, 2014-2018. https://docs.dask.org/en/latest/scheduling.html.

[AcRc18] Inc. Anaconda and contributors Revision 66c1ae2d, 2014-2018. https://docs.dask.org/en/latest/dataframe-best-practices.html.

[Das19] Dask, 2019. https://www.youtube.com/watch?v=nnndxbr_Xq4.

[Dev18a] Dask Developers, 2018. https://tutorial.dask.org/03_array.html.

[Dev18b] Dask Developers, 2018. https://tutorial.dask.org/04_dataframe.html.

[Dev18c] Dask Developers, 2018. https://tutorial.dask.org/01_dask.delayed.html.

[Dev18d] Dask Developers, 2018. https://tutorial.dask.org/05_distributed.html.

[fMRWF] European Centre for Medium-Range Weather Forecasts. https://www.ecmwf.int/en/forecasts/datasets/reanalysis-datasets/era5.

A. Dask Array

```
1  import matplotlib.pyplot as plt
2  import xarray as xr
3
4  data = './era5-t2-1979-2019_ns.nc'
5  ds = xr.open_dataset(data, chunks={'lat':-1, 'lon':-1,
       ↪ 'time':50000})
6  ds = ds.rename_vars({'var167':'temperature'})
7
8  ds.temperature.isel(time=[0].plot())
9  m = ds.temperature.mean('time')
10 m.compute()
11 m.plot()
```

Listing A.1: Beispielprogramm zum Dask Array

xarray.Dataset

▶ Dimensions: (**lat**: 34, **lon**: 61, **time**: 359400)

▼ Coordinates:

time	(time)	datetime64[ns]	1979-01-01 ... 2019-12-31T23:00:00
lon	(lon)	float64	-3.938 -3.656 ... 12.66 12.94
standard_name :	longitude		
long_name :	longitude		
units :	degrees_east		
axis :	X		
lat	(lat)	float64	60.28 60.0 59.72 ... 51.29 51.01

▼ Data variables:

temperature	(time, lat, lon)	float32	dask.array<chunksize=(50000, 34, 61), meta=…

	Array	Chunk
Bytes	2.98 GB	414.80 MB
Shape	(359400, 34, 61)	(50000, 34, 61)
Count	9 Tasks	8 Chunks
Type	float32	numpy.ndarray

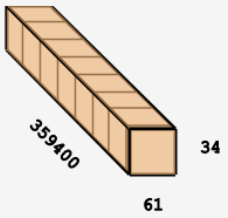


Abbildung A.1.: Ergebnis der Ausführung von Listing A.1 Zeile 1-6. Die ERA5-Daten der Temperatur sind als xarray Dataset eingelesen. Die Temperatur ist als Dask Array eingelesen mit einer Aufteilung der Daten entlang der Zeitachse.

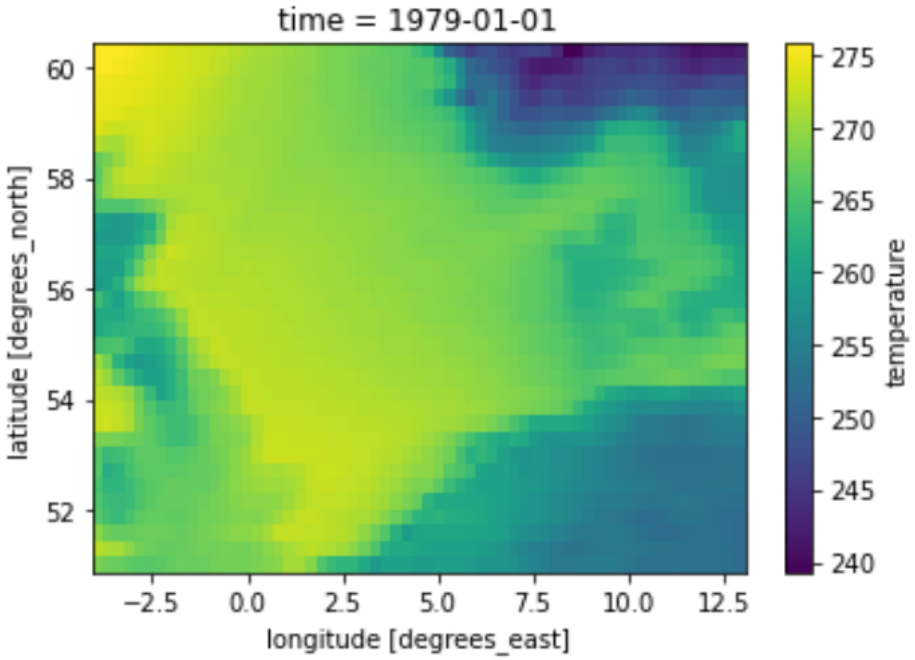


Abbildung A.2.: Ergebnis der Ausführung von Listing A.1 Zeile 1-8. Dargestellt ist die 2 m Temperatur eines Gebietes bei der Nordsee am 01.01.1979.

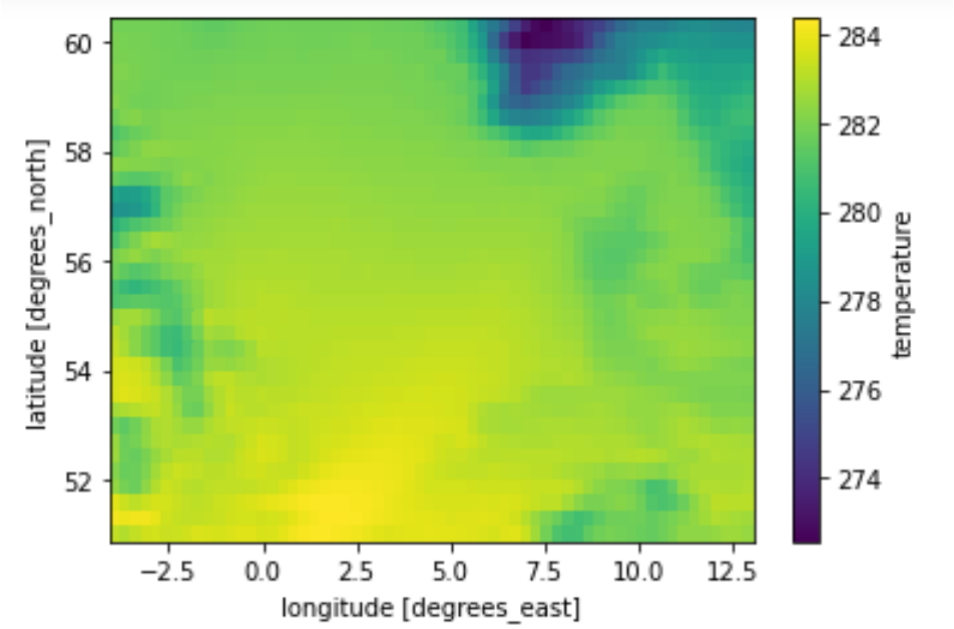


Abbildung A.3.: Ergebnis der Ausführung von Listing A.1 mit Ausnahme von Zeile 8. Die Darstellung zeigt das zeitliche Mittel der 2 m Temperatur von 1979 bis 2019 über ein Gebiet bei der Nordsee.

B. Dask DataFrame

```
1  import matplotlib.pyplot as plt
2  import xarray as xr
3  import dask
4
5  data = './era5/*_ns.nc'
6  ds = xr.open_mfdataset(data, parallel=True,
     ↪ chunks={'lat':-1, 'lon':-1, 'time': 1000})
7  ds = ds.rename_vars({'var134':'sp', 'var165':'u',
     ↪ 'var166':'v', 'var167':'temp', 'var34':'sst'})
8
9  df = ds.to_dask_dataframe()
10
11 df.head()
12
13 monmean = df.groupby(by=[df.time.dt.month]).temp.mean()
14 plt.plot(monmean.compute())
15
16 df.map_partitions(len).compute()
```

Listing B.1: Beispielprogramm zu Dask DataFrames

xarray.Dataset

▸ Dimensions: (**lat**: 34, **lon**: 61, **time**: 8784)

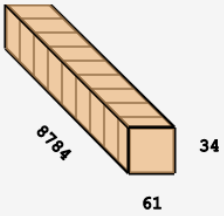
▾ Coordinates:

time	(time)	datetime64[ns]	2000-01-01 ... 2000-12-31T23:00:00
lon	(lon)	float64	-3.938 -3.656 ... 12.66 12.94
lat	(lat)	float64	60.28 60.0 59.72 ... 51.29 51.01

▾ Data variables:

sp	(time, lat, lon)	float32	dask.array<chunksize=(1000, 34, 61), meta=n…

	Array	Chunk
Bytes	72.87 MB	8.30 MB
Shape	(8784, 34, 61)	(1000, 34, 61)
Count	10 Tasks	9 Chunks
Type	float32	numpy.ndarray



u	(time, lat, lon)	float32	dask.array<chunksize=(1000, 34, 61), meta=n…
v	(time, lat, lon)	float32	dask.array<chunksize=(1000, 34, 61), meta=n…
temp	(time, lat, lon)	float32	dask.array<chunksize=(1000, 34, 61), meta=n…
sst	(time, lat, lon)	float32	dask.array<chunksize=(1000, 34, 61), meta=n…

Abbildung B.1.: Ergebnis der Ausführung von Listing B.1 Zeile 1-7. Die ERA5-Daten des Drucks *sp*, der u- und v-Komponente des Windes *u, v*, der Temperatur *temp* und der Meeresoberflächentemperatur *sst* sind als xarray Dataset eingelesen. Jede der Variablen entspricht einem Dask Array, das entlang der Zeitachse aufgeteilt ist.

Dask DataFrame Structure:

	lat	lon	time	sp	u	v	temp	sst
npartitions=10								
0	float64	float64	datetime64[ns]	float32	float32	float32	float32	float32
1607472	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
14467248	...	...	...	...	...	...	...	...
18218015	...	...	...	...	...	...	...	...

Dask Name: concat-indexed, 662 tasks

Abbildung B.2.: Ergebnis der Ausführung von Listing B.1 Zeile 1-9. Das xarray Dataframe ist zu einem Dask DataFrame umgewandelt. Die geographische Breite *lat*, die geographische Länge *lon* und die Zeit *time* sind Spaltenvariablen. Das Dask DataFrame ist in 10 Partitions aufgeteilt, die jeweils ein Pandas DataFrame entsprechen.

	lat	lon	time	sp	u	v	temp	sst
0	60.281	-3.9375	2000-01-01 00:00:00	100459.687500	12.614838	0.871338	280.548950	282.311859
1	60.281	-3.9375	2000-01-01 01:00:00	100558.890625	13.538696	0.080917	280.860657	282.311859
2	60.281	-3.9375	2000-01-01 02:00:00	100648.265625	11.738480	-0.500160	281.017456	282.311859
3	60.281	-3.9375	2000-01-01 03:00:00	100753.617188	10.899223	0.816841	281.060913	282.311859
4	60.281	-3.9375	2000-01-01 04:00:00	100808.875000	10.576469	3.201736	281.148376	282.311859

Abbildung B.3.: Ergebnis der Ausführung von Listing B.1 Zeile 1-11. Die Funktion head() gibt den Inhalt der ersten Zeilen aus.

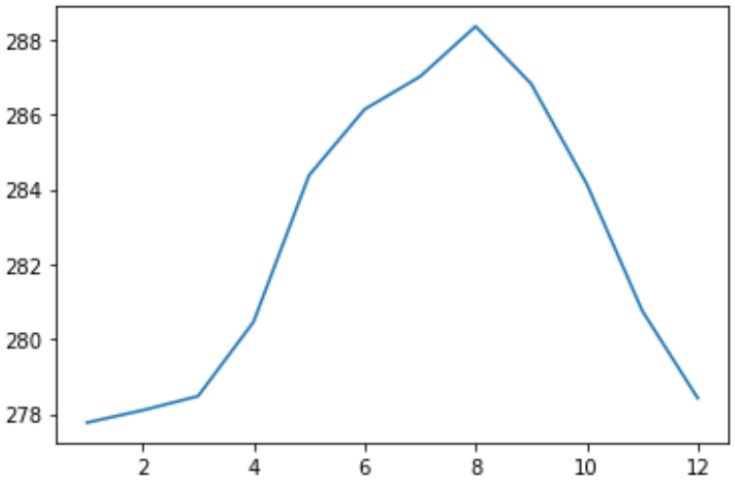


Abbildung B.4.: Ergebnis der Ausführung von Listing B.1 Zeile 1-9 und 13-14. Dargestellt sind die Monatsmittel der Temperatur des Jahres 2000 über ein Gebiet bei der Nordsee. Auf der x-Achse sind die Monate und auf der y-Achse die Temperatur in Kelvin.

```
Out[26]: 0    1607472
         1    1607472
         2    1607472
         3    1607472
         4    1607472
         5    1607472
         6    1607472
         7    1607472
         8    1607472
         9    3750768
         dtype: int64
```

Abbildung B.5.: Ergebnis der Ausführung von Listing B.1 Zeile 1-9 und 16. Dargestellt ist die Länge der einzelnen Partitions.

C. Dask Delayed

```
import numpy as np
import xarray as xr
from dask import delayed

datau = './era5-u10-1979-2019_ns.nc'
datav = './era5-v10-1979-2019_ns.nc'
du = xr.open_dataset(datau)
du = du.rename_vars({'var165':'u'})
dv = xr.open_dataset(datav)
dv = dv.rename_vars({'var166':'v'})
usel = du.u.sel(time=slice('2000-01-01','2009-12-31'))
vsel = dv.v.sel(time=slice('2000-01-01','2009-12-31'))
usel.load()
vsel.load()

def wurzel(x):
    w = np.sqrt(x)
    return w
def add(x,y):
    return x + y
def quad(x):
    q = x * x
    return q
%time speed = wurzel(add(quad(usel),quad(vsel)))

@delayed
def wurzel_del(x):
    w = np.sqrt(x)
    return w
@delayed
def add_del(x,y):
    return x + y
@delayed
def quad_del(x):
    q = x * x
    return q
```

```
37 %time speed =
      ↪ wurzel_del(add_del(quad_del(usel),quad_del(vsel)))
38 %time speed.compute()
39 speed.visualize()
```

Listing C.1: Beispielprogramm zu Dask Delayed

Tabelle C.1.: Zeitmessungen für die Berechnung der Windgeschwindigkeit nach Listing C.1. Es wurden fünf Zeitmessungen für die Berechnung ohne Dask (Listing C.1 Zeile 24) und mit Dask Delayed (Listing C.1 Zeile 38) durchgeführt.

	Messung 1	Messung 2	Messung 3	Messung 4	Messung 5	Mittelwert
Normal [s]	3,48	1,72	1,63	1,72	1,59	2,03
Delayed [s]	2,23	1,94	1,42	1,46	1,43	1,70

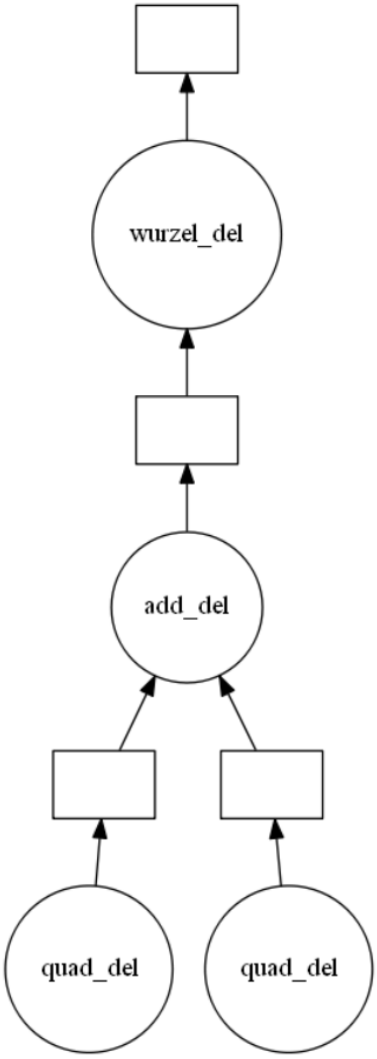


Abbildung C.1.: Task Graph, der durch Ausführung von Listing C.1 Zeile 39 entsteht. Die Kreise entsprechen Python Funktionen und die Rechtecke ein Python Objekt, das Output eines Tasks ist und gleichzeitig Input für einen anderen Task.

D. Scheduler

```
1  import xarray as xr
2  import time
3
4  data = './era5-t2t1979-2019_ns.nc'
5  ds = xr.open_dataset(data, chunks={'lat':-1, 'lon':-1,
       ↪ 'time':50000})
6  ds.rename_vars({'var167':'temperature'})
7
8  m = ds.temperature.mean('time')
9
10 for sch in ['threading', 'processes', 'sync']:
11     t0 = time.time()
12     calc_mean = m.compute(scheduler=sch)
13     t1 = time.time()
14     print(f"{sch:>10}, {t1 - t0:0.4f}")
15
16 from dask.distributed import Client
17 client = Client()
18 client.cluster
19 m.compute()
20 client.close()
```

Listing D.1: Beispielprogramm zu Scheduler. Verändert nach [Dev18d]

Tabelle D.1.: Zeitmessungen für die Berechnung des zeitlichen Mittels der Temperatur nach Listing D.1 mit den verschiedenen Scheduler.

	Messung 1	Messung 2	Messung 3	Messung 4	Messung 5	Mittelwert
Threads	10,9237	6,5242	6,3175	6,2028	6,5375	7,3011
Prozesse	19,0876	10,2072	9,8749	9,9622	9,6988	11,7661
Sequenziell	17,1061	9,7824	9,9271	9,7938	9,4695	11,2158
Distributed	5,6773	5,2635	5,0469	5,4713	7,0101	5,6938

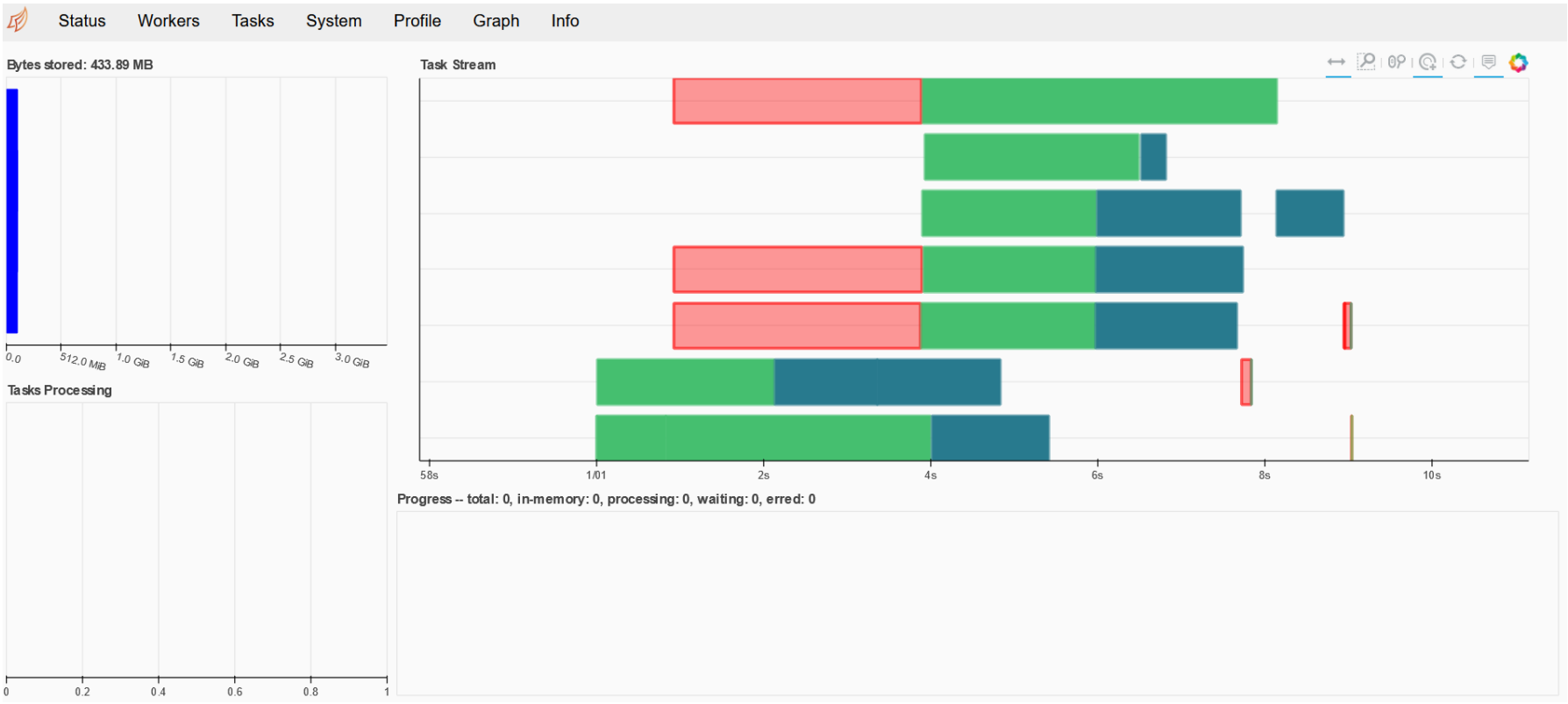


Abbildung D.1.: Dashboard, das nach Ausführung von Listing D.1 Zeile 16-18 entsteht. Es zeigt an, wie viele Bytes aktuell gespeichert sind (oben links), den aktuellen Fortschritt der Tasks (unten links) und den Task Stream (rechts). Zu sehen ist der Zustand des Dashbordes nach der Bildung des Mittelwerts (Listing D.1 Zeile 19). Im Task Stream steht die rote Markierung für Datentransfer, die grüne Markierung für das Öffnen der Daten und die blaue Markierung für die eigentliche Berechnung.