# Analyse mehrdimensionaler Arrays auf Hochleistungsrechnern

Aaron Spring

2019-07-01

## 1 Einführung

In den letzten Jahren ist es fuer Wissenschaftler und Unternehmen einfacher geworden ihre Daten mit Python-basierten Programmen zu analysieren. Zwei maßgebliche Faktoren dafuer sind die interactiven notebooks mit ipython [Shen, 2014] und die Breite der freiverfügbaren, kombinierbaren und kostenlosen Analyse-Tools [Abb. 1], die dem Anwender erlauben, den Fokus von der technischen Analyse zum Ausarbeiten der Geschichte hinter den Daten zu verschieben erlauben.

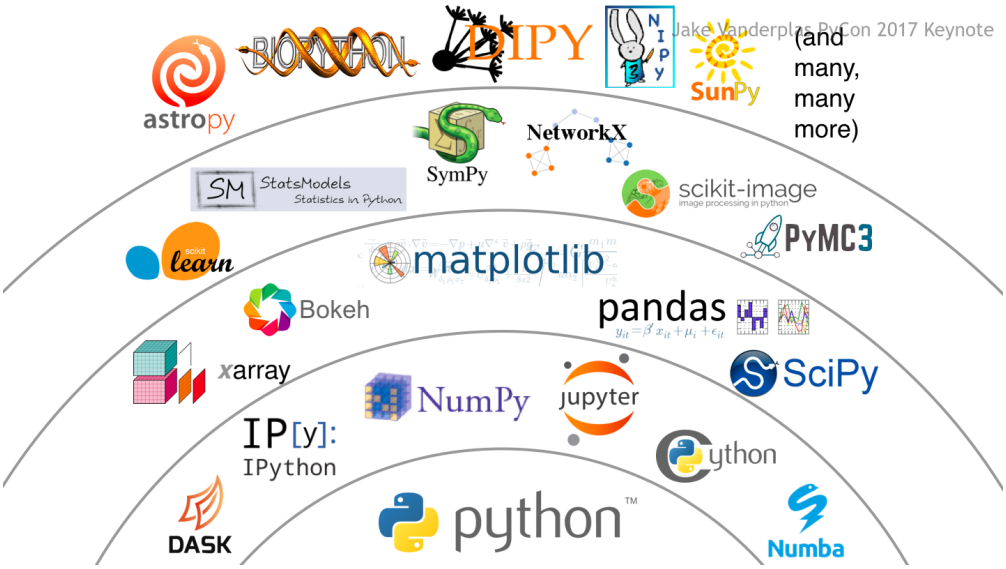



Abbildung 1: Ökosystem des wissenschaftlichen Rechnens mit Python [*Python Visualization Landscape*]

Das Ökosystem des wissenschaftlichen Rechnens mit Python nach Jake VanderPlas unterteilt die Vielzahl von Pythonprogrammen anhand ihres Abstraktionsniveaus, also wie direkt der Benutzer mit den mathematischen Methoden (wie `numpy` [Walt u. a., 2011]) arbeitet oder diese durch eine Bibliothek wie sk-learn in den Abhängigkeiten nutzt [Abb. 1].

Viele der Programme aus Abb. 1 wurden bereits im Proseminar beschrieben. Der heutige Vortrag über `xarray` und `dask` stellt das Zugpferd für die Analyse mehrdimensionaler Daten vor und versucht gleichzeitig die Verbindungen dieser Programme zu den Anderen aufzuzeigen.

Mehrdimensionale Daten sind auf n Dimensionen beschrieben. Eine davon ist meistens die Zeit und andere können wie bei Klimadaten Longitude, Latitude oder Höhe/Tiefe sein. Dazu kommt dann noch ein dazugehöriger Wert wie es bei Börsendaten der Preis ist oder bei Klimadaten die Temperatur. Bei der Analyse mehrdimensionaler Daten werden meistens alle Werte einer Dimension einer Rechenoperation unterzogen, weswegen im Hintergrund mit numpy gerechnet wird. Allerdings versteht numpy nur die Indizes der Achsen und erkennt nicht automatisch die Namen der Dimensionen. Aus diesem Grund wurde `xarray` ins Leben gerufen.

In dieser Ausarbeitung werde ich zuerst aufzeigen wie die Datenanalyse von Klimamodelloutput vor dem Zeitalter von `xarray` funktionierte. Danach werde ich `xarray` demonstrieren. Bei sehr großen (mehrere 100GB) Datensätzen wird `xarray` allerdings beim Einlesen und Berechnen sehr langsam, was den verteilten Arbeitsanweisungsplaner `dask` auf den Plan ruft, welcher in `xarray` integriert ist. Abschließend werde ich dann noch die Erweiterungsmöglichkeiten und Datentyptransformationen zwischen den verschiedenen Python Programmen aufzeigen. Das Jupyter Notebook zu diesem Vortrag ist verfügbar unter https://nbviewer.jupyter.org/github/aaronspring/xarray_dask_talk_unihh/blob/master/notebook/xarray_dask.ipynb.

## 2 Klimadaten einlesen

Klimadaten von Erdsystemmodellen, Wettervorhersagemodellen, Satelliten oder einfache Messreihen von Beobachtungsstationen sind häufig im Format `netCDF` gespeichert. Dies ist ein Dateiformat zum Austausch wissenschaftlicher Daten und ist dem Hierarchichal Data Format (HDF, siehe HDF-Vortrag) ähnlich. NetCDF Daten sind selbstbeschreibend, da sie im Header die Metadaten und Form der in der Datei vorliegenden Daten gespeichert haben.

Vor es `xarray` gab (bis 2015), mussten Wissenchaftler `netCDF` auf die harte Tour über das Package netcdf4 einlesen. Dafür wurde ein mit `Dataset` eine Datei geöffnet. Auf Metadaten ist zugreifbar über die API und die Werte sind mit Dataset.[variable][1,:,:] in numpy Zugriffs-API verfügbar. Zweidimensionale Daten können mit `matplotlib` [Hunter, 2007] dargestellt werden.

```
In [3]: # auf mistral
        urlpath = '/pool/data/ICDC/ocean/hadisst1/DATA/HadISST_sst.nc'

In [5]: from netCDF4 import Dataset
        import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy as np

In [6]: ds = Dataset(urlpath)

In [7]: ds

Out[7]: <class 'netCDF4._netCDF4.Dataset'>
        root group (NETCDF3_CLASSIC data model, file format NETCDF3):
            Title: Monthly version of HadISST sea surface temperature component
            description: HadISST 1.1 monthly average sea surface temperature
            institution: Met Office Hadley Centre
            source: HadISST
            reference: Rayner, N. A., Parker, D. E., Horton, E. B., Folland, C. K., Alexander, L
            Conventions: CF-1.0
```

```
    history: Mon Apr 29 12:23:08 2019: ncatted -a _FillValue,sst,m,f,-1000 HadISST_sst_o
12/3/2019 converted to netcdf from pp format
    supplementary_information: Updates and supplementary information will be available f
    comment: Data restrictions: for academic research use only. Data are Crown copyright
    dimensions(sizes): time(1789), latitude(180), longitude(360), nv(2)
    variables(dimensions): float32 [time[(time), float32 [time_bnds[(time,nv), float32 [
    groups:
```

```
In [8]: ds.variables.keys()
```

```
Out[8]: odict_keys(['time', 'time_bnds', 'latitude', 'longitude', 'sst'])
```

```
In [9]: sst = ds.variables['sst']
        sst
```

```
Out[9]: <class 'netCDF4._netCDF4.Variable'>
        float32 sst(time, latitude, longitude)
            _FillValue: -1000.0
            standard_name: sea_surface_temperature
            long_name: sst
            units: C
            cell_methods: time: lat: lon: mean
            missing_value: -1e+30
        unlimited dimensions: time
        current shape = (1789, 180, 360)
        filling on
```

```
In [10]: sst.size/1e6
```

```
In [11]: sst.shape
```

```
Out[11]: (1789, 180, 360)
```

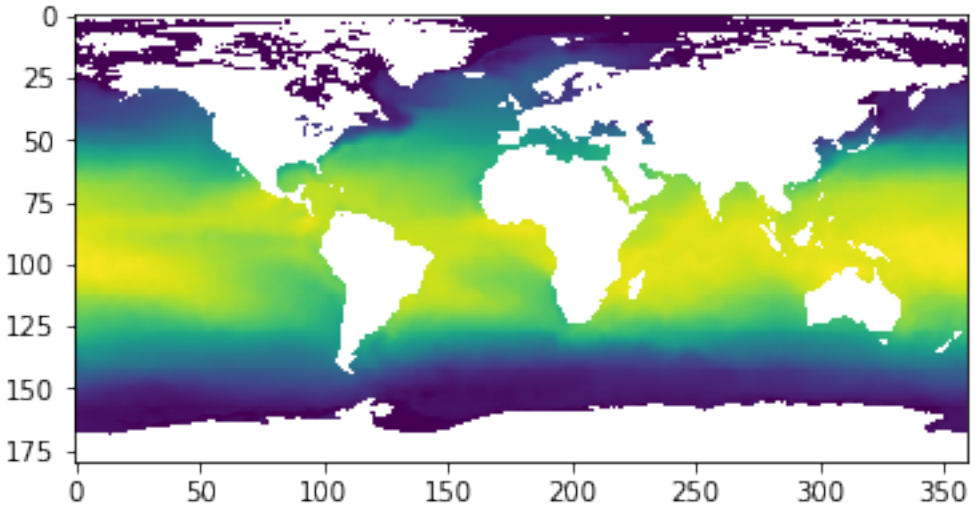
```
In [12]: time = ds.variables['time']
         time
```

```
Out[12]: <class 'netCDF4._netCDF4.Variable'>
         float32 time(time)
             units: days since 1870-1-1 0:0:0
             calendar: gregorian
             long_name: Time
             standard_name: time
         unlimited dimensions: time
         current shape = (1789,)
         filling on, default _FillValue of 9.969209968386869e+36 used
```

```
In [13]: last_timestep = sst[-1,:,:]
```

```
In [12]: type(last_timestep)
```

```
Out[12]: numpy.ma.core.MaskedArray

In [14]: last_timestep.shape

Out[14]: (180, 360)

In [15]: plt.imshow(last_timestep)

Out[15]: <matplotlib.image.AxesImage at 0x2b7b52f4c9e8>
```




Die Datenanalyse von netcdf Datei war also möglich, allerdings mussten die Werte immer erst aus dem Dataset ausgepackt werden um dann mit numpy Rechenoperationen auszufüren. Ein direkter und intuitiver Zugang war das nicht. Außerdem ist die Information auf der Zeitachse dekodiert (hier als Tage seit einem Referenzdatum angegeben). Für die zeitliche Analyse ist numpy hier nicht intuitiv ausgelegt. Weiterhin ist die Darstellung frei von allen Metainformationen der Datei.

# 3 xarray

Unter diesen Umständen entwickelten Hoyer und Hamman, 2017 mit dem Whitepaper "Xarray: N-D Labeled Arrays and Datasets in Pythonëine durch pandas [McKinney, 2010] inspierten Alternative zu Python Package netcdf4 für die Analyse von mehrdimensionalen Daten, die auf das Dateiformat netcdf und die Nutzung durch Klimawißenschaftler zugeschnitten ist. Das Einlesen, Bearbeiten und Darstellen ist so viel einfacher.

```
In [16]: import xarray as xr

In [17]: ds = xr.open_dataset(urlpath)

In [18]: ds

Out[18]: <xarray.Dataset>
         Dimensions:    (latitude: 180, longitude: 360, nv: 2, time: 1789)
         Coordinates:
           * time       (time) datetime64[ns] 1870-01-16T11:59:59.505615234 ... 2019-01-16T12:00
           * latitude   (latitude) float32 89.5 88.5 87.5 86.5 ... -87.5 -88.5 -89.5
           * longitude  (longitude) float32 -179.5 -178.5 -177.5 ... 177.5 178.5 179.5
         Dimensions without coordinates: nv
         Data variables:
             time_bnds  (time, nv) float32 ...
             sst        (time, latitude, longitude) float32 ...
         Attributes:
             Title:                      Monthly version of HadIsst sea surface temper...
             description:                HadIsst 1.1 monthly average sea surface tempe...
             institution:                Met Office Hadley Centre
             source:                     HadIsst
             reference:                  Rayner, N. A., Parker, D. E., Horton, E. B., ...
             Conventions:                CF-1.0
             history:                    Mon Apr 29 12:23:08 2019: ncatted -a _FillVal...
             supplementary_information:  Updates and supplementary information will be...
             comment:                    Data restrictions: for academic research use ...
```

## 3.1 Datenmodell

xarray.Datasets sind Container für mehrere xarray.DataArrays [Fig. 2]. Die meisten xarray Rechenoperationen können entweder auch ein xarray.DataArray oder auf alle Variablen in einem xarray.Dataset gleichzeitig angewandt werden. Ein xarray.DataArray besitzt seine selbstbeschreibenden Metadaten und die Werte sind als numpy.ndarray hinterlegt, woher die Performance for xarray Operationen stammt. xarray.DataArrays teilen sich jeweils die Koordinaten und haben ihre eigenen Namen und Attribute, die als Dictionary die Metadaten speichern.

```
In [19]: ds['sst']

Out[19]: <xarray.DataArray 'sst' (time: 1789, latitude: 180, longitude: 360)>
         [115927200 values with dtype=float32]
         Coordinates:
```

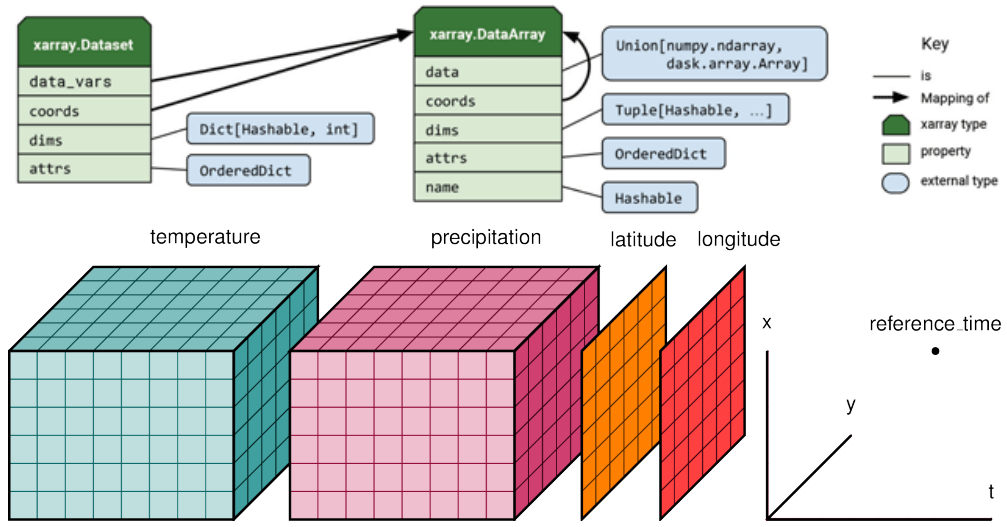



Abbildung 2: `xarray` Datenmodel [*Xarray Documentation*]

```
  * time       (time) datetime64[ns] 1870-01-16T11:59:59.505615234 ... 2019-01-16T12:00
  * latitude   (latitude) float32 89.5 88.5 87.5 86.5 ... -87.5 -88.5 -89.5
  * longitude  (longitude) float32 -179.5 -178.5 -177.5 ... 177.5 178.5 179.5
Attributes:
    standard_name:  sea_surface_temperature
    long_name:      sst
    units:          C
    cell_methods:   time: lat: lon: mean
```

```
In [20]: type(ds['sst'].values)
```

```
Out[20]: numpy.ndarray
```

```
In [21]: # Dimensionen benennen die Axen
         ds['sst'].dims
```

```
In [22]: # Koordinaten beschreiben die Werte auf dem Grid
         ds['sst'].coords
```

```
Out[22]: Coordinates:
           * time       (time) datetime64[ns] 1870-01-16T11:59:59.505615234 ... 2019-01-16T12:00
           * latitude   (latitude) float32 89.5 88.5 87.5 86.5 ... -87.5 -88.5 -89.5
           * longitude  (longitude) float32 -179.5 -178.5 -177.5 ... 177.5 178.5 179.5
```

```
In [23]: ds['sst'].attrs
```

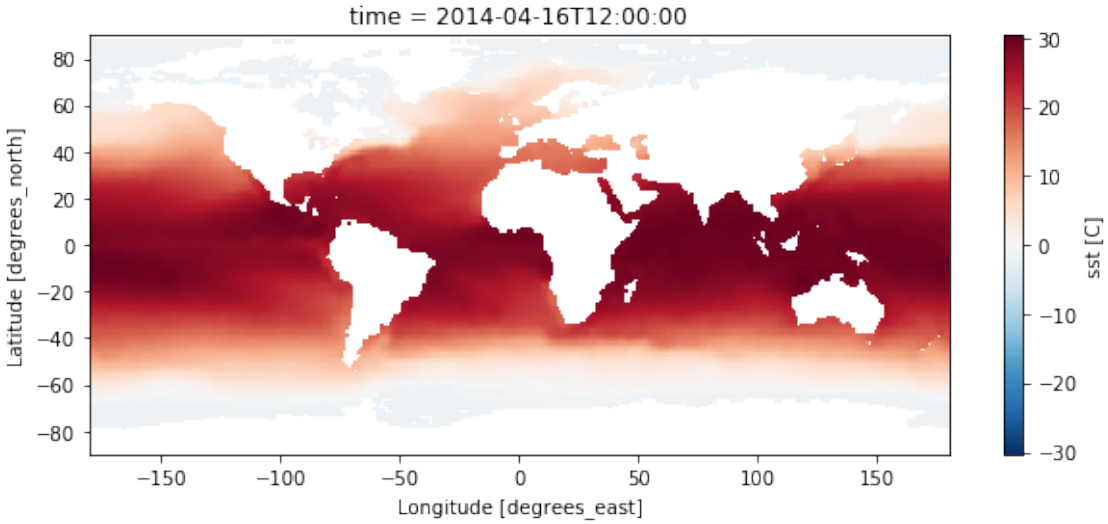
```
Out[23]: OrderedDict([('standard_name', 'sea_surface_temperature'),
                      ('long_name', 'sst'),
                      ('units', 'C'),
                      ('cell_methods', 'time: lat: lon: mean')])
```

```
In [24]: ds['sst'].attrs['units']

Out[24]: 'C'
```

Der große Game-changer ist die intuitive API, die an pandas angelehnt ist [http://xarray.pydata.org/en/stable/api.html]. Es können nicht nur einzelne Zeitschritte nach Index ausgewählt werden, sondern auch nach ihrer Bezeichnung. Mit der selben Syntax wie in pandas können Jahre oder Monate mit String Argumenten ausgewählt werden.

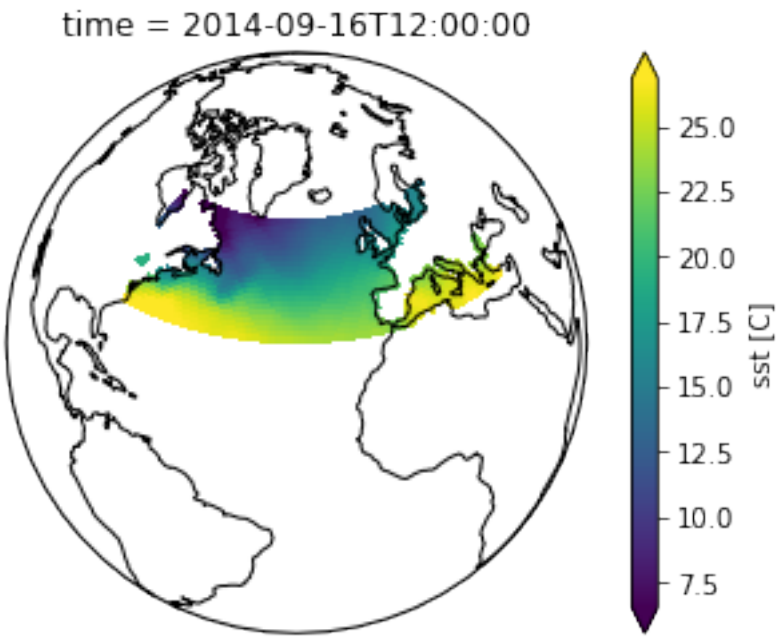
Außerdem ist `xarray`.plot() ein Wrapper von matplotlib und versteht einige deßen Argumente.

```
In [26]: ds.sel(time='2014-04')['sst'].plot(figsize=(10,4))

Out[26]: <matplotlib.collections.QuadMesh at 0x2ab4d3c619b0>
```




Für ein korrektere geographische Darstellung und verschiedene Projektionen der Daten kann `xarray` auch `cartopy` [Met Office, 2010–2015] einbinden.

```
In [26]: import cartopy.crs as ccrs
         ax = plt.axes(projection=ccrs.Orthographic(-30, 35))
         ds.sel(time='2014-09',longitude=slice(-80,30),
                               latitude=slice(60,35))['sst'].plot(ax=ax,
                                                                  transform=ccrs.PlateCarree(),
                                                                  robust=True)
         ax.set_global()
         ax.coastlines()

Out[26]: <cartopy.mpl.feature_artist.FeatureArtist at 0x2b7b5d621940>
```

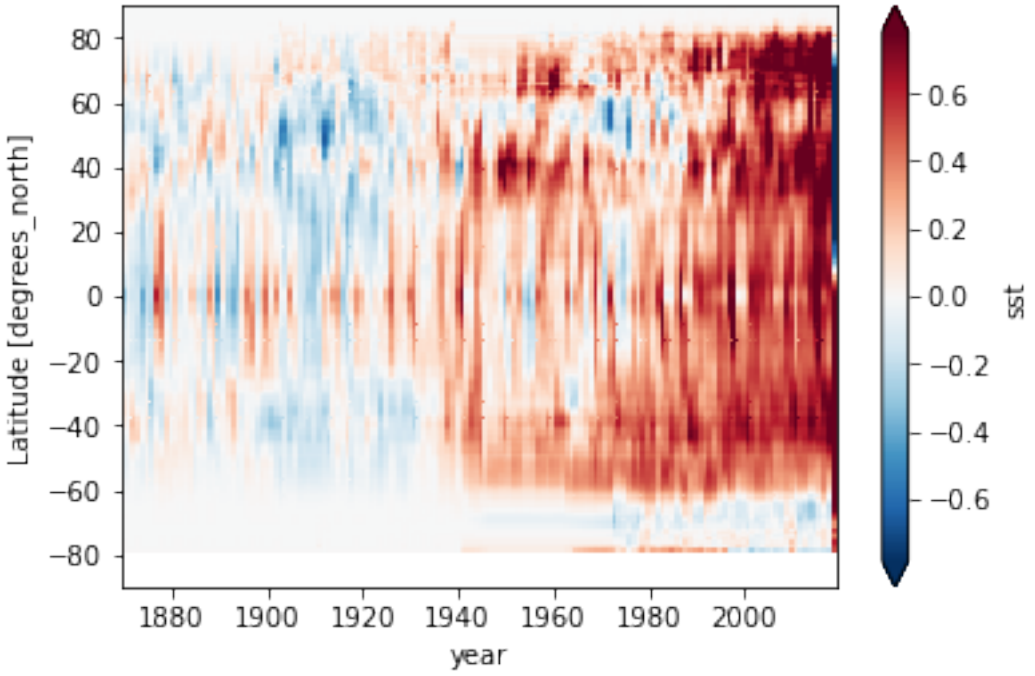



### 3.2 Beispiel: Ozeanerwärmung pro Breitengrad über die Zeit

Zum schnellen Erstellen einer Grafik lassen sich auch mehrere Operationen hintereinander ausführen. Beispielhaft wird hier die mittlere Ozeanerwärmung pro Breitenjahr pro Jahr zu einem Referenzzeitraum geplottet.

```
In [31]: (ds['sst']-ds['sst'].sel(
         time=slice('1850','1900')).mean('time')).groupby(
                     'time.year').mean('time').mean('longitude').T.plot(robust=True)
```

```
Out[31]: <matplotlib.collections.QuadMesh at 0x2ab503ac5a20>
```




Schritt-für-Schritt bewirken die einzelnen Operationen:

1. `robust=True` schränkt die Colorbar auf das 2- und 98-Prozentil ein.
2. `T` transponiert die Matrix und vertauscht so die Axen
3. `mean('longitude')` mittelt über die Breitengrade
4. `groubpy('time.year').mean('time')` erstellt Jahresmittelwerte
5. `sel(time=slice('1850','1990'))` wählt alle Jahre zwischen den beiden angegebenen Jahren aus, wodurch eine Anomalie zu diesen Jahren dargestellt werden kann

### 3.3 Limitierung

So intuitiv und einfach die Datenbearbeitung mit `xarray` ist, ist ein Nachteil, dass mit `xr.open_dataset()` der gesamte Datensatz inkl. Werte geladen wird. Bei 400 MB ist das verkraftbar. Wenn allerdings mehrere Dateien gleichzeitig geladen werden, wird Input/Output zum Flaschenhals. Besonders deutliche wird es beim Bearbeiten von Satellitendaten. Der MODIS-SST Satellitenprodukt beherbergt pro Jahr 2.7 GB in 12 Dateien. Für den Zeitraum von 2002 bis 2019 sind dasinsgesamt 120 GB. Das ist zu groß für jeden Laptop und auch für das Dateisystem eines Supercomputes ein sehr große Herausforderung.

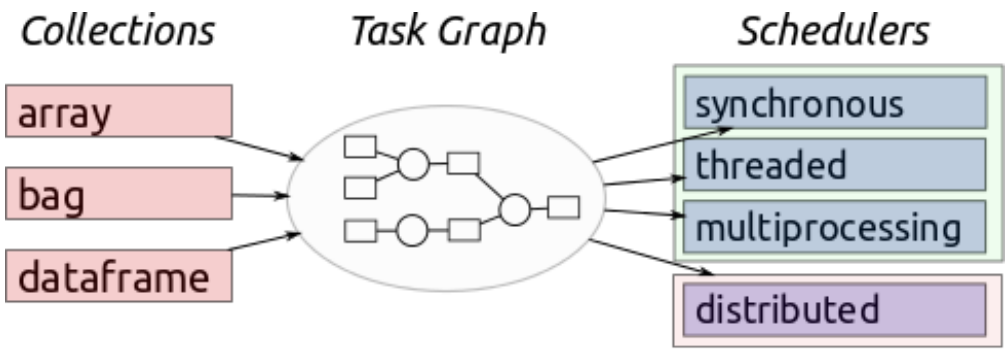



Abbildung 3: `dask` Datenmodell [*Dask Documentation*]

# 4 `dask` package

Um Big Data mit Python zu analysieren haben Rocklin, 2015 mit dem White-Paper "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling"`dask` entwickelt. Dies ist ein dynamischer Task Scheduler, der große Datensätz in kleinere aufteilt (chunkt) und diese Chunks dann auf `multiprocessing`, `threading` oder `concurrent` an verschiedene Worker aufteilt und somit den Arbeitsaufwand parallelisiert. Hier wird wiederum viel auf die API von pandas gesetzt.`dask` skaliert vom Laptop zum Supercomputer. Laptops können durch Chunking Datensätze bearbeiten, welche viel größer als der Arbeitsspeicher (out-of-memory) sind. Supercomputer oder generell Systeme mit mehreren Prozessoren können die Rechenoperationen parallelisieren.

## 4.1 Datenmodel

Das`dask` Datenmodell basiert auf bisherigen Projekten und erweiterte diese um Chunking mit verschiedenen Ansätzen.

`dask.array` ist dem `numpy.ndarray` angelehnt und wird für `xarray` verwendet. `dask.bag` kann verschiedene Operationen, die auf `iterable`, also zum Beispiel Listen, basieren, parallelisieren. Das `dask.dataframe` ist das`dask` äquivalent zum `pandas.DataFrame` [Fig. 3]. Die so gechunkten Tasks verwerden an die verschiedenen Parallelisierungsframeworks gesendet, wovon der Benutzer nichts direkt mitbekommt.

`xarray` kann den Taskgraphen leider nicht visualisieren, daher hier das minimale Beispiel mit Zufallszahlen. Allerdings kann `dask.array` keine Gruppierungen.

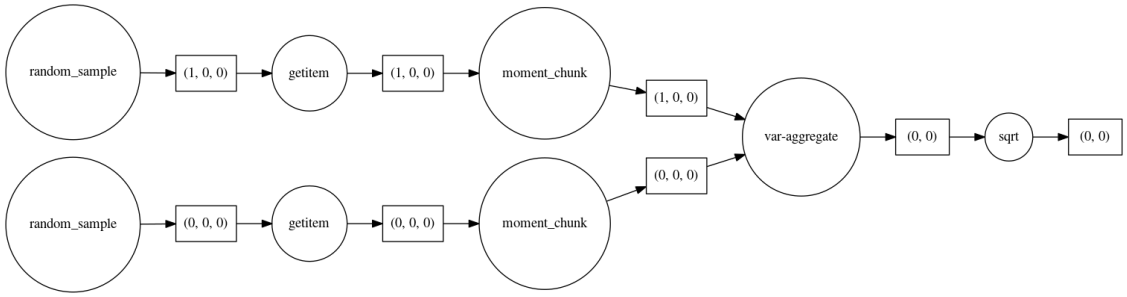
```
In [27]: import dask.array

In [35]: x = dask.array.random.random((500, 500, 500), chunks=(250, 500, 500))
         x

Out[35]: dask.array<random_sample, shape=(500, 500, 500), dtype=float64, chunksize=(250, 500, 50
```

## 4.2 Visualization des Taskgraphen

Als Beispiel einer Baumreduktion wird der Taskgraph der Standardabweichung aufgezeigt. In`dask` wird die Berechnung erst am Ende ausgeführt mit dem Befehl `compute()`. Vorher wird nur der Taskgraph weiter aufgebaut, der dann von den Workern abgearbeitet wird. An dieser Stelle muss der Nutzer die Chunks der Daten verstehen, die zu der Berechnung optimal passt. Im Beispiel

nehmen wir jeden 12. Wert aus der ersten Dimension, was grob einem Jahreswert entspricht und berechnen die Standardabweichung über genau diese Axe. Mit `visualize()` wird der Taskgraph aufgezeigt. Da die Daten in auch genau der Zeitdimension in zwei geteilt ist wird der erste Schritt der Jahreswerte getrennt parallel berechnet. Erst als alle Daten gebraucht werden für die Standardabweichung über beide Chunks vereint sich die Berechnung und reduziert die Dimension.

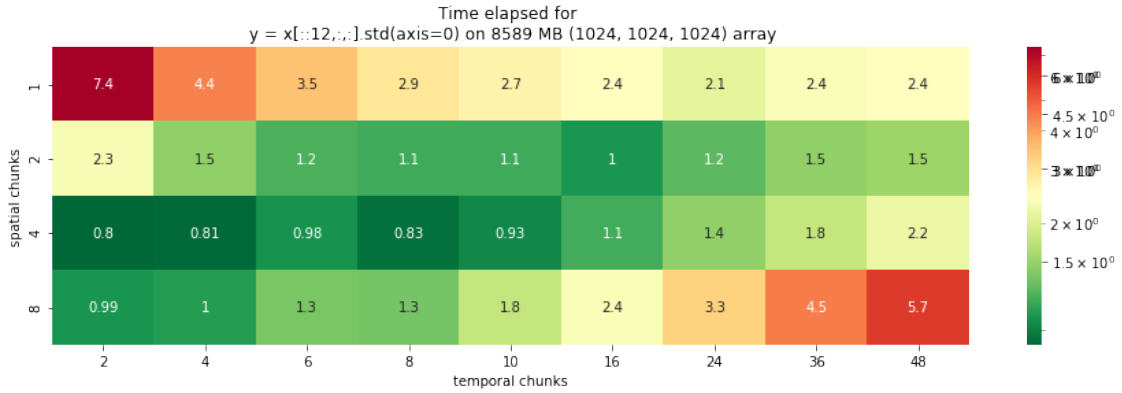
```
In [37]: y = x[::12,:,:].std(axis=0)

In [38]: y

Out[38]: dask.array<sqrt, shape=(500, 500), dtype=float64, chunksize=(500, 500)>

In [39]: y.visualize(optimize_graph=False,rankdir='LR')

Out[39]:
```




```
In [40]: %time y=y.compute()
         y.shape

CPU times: user 1.21 s, sys: 537 ms, total: 1.75 s
Wall time: 947 ms


Out[40]: (500, 500)
```

### 4.3 Benchmark chunking

Das Ergebnis ist unabhängig vom Chunking. Allerdings kann zu viel oder zu wenig die Performance mindern. *Dask Documentation* empfiehlt chunks von 1000x1000 Werten, was allerdings von der Art der Berechnung und der zur Verfügung stehenden Hardware abhängig ist, was die nachfolgende Heatmap aufweist (Details siehe Notebook).




Allerdings ist Parallelisierung nicht automatisch schneller. Der Task Scheduler produziert Overhead, sodass erst genügend große Datensätz zu einem Performancegewinn führen.

```
In [67]: x1 = dask.array.random.random((ntsize, nssize, nssize), chunks=(int(ntsize/t), int(nssi
         x1=x1.compute()
         %time y=x1[::12,:,:].std(axis=0)

CPU times: user 576 ms, sys: 362 ms, total: 938 ms
Wall time: 820 ms
```

### 4.4 `xarray` nutzt `dask`

Sobald beim Landen der Daten mit `xarray` das Argument `chunks` mitgegeben wird oder das `xarray.Dataset` nachträglich gechunkt wird mit `chunk()`, wird läuft dask im Hintergrund ab, was man an der `chunksize` im repl sehen kann. Der Datenanalyse-Algorithmus muss dann häufig nicht mal mehr angepasst werden.

```
In [69]: # xr.open_mfdataset oder xr.open_dataset(chunks={dict})
         hadisst = xr.open_mfdataset(urlpath,chunks={'time':450})['sst'].squeeze()

In [70]: hadisst

Out[70]: <xarray.DataArray 'sst' (time: 1789, latitude: 180, longitude: 360)>
         dask.array<shape=(1789, 180, 360), dtype=float32, chunksize=(450, 180, 360)>
         Coordinates:
           * time       (time) datetime64[ns] 1870-01-16T11:59:59.505615234 ... 2019-01-16T12:00
           * latitude   (latitude) float32 89.5 88.5 87.5 86.5 ... -87.5 -88.5 -89.5
           * longitude  (longitude) float32 -179.5 -178.5 -177.5 ... 177.5 178.5 179.5
         Attributes:
             standard_name:  sea_surface_temperature
             long_name:      sst
             units:          C
             cell_methods:   time: lat: lon: mean
```

# 5 dask auf Hochleistungsrechnern

## 5.1 Client

Für mehr Einblicke in was dask im Hintergrund tut, kann man einen Client starten. Wenn dieser initialisiert ist und das dask Dashboard [https://distributed.dask.org/en/latest/web.html] installiert ist, sind einige Diagnostiken schön visualisert und in Jupyter einsehbar, die durch Profiling erstellt wurden.

```
In [59]: from dask.distributed import Client
         client = Client()

In [60]: client

Out[60]: <Client: scheduler='tcp://127.0.0.1:40775' processes=9 cores=72>
```

## 5.2 Cluster

Um nicht nur auf einem Knoten eines Supercomputers zu rechnen, sondern auf mehreren, kann auch ein Cluster starten, um die Rechenlast auf mehrere Knoten zu verteilen. Dazu wird im Hintergrund ein SLURM Job erstellt und damit freie Knoten reserviert. Der Datenanalyse-Code muss meistens nicht oder nur kaum angepasst werden. Allerdings ist bei Parallelisierung vorsicht geboten. Während es mit dask sehr einfach ist sehr viel Rechenkapazität für eine Berechnung bereitszustellen, muss vorher das Parallelisierungsproblem genügend verstanden sein, damit der Overhead nicht die Parallelizität übertrumpft.

```
In [137]: from dask.distributed import Client, LocalCluster
          from dask_jobqueue import SLURMCluster

In [137]: cluster = SLURMCluster()
          cluster.scale(30)
```

# 6 Ausblick

Im Ausblick folgen einige Projekte und python Packages, die in Kombination mit `xarray` die Datenanalyse sehr vereinfachen und die Datenformatkonversionen bereitstellen.

## 6.1 Nützliche Projekte und Erweiterungen

- `scipy` : (fast) alle Funktionen anwendbar mit `xr.apply_ufunc`
- `cartopy` : Kartenprojektionen
- `seaborn` : Visualisierung von statistischen Graphiken
- `bokeh` : Dynamische Visualisierung von statistischen Graphiken
- `geoviews` : Dynamische Visualisierung von Kartenprojektionen
- `intake` : Laden von ähnlichen `.csv`-Dateien durch Kataloge
- `intake-xarray` : `intake` für `netcdf`
- `intake-esm` : `intake` für Erdsystemmodeloutput (CMIP auf mistral)
- `climpred` : Vorhersage-Verifikation
- ... http://xarray.pydata.org/en/stable/related-projects.html

## 6.2 Datentyp-Kompatibilität

- `ds.to_dataframe()` : `xarray` → `pandas`
- `ds.from_dataframe(df)` : `pandas.df` → `xarray`
- `ds['var].values` : `xarray` → `numpy.ndarray`
- `ds.to_netcdf()` : `xarray` → `netcdf`
- `ds.to_zarr()` : `xarray` → `zarr` (Cloudspeicherformat)
- `intake.cat.item.to_dask()` : Katalogisierte `netcdf` → `xarray.dask`
- `cdo.operator(input=ifile, returnXDataset=True)` : `cdo-py` Output → `xarray.dataset`
- ... http://xarray.pydata.org/en/stable/api.html#io-conversion

# 7 Fazit

xarray ist ein intuitives Interface zum Bearbeiten mehrdimensionaler Daten, wie sie unter Anderem bei Klimamodeloutput vorkommen. Mit `dask` kann Big Data gechunkt werden und damit mehrere Prozessoren ausgenutzt werden. Parallelisierung macht die Berchnung nicht immer automatisch schneller. Parallelisierung muss trotzdem vom Nutzer verstanden werden und die Datengrundlage groß genug für sinnvolle Chunkgrößen sein. Das Ökosystem des wissenschaftlichen Rechnen in Python bietet viele andockende Programme zu array, welche meist auf höheren Abstraktionsebene arbeiten, wobei `xarray` auch selbst low-level Programme nutzt. Beim Arbeiten mit `xarray` und `dask` empfiehlt sich das Studium der Dokumentations-Webseiten. Auch hier gilt RTFM (Read the fucking manual). Da die weitverbreitete Berechnungs- und Visualisierungssoftware für Klimawissenschaftler NCL (`http://ncl.ucar.edu/`) auch auf Python basierend auf `xarray` umsteigen wird, müssen einige Klimawissenschaftler auf kurz oder lang sowie auf python und `xarray` umsteigen. Alle, die den Sprung gewagt haben, sind dankbar für diese Entscheidung (q.e.d.).

# Literatur


*Dask Documentation*. URL: `https://docs.dask.org/en/latest/` (besucht am 04. 06. 2019) (siehe S. 10, 11).

Hoyer, Stephan und Joe Hamman (2017). "Xarray: N-D Labeled Arrays and Datasets in Python". In: *Journal of Open Research Software* 5.1. DOI: `10/gdqdmw` (siehe S. 5).

Hunter, J. D. (2007). "Matplotlib: A 2D Graphics Environment". In: *Computing in Science Engineering* 9.3, S. 90–95. DOI: `10/drbjhg` (siehe S. 2).

*Python Visualization Landscape*. *Jake VanderPlas The Python Visualization Landscape PyCon 2017* (siehe S. 1).

McKinney, Wes (2010). "Data Structures for Statistical Computing in Python". In: Proceedings of the 9th Python in Science Conference. Hrsg. von Stéfan van der Walt und Jarrod Millman, S. 51–56 (siehe S. 5).

Met Office (2010–2015). *Cartopy: A Cartographic Python Library with a Matplotlib Interface* (siehe S. 7).

Rocklin, Matthew (2015). "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling". In: Python in Science Conference. Austin, Texas, S. 126–132. DOI: `10/gfz6s5` (siehe S. 10).

Shen, Helen (2014). "Interactive Notebooks: Sharing the Code". In: *Nature News* 515.7525, S. 151. DOI: `10/gdvjdw` (siehe S. 1).

Walt, S. van der, S. C. Colbert und G. Varoquaux (2011). "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science Engineering* 13.2, S. 22–30. DOI: `10/d8k4p9` (siehe S. 1).

*Xarray Documentation*. URL: `http://xarray.pydata.org/en/stable/index.html` (besucht am 04. 06. 2019) (siehe S. 6).