xmitgcm Documentation

Release 0.2.2-11-g4d7e720

Ryan Abernathey

Sep 22, 2018

Contents

xmitgcm is a python package for reading MITgcm binary MDS files into xarray data structures. By storing data in dask arrays, xmitgcm enables parallel, out-of-core analysis of MITgcm output data.

CHAPTER 1

Installation

1.1 Requirements

xmitgcm is compatible with python 3 and python 2.7. It requires xarray (>= version 0.8.2) and dask (>= version 0.11.2). These packages are most reliably installed via the conda environment management system, which is part of the Anaconda python distribution. Assuming you have conda available on your system, the dependencies can be installed with the command:

```
conda install xarray dask
```

If you are using earlier versions of these packages, you should update before installing xmitgcm.

1.2 Installation via conda

If you just want to use xmitgcm, anaconda users can install with:

```
conda install -c conda-forge xmitgcm
```

This will install the latest conda-forge build.

1.3 Installation via pip

Alternatively, xmitgcm can be installed via pip:

```
pip install xmitgcm
```

This will automatically install the latest release from pypi.

1.4 Installation from github

xmitgcm is under active development. To obtain the latest development version, you may clone the source repository and install it:

```
git clone https://github.com/xgcm/xmitgcm.git
cd xmitgcm
python setup.py install
```

Users are encouraged to fork xmitgcm and submit issues _ and pull requests.

CHAPTER 2

Quick Start

First make sure you understand what an xarray Dataset object is. Then find some MITgcm MDS data. If you don't have any data of your own, you can download the xmitgcm test repositories To download the some test data, run the shell commands:

```
$ curl -L -J -O https://ndownloader.figshare.com/files/6494718
$ tar -xvzf global_oce_latlon.tar.gz
```

This will create a directory called `global_oce_latlon` which we will use for the rest of these examples. If you have your own data, replace this with the path to your mitgcm files.

To opean MITgcm MDS data as an xarray.Dataset, do the following in python:

```
from xmitgcm import open_mdsdataset
data_dir = './global_oce_latlon'
ds = open_mdsdataset(data_dir)
# display the contents of the Dataset
print(ds)
>>> <xarray.Dataset>
Dimensions:                (XC: 90, XG: 90, YC: 40, YG: 40, Z: 15, Zl: 15, Zp1: 16,␣
↪Zu: 15, layer_1RHO_bounds: 31, layer_1RHO_center: 30, layer_1RHO_interface: 29,␣
↪time: 1)
Coordinates:
    iter                   (time) int64 39600
  * time                   (time) int64 39600
  * XC                     (XC) >f4 2.0 6.0 10.0 14.0 18.0 22.0 26.0 30.0 ...
  * YC                     (YC) >f4 -78.0 -74.0 -70.0 -66.0 -62.0 -58.0 -54.0 ...
  * XG                     (XG) >f4 0.0 4.0 8.0 12.0 16.0 20.0 24.0 28.0 32.0 ...
  * YG                     (YG) >f4 -80.0 -76.0 -72.0 -68.0 -64.0 -60.0 -56.0 ...
  * Zl                     (Zl) >f4 0.0 -50.0 -120.0 -220.0 -360.0 -550.0 ...
  * Zu                     (Zu) >f4 -50.0 -120.0 -220.0 -360.0 -550.0 -790.0 ...
  * Z                      (Z) >f4 -25.0 -85.0 -170.0 -290.0 -455.0 -670.0 ...
  * Zp1                    (Zp1) >f4 0.0 -50.0 -120.0 -220.0 -360.0 -550.0 ...
    dxC                    (YC, XG) >f4 92460.4 92460.4 92460.4 92460.4 ...
    rAs                    (YG, XC) >f4 3.43349e+10 3.43349e+10 3.43349e+10 ...
```

(continues on next page)

(continued from previous page)

```
    rAw                    (YC, XG) >f4 4.11097e+10 4.11097e+10 4.11097e+10 ...
    Depth                  (YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    rA                     (YC, XC) >f4 4.11097e+10 4.11097e+10 4.11097e+10 ...
    dxG                    (YG, XC) >f4 77223.1 77223.1 77223.1 77223.1 ...
    dyG                    (YC, XG) >f4 444710.0 444710.0 444710.0 444710.0 ...
    rAz                    (YG, XG) >f4 3.43349e+10 3.43349e+10 3.43349e+10 ...
    dyC                    (YG, XC) >f4 444710.0 444710.0 444710.0 444710.0 ...
    PHrefC                 (Z) >f4 245.25 833.85 1667.7 2844.9 4463.55 6572.7 ...
    drC                    (Zp1) >f4 25.0 60.0 85.0 120.0 165.0 215.0 265.0 ...
    PHrefF                 (Zp1) >f4 0.0 490.5 1177.2 2158.2 3531.6 5395.5 ...
    drF                    (Z) >f4 50.0 70.0 100.0 140.0 190.0 240.0 290.0 ...
    hFacS                  (Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    hFacC                  (Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    hFacW                  (Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
  * layer_1RHO_center      (layer_1RHO_center) float32 20.1999 20.6922 21.169 ...
  * layer_1RHO_interface   (layer_1RHO_interface) >f4 20.4499 20.9345 21.4034 ...
  * layer_1RHO_bounds      (layer_1RHO_bounds) >f4 19.9499 20.4499 20.9345 ...
Data variables:
    tFluxtave              (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    PHLtave                (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    Stave                  (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    UUtave                 (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    LaHw1RHO               (time, layer_1RHO_center, YC, XG) >f4 0.0 0.0 0.0 ...
    LaPs1RHO               (time, layer_1RHO_center, YG, XC) >f4 0.0 0.0 0.0 ...
    LaHs1RHO               (time, layer_1RHO_center, YG, XC) >f4 0.0 0.0 0.0 ...
    LaUH1RHO               (time, layer_1RHO_center, YC, XG) >f4 0.0 0.0 0.0 ...
    LaVH1RHO               (time, layer_1RHO_center, YG, XC) >f4 0.0 0.0 0.0 ...
    LaPw1RHO               (time, layer_1RHO_center, YC, XG) >f4 0.0 0.0 0.0 ...
    UVtave                 (time, Z, YG, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    uFluxtave              (time, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    VStave                 (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    VTtave                 (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    TTtave                 (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    PhHytave               (time, Z, YC, XC) >f4 -8.30019 -8.30019 -8.30019 ...
    sFluxtave              (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    W                      (time, Zl, YC, XC) >f4 -0.0 -0.0 -0.0 -0.0 -0.0 ...
    ETAtave                (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    VVtave                 (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    Ttave                  (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    PH                     (time, Z, YC, XC) >f4 -8.30019 -8.30019 -8.30019 ...
    vVeltave               (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    UTtave                 (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    PHL2tave               (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    UStave                 (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    uVeltave               (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    S                      (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    Eta                    (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    Eta2tave               (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    DFxE_TH                (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    ADVy_TH                (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    VTHMASS                (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    DFrE_TH                (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    WTHMASS                (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    TOTTTEND               (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    ADVr_TH                (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    DFyE_TH                (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    UTHMASS                (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
```

(continues on next page)

(continued from previous page)

```
    DFrI_TH                (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    ADVx_TH                (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    surForcT               (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    TFLUX                  (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    surForcS               (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    SFLUX                  (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    V                      (time, Z, YG, XC) >f4 -0.0 -0.0 -0.0 -0.0 -0.0 0.0 ...
    LaTs1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LTha1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LaSz1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LSto1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LSha1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LaTz1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LaSs1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LaTh1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LTto1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LTza1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LaSh1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    LSza1RHO               (time, layer_1RHO_interface, YC, XC) >f4 0.0 0.0 ...
    vFluxtave              (time, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    WTtave                 (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    WStave                 (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    GM_Kwz-T               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    PHL                    (time, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    THETA                  (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    UVEL                   (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    VVEL                   (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    WVEL                   (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    SALT                   (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    DFrI_SLT               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    WSLTMASS               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    ADVx_SLT               (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    ADVr_SLT               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    TOTSTEND               (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    USLTMASS               (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    DFxE_SLT               (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    DFyE_SLT               (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    DFrE_SLT               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    ADVy_SLT               (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    VSLTMASS               (time, Z, YG, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    Convtave               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    wVeltave               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    GM_Kwy-T               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    U                      (time, Z, YC, XG) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    Tdiftave               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    GM_Kwx-T               (time, Zl, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
    T                      (time, Z, YC, XC) >f4 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
```

`data_dir`, should be the path (absolute or relative) to an MITgcm run directory. xmitgcm will automatically scan this directory and try to determine the file prefixes and iteration numbers to read. In some configurations, the `open_mdsdataset` function may work without further keyword arguments. In most cases, you will have to specify further details.

For more details about the options and internals, consult *Reading MDS Data*.

CHAPTER 3

Reading MDS Data

3.1 open_mdsdataset

All loading of data in xmitgcm occurs through the function `open_mdsdataset`. Its full documentation below enumerates all the possible options.

`xmitgcm.`**`open_mdsdataset`**(*data_dir*, *grid_dir=None*, *iters='all'*, *prefix=None*, *read_grid=True*, *delta_t=1*, *ref_date=None*, *calendar='gregorian'*, *geometry='sphericalpolar'*, *grid_vars_to_coords=True*, *swap_dims=None*, *endian='>'*, *chunks=None*, *ignore_unknown_vars=False*, *default_dtype=None*, *nx=None*, *ny=None*, *nz=None*, *llc_method='smallchunks'*, *extra_metadata=None*)

Open MITgcm-style mds (.data / .meta) file output as xarray datset.

Parameters **data_dir** : string

Path to the directory where the mds .data and .meta files are stored

grid_dir : string, optional

Path to the directory where the mds .data and .meta files are stored, if different from `data_dir`.

iters : list, optional

The iterations numbers of the files to be read. If `None`, no data files will be read. If `'all'` (default), all iterations will be read.

prefix : list, optional

List of different filename prefixes to read. Default (`None`) is to read all available files.

read_grid : bool, optional

Whether to read the grid data

delta_t : number, optional

The timestep used in the model. (Can't be inferred.)

ref_date : string, optional

An iSO date string corresponding to the zero timestep, e.g. "1990-1-1 0:0:0" (See CF conventions [R1])

calendar : string, optional

A calendar allowed by CF conventions [R1]

geometry : {'sphericalpolar', 'cartesian', 'llc', 'curvilinear'}

MITgcm grid geometry specifier

grid_vars_to_coords : boolean, optional

Whether to promote grid variables to coordinate status

swap_dims : boolean, optional

Whether to swap the logical dimensions for physical ones. If `None`, will be set to `False` for `geometry==llc` and `True` otherwise.

endian : {'=', '>', '<'}, optional

Endianness of variables. Default for MITgcm is ">" (big endian)

chunks : int or dict, optional

If chunks is provided, it used to load the new dataset into dask arrays.

ignore_unknown_vars : boolean, optional

Don't raise an error if unknown variables are encountered while reading the dataset.

default_dtype : numpy.dtype, optional

A datatype to fall back on if the metadata can't be read.

nx, ny, nz : int, optional

The numerical dimensions of the model. These will be inferred from `XC.meta` and `RC.meta` if they are not specified. If `geometry==llc`, `ny` does not have to specified.

llc_method : {"smallchunks", "bigchunks"}, optional

Which routine to use for reading LLC data. "smallchunks" splits the file into a individual dask chunk of size (nx x nx) for each face of each level (i.e. the total number of chunks is 13 * nz). "bigchunks" loads the whole raw data file (either into memory or as a numpy.memmap), splits it into faces, and concatenates those faces together using `dask.array.concatenate`. The different methods will have different memory and i/o performance depending on the details of the system configuration.

extra_metadata : dict, optional

Allow to pass information on llc type grid (global or regional). The additional metadata is typically such as :

aste = {'has_faces': True, 'ny': 1350, 'nx': 270, 'ny_facets': [450,0,270,180,450], 'pad_before_y': [90,0,0,0,0], 'pad_after_y': [0,0,0,90,90], 'face_facets': [0, 0, 2, 3, 4, 4], 'facet_orders' : ['C', 'C', 'C', 'F', 'F'], 'face_offsets' : [0, 1, 0, 0, 0, 1], 'transpose_face' : [False, False, False,

True, True, True]}

For global llc grids, no extra metadata is required and code will set up to global llc default configuration.

Returns **dset** : xarray.Dataset

Dataset object containing all coordinates and variables.

References

[R1]

The optional arguments are explained in more detail in the following sections and examples.

3.2 Lazy Evaluation and Dask Chunking

`open_mdsdataset` does not actually load all of the data into memory when it is invoked. Rather, it opens the files using numpy.memmap, which means that the data is not read until it is actually needed for a computation. This is a cheap form of lazy evaluation.

Additional performance benefits can be achieved with xarray dask chunking:

```
ds_chunked = open_mdsdataset(data_dir, chunks={'Z':1, 'Zl':1})
```

In the example above, the each horizontal slice of the model is assigned to its own chunk; dask will automatically try to parellize operations across these chunks using all your CPU cores. For this small example dataset, no performance boost is expected (due to the overhead of parallelization), but very large simulations will certainly benefit from chunking.

When chunking is applied, the data are represetned by dask arrays, and all operations are evaluated lazily. No computation actually takes place until you call `.load()`:

```
# take the mean of the squared zonal velocity
(ds_chunked.U**2).mean()
>>> <xarray.DataArray 'U' ()>
dask.array<mean_ag..., shape=(), dtype=float32, chunksize=()>
# load the value and execute the dask computational graph
>>> <xarray.DataArray 'U' ()>
array(0.00020325234800111502, dtype=float32)
```

Note: In certain cases, chunking is applied automatically. These cases are

- If there is more than one timestep to be read (see *Expected Files*) the time dimension is automatically chunked.
- In llc simulations, (see *Grid Geometries*) the `face` dimension is automatically chunked.

3.3 Expected Files

MITgcm writes MDS output as pairs of files with the suffixes `*.data` and `*.meta`. The model timestep iteration number is represented by a ten-digit number at the end of the filename, e.g. `T.0000000090.data` and `T.0000000090.meta`. MDS files without an iteration number are grid files.

xmitgcm has certain expectations regarding the files that will be present in `datadir`. In particular, it assumes `datadir` is an MITgcm "run" directory. By default, *open_mdsdataset* will read the grid files which describe the geometry of the computational domain. If these files are not present in `datadir`, this behavior can be turned off by setting `read_grid=False`.

In order to determine the dimensions of the model domain `open_mdsdataset` needs to peek at the metadata in two grid files: `XC.meta` and `RC.meta`. (even when `read_grid=False`). If these files are not available, you have the option to manually specify the parameters `nx`, `ny`, and `nz` as keyword arguments to `open_mdsdataset`. (`ny` is not required for `geometry='llc'`).

By default, `open_mdsdataset` attempts to read all the data files in `datadir`. The files and iteration numbers to read are determined in the following way:

1. First `datadir` is scanned to determine all iteration numbers present in the directory. To override this behavior and manually specify the iteration numbers to read, use the `iters` keyword argument, e.g. `iters=[10,20,30]`.
2. To dertmine the file prefixes to read, `open_mdsdataset` looks for all `*.data` filenames which match the *first* iteration number. To override this behavior and manually specify the file prefixes via the `prefix` keyword argument, e.g. `prefix=['UTave', 'VTave']`.
3. `open_mdsdataset` then looks for each file prefix at each iteration number.

This approach works for the test examples, but perhaps it does not suit your model configuration. Suggestions are welcome on how to improve the discovery of files in the form of issues and pull requests.

Warning: If you have certain file prefixes that are present at the first iteration (e.g. `T.0000000000.data`) but not at later iterations (e.g `iters=[0,10]`) but there is no `T.0000000010.data` file, `open_mdsdataset` will raise an error because it can't find the expected files. To overcome this you need to manually specify `iters` and / or `prefix`.

To determine the variable metadata, xmitgcm is able to parse the model's `available_diagnostics.log` file. If you use diagnostic output, the `available_diagnostics.log` file corresponding with your model run should be present in `datadir`.

Note: If the `available_diagnostics.log` file can't be found, a default version will be used. This could lead to problems, since you may have custom diagnostics enabled in your run that are not present in the default. The default `available_diagnostics.log` file was taken from the ECCOv4 `global_oce_llc90` experiment.

For non-diagnostic output (e.g. default "state" or "timeave" output), xmitgcm assigns the variable metadata based on filenames. The additional metadata makes the internal represtation of the model variables more verbose and ensures compliance with CF Conventions.

3.4 Dimensions and Coordinates

One major advantage of using xarray to represent data is that the variable dimensions are *labeled*, much like netCDF data structures. This labeling enables much clearer code. For example, to take a time average of a Dataset, one just says `ds.mean(dim='time')` without having to remember which logical axis is the time dimension.

xmitgcm distinguishes between *logical dimensions* and *physical dimensions* or coordinates. Open `open_mdsdataset` will attempt to assign physical dimensions to the data. The physical dimensions correspond to the axes of the MITgcm grids in `cartesian` or `sphericalpolar` coordinates. The standard names have been assigned according to CF Conventions.

name	standard_name
time	time
XC	longitude
YC	latitude
XG	longitude_at_f_location
YG	latitude_at_f_location
Zl	depth_at_upper_w_location
Zu	depth_at_lower_w_location
Z	depth
Zp1	depth_at_w_location
layer_1RHO_center	ocean_layer_coordinate_1RHO_center
layer_1RHO_interface	ocean_layer_coordinate_1RHO_interface
layer_1RHO_bounds	ocean_layer_coordinate_1RHO_bounds

The physical dimensions of typical veriables are:

```
print(ds.THETA.dims)
>>> ('time', 'Z', 'YC', 'XC')
print(ds.UVEL.dims)
>>> ('time', 'Z', 'YC', 'XG')
print(ds.VVEL.dims)
>>> ('time', 'Z', 'YG', 'XC')
print(ds.WVEl.dims)
>>> ('time', 'Zl', 'YC', 'XG')
```

In order for physical dimensions to be assigned `open_mdsdataset` must be involved with `read_grid=True` (default). For a more minimalistic approach, one can use `read_grid=False` and assign logcial dimensions. Logical dimensions can also be chosen by explicitly setting `swap_dims=False`, even with `read_grid=False`. Physical dimension only work with `geometry=='cartesian'` or `geometry=='sphericalpolar'`. For `geometry=='llc'` or `geometry=='curvilinear'`, it is not possible to replace the logical dimensions with physical dimensions, and setting `swap_dims=False` will raise an error.

Logical dimensions follow the naming conventions of the MITgcm numerical grid. The dimension names are augmented with metadata attributes from the Comodo conventions. These logical spatial dimensions are

name	standard_name	axis	c_grid_axis_shift
i	x_grid_index	X	
i_g	x_grid_index_at_u_location	X	-0.5
j	y_grid_index	Y	
j_g	y_grid_index_at_v_location	Y	-0.5
k	z_grid_index	Z	
k_u	z_grid_index_at_lower_w_location	Z	-0.5
k_l	z_grid_index_at_upper_w_location	Z	0.5
k_p1	z_grid_index_at_w_location	Z	(-0.5, 0.5)

As explained in the Comodo documentation, the use of different dimensions is necessary to represent the fact that, in c-grid ocean models, different variables are staggered in different ways with respect to the model cells. For example, tracers and velocity components are all have different logical dimensions:

```
print(ds.THETA.dims)
>>> ('time', 'k', 'j', 'i')
print(ds.UVEL.dims)
>>> ('time', 'k', 'j', 'i_g'
```

(continues on next page)

(continued from previous page)

```
print(ds.VVEL.dims)
>>> ('time', 'k', 'j_g', 'i')
print(ds.WVEl.dims)
>>> ('time', 'k_l', 'j', 'i')
```

xarray distinguishes between "coordinates" and "data_vars". By default, `open_mdsdataset` will promote all grid variables to coordinates. To turn off this behavior and treat grid variables as data_vars, use `grid_vars_to_coords=False`.

3.5 Time

`open_mdsdataset` attemts to determine the time dimension based on `iters`. However, addtiional input is required from the user to fully exploit this capability. If the user specifies `delta_t`, the numerical timestep used for the MITgcm simulation, it is used to muliply `iters` to determine the time in seconds. Additionally, if the user specifies `ref_date` (an ISO date string, e.g. `"1990-1-1 0:0:0"`), the time dimension will be converted into a datetime index, exposing all sorts of useful timeseries functionalty within xarray.

3.6 Grid Geometries

The grid geometry is not inferred; it must be specified via the `geometry` keyword. xmitgcm currently supports four MITgcm grid geometries: `cartesian`, `sphericalpolar`, `curvilinear`, and `llc`. The first two are straightforward. The `curvilinear` is used for curvilinear cartesian grids. The `llc` ("lat-lon-cap") geometry is more complicated. This grid consists of four distinct faces of the same size plus a smaller north-polar cap. Each face has a distinct relatioship between its logical dimensions and its physical coordinates. Because netCDF and xarray.Dataset data structures do not support this sort of complex geometry (multiple faces of different sizes), our approach, inspired by nc-tiles, is to split the domain into 13 equal-sized "faces". `face` then becomes an additional dimension of the data.

To download an example llc dataset, run the following shell commands:

```
$ curl -L -J -O https://ndownloader.figshare.com/files/6494721
$ tar -xvzf global_oce_llc90.tar.gz
```

And to read it, in python:

```
ds_llc = open_mdsdataset('./global_oce_llc90/', iters=8, geometry='llc')
print(ds_llc['S']dims)
>>> ('time', 'k', 'face', 'j', 'i')
```

xmitgcm is not nearly as comprehensive as gcmfaces. It does not offer sophisticated operations involving exchanges at face boundaries, integrals across sections, etc. The goal of this package is simply to read the mds data. However, by outputing an xarray data structure, we can use all of xarray's usual capabilities on the llc data, for example:

```
# calculate the mean squared salinity as a function of depth
(ds_llc.S**2).mean(dim=['face', 'j', 'i'])
>>> <xarray.DataArray 'S' (time: 1, k: 50)>
dask.array<mean_ag..., shape=(1, 50), dtype=float32, chunksize=(1, 50)>
Coordinates:
  * k         (k) int64 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...
    iter      (time) int64 8
  * time      (time) int64 8
```

(continues on next page)

(continued from previous page)

```
    Z        (k) >f4 -5.0 -15.0 -25.0 -35.0 -45.0 -55.0 -65.0 -75.005 ...
    PHrefC   (k) >f4 49.05 147.15 245.25 343.35 441.45 539.55 637.65 735.799 ...
    drF      (k) >f4 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.01 10.03 10.11 ...
```

(Note that this simple example does not perform correct volume weighting or land masking in the average.)

`open_mdsdataset` offers two different strategies for reading LLC data, `method='smallchunks'` (the default) and `method='bigchunks'`. The details and tradeoffs of these strategies are described in *Performance Issues*.

CHAPTER 4

Example Usage

Once you have loaded your data, you can analyze it using all the capabilities available in xarray. Here are a few quick examples.

4.1 Land Masks

xmitgcm simply reads the MDS data directly for the disk; it does not attempt to mask land points, which are usually filled with zero values. To mask land, you can use xarray's `where` function together the the `hFac` variables related to MITgcm's partially filled cells. For example, with the `global_oce_latlon` dataset, an unmasked average of salinity gives:

```
ds.S.mean()
>>> <xarray.DataArray 'S' ()>
array(18.85319709777832, dtype=float32)
```

This value is unrealistically low because it includes all of the zeros inside the land which shold be masked. To take the masked average, instead do:

```
ds.S.where(ds.hFacC>0).mean()
>>> <xarray.DataArray ()>
array(34.73611831665039)
```

This is a more correct value.

4.2 Volume Weighting

However, it is still not properly volume weighted. To take a volume-weighted average, you can do:

```
volume = ds.hFacC * ds.drF * ds.rA
(ds.S * volume).sum() / volume.sum()
```

(continues on next page)

(continued from previous page)

```
>>> <xarray.DataArray ()>
array(34.779126627139945)
```

This represents the correct mean ocean salinity. A different land mask and volume weighting is required for variables located at the u and v points.

4.3 netCDF conversion

Thanks to xarray, it is trivial to convert our dataset to netCDF:

```
ds.to_netcdf('myfile.nc')
```

It can then be reloaded directly with xarray:

```
import xarray as xr
ds = xr.open_dataset('myfile.nc')
```

This is an attractive option for archiving MDS data in a self-contained way.

CHAPTER 5

Performance Issues

A major goal of xmitgcm is to achieve scalable performance with very large datasets. We were motivated by the new LLC4320 simulations run by Dimitris Menemenlis and Chris Hill on NASA's Pleiades supercomputer.

This page documents ongoing research into the performance of xmitgcm.

5.1 LLC Reading Strategies

The physical layout of the LLC MDS files creates a challenge for performance. Some of the cube facets are written with C order, while others are written with Fortran order. This means that complex logic is required to translate the raw data on disk to the desired logical array layout within xarray. Because of this complication, the raw data cannot be accessed using the numpy ndarray data model.

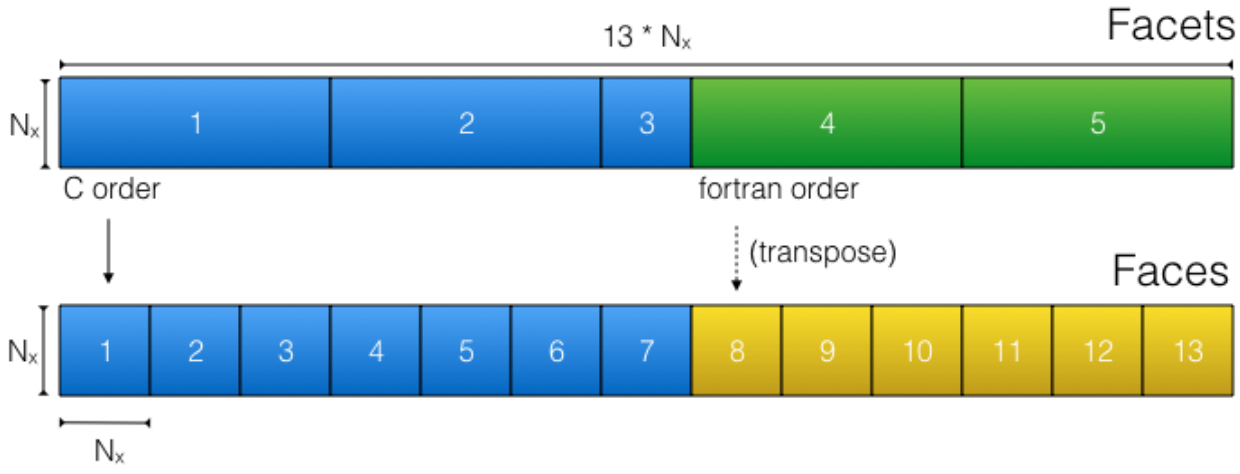

Fig. 1: The physical layout of a single level of LLC data in terms of facets (top) and the translation by xmitgcm to faces (bottom).

Two different approaches to reading LLC data have been developed. This option is specified via the `llc_method` keyword in `open_mdsdataset`.

5.1.1 smallchunks

`method="smallchunks"` creates a large number of small dask chunks to represent the array. A chunk (of size nx x nx) is created for each face of each vertical level. The total number of chunks is consequently 13 * nz. This appears to perform better when you want to access a small subset of a very large model, since only the required data will be read. It has a much lower memory footprint than bigchunks. No memmaps are used withsmallchunks, since that implies leaving a huge number of files open. Instead, each chunk is read directly by `numpy.fromfile`.

5.1.2 bigchunks

`method="bigchunks"` loads the entire raw data on disk as either a `numpy.memmap` (default) or directly as a numpy array. It then slices this array into facets, reshapes them as necessary, slices each facet into faces, and concatenates the faces along a new dimension using `dask.array.concatentate`. This approach can be more efficient if the goal is to read all of the array data into memory. Any attempt to read data from the reshaped faces (faces 8-13 in the cartoon above) will trigger the *entire facet* to be loaded into memory. For this reason, the bigchunks method is impractical for very large LLC datasets that don't easily fit into memory

5.1.3 comparison

A test script was developed to evaluate the two strategies for reading LLC4320 data on Pleiades. Files were selected for analysis randomly from over 10000 files on disk in order to avoid any caching from the filesystem. The data were read using the low level routine `read_3d_llc_data` (see *Low Level Utilities*). Tests were performed on both 2D data (4320 x 56160 32-bit floats) and 3D data (4320 x 56160 x 90 32-bit floats).

The first task was a reduction: computing the sum of the array. For 2D data, the smallchunks method performed marginally better in terms of speed and memory usage.

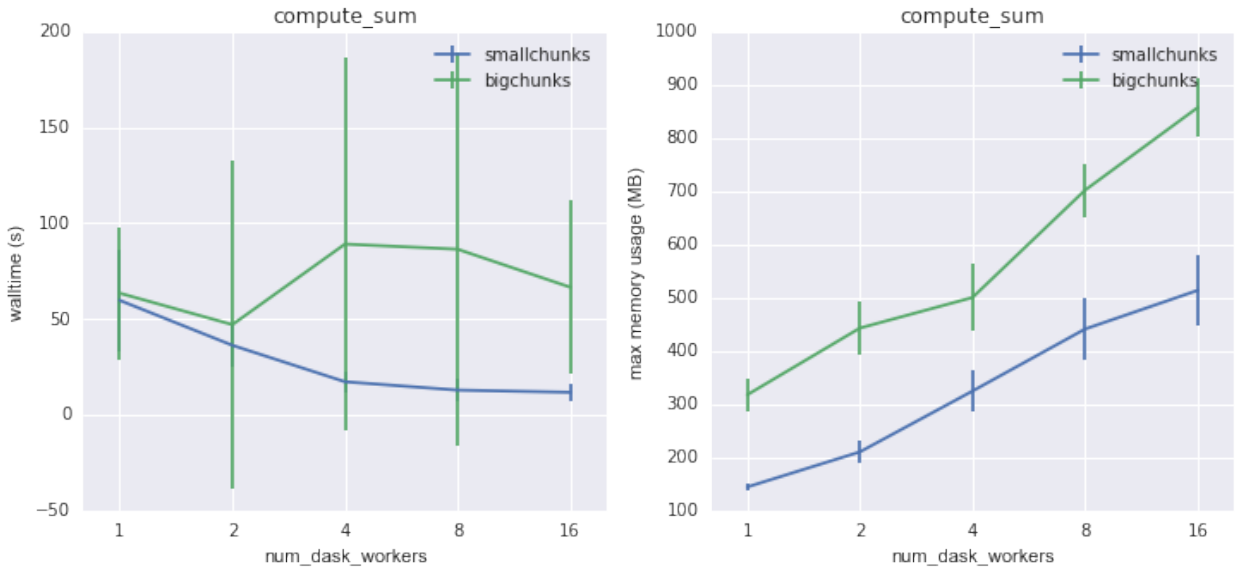

Fig. 2: Walltime and memory usage for `compute_sum` on 2D data as a function of number of dask workers

However, a dramatic difference was evident for 3D data. The inefficient memory usage of bigchunks is especially evident for large numbers of dask workers, since each worker repeatedly triggers the loading of whole array facets.

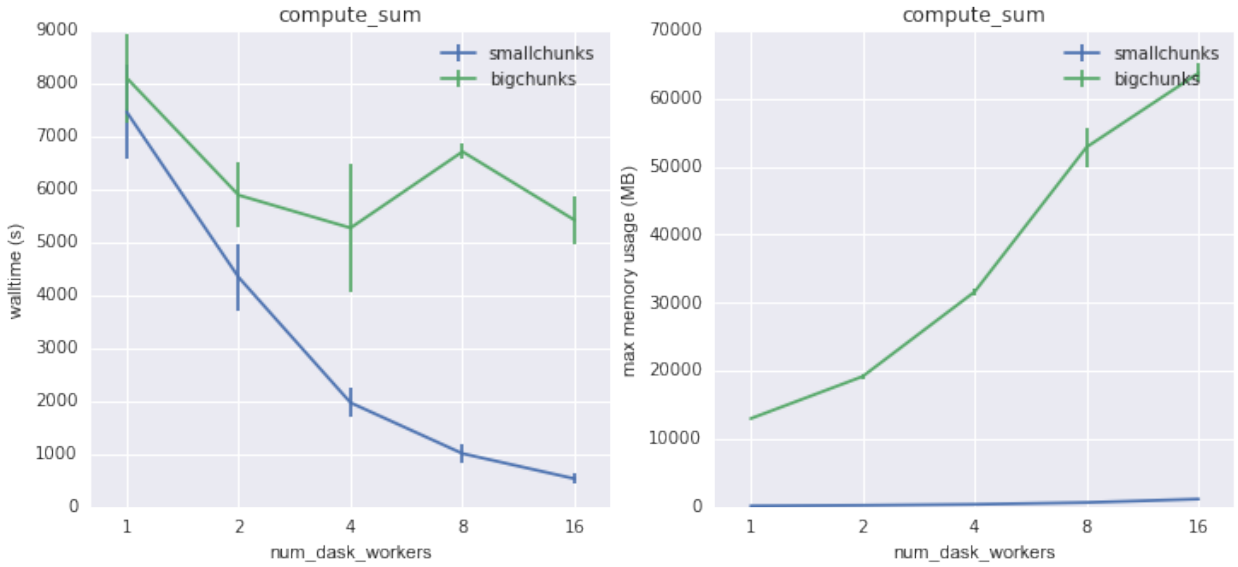

Fig. 3: Walltime and memory usage for `compute_sum` on 3D data as a function of number of dask workers

For this sort of reduction workflow, smallchunks with a large number of dask workers is the clear winner.

A second common workflow is subsetting. In the test script, we load into memory 1080 x 1080 region from chunk 2.

Again, smallchunks is the clear winner here, with much faster execution and lower memory usage. Interestingly, there is little speedup using multiple multiple workers. All the same conclusions are true for 3D data.

A final workload is simply loading the whole array into memory. (This turned out to be impossible for 3D data, since the compute nodes ran out of memory in the process.) This is the only workload where bigchunks has some advantages. Here a tradeoff between speed and memory usage is clear: bigchunks goes faster because it reads the data in bigger chunks, but it also uses much more memory.

It is useful to compare these numbers to the speed of a raw `numpy.fromfile` read of the data. This measures the overhead associated with chunking and reshaping the data from its physical layout on disk to the desired logical layout. Reading with smallchunks takes about 150 times the raw read time, while for bigchunks it is more like 10 times. Here there is a *disadvantage* to using multiple dask workers; while there is no speed improvement, the memory usage increases with number of workers for bigchunks.

5.2 Running xmitgcm on Pleaides

These instructions describe how to get a working xmitgcm environment on a cluster such as Pleiades. (See related blog post)

5.2.1 Step 1: Install miniconda in user space

Miniconda is a mini version of Anaconda that includes just conda and its dependencies. It is a very small download. If you want to use python 3 (recommended) you can call:

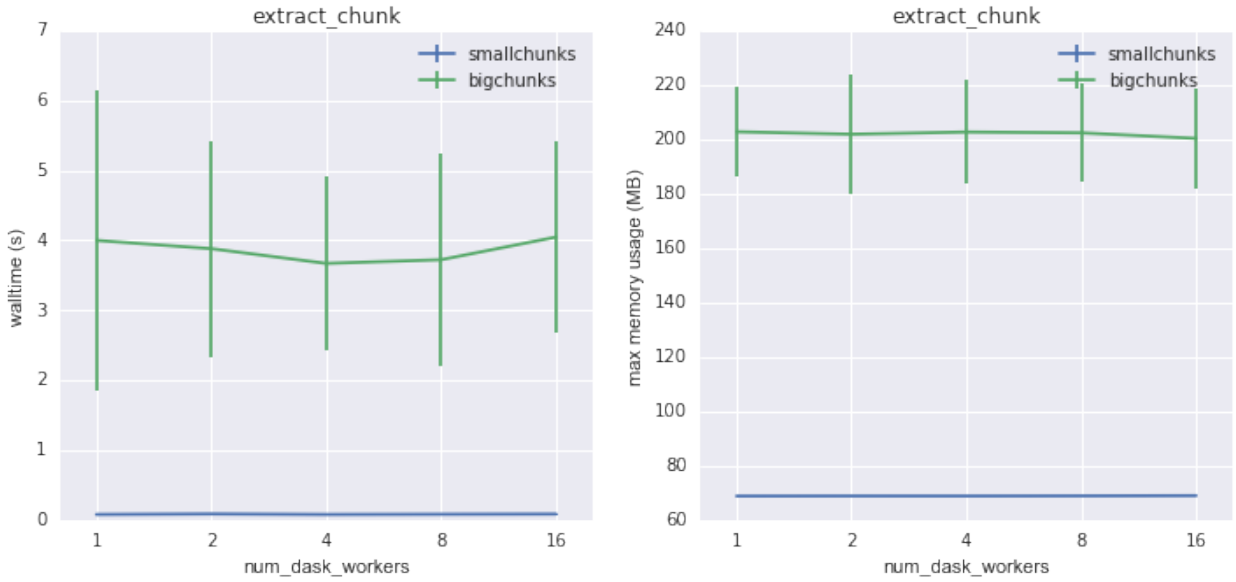

Fig. 4: Walltime and memory usage for `extract_chunk` on 2D data as a function of number of dask workers

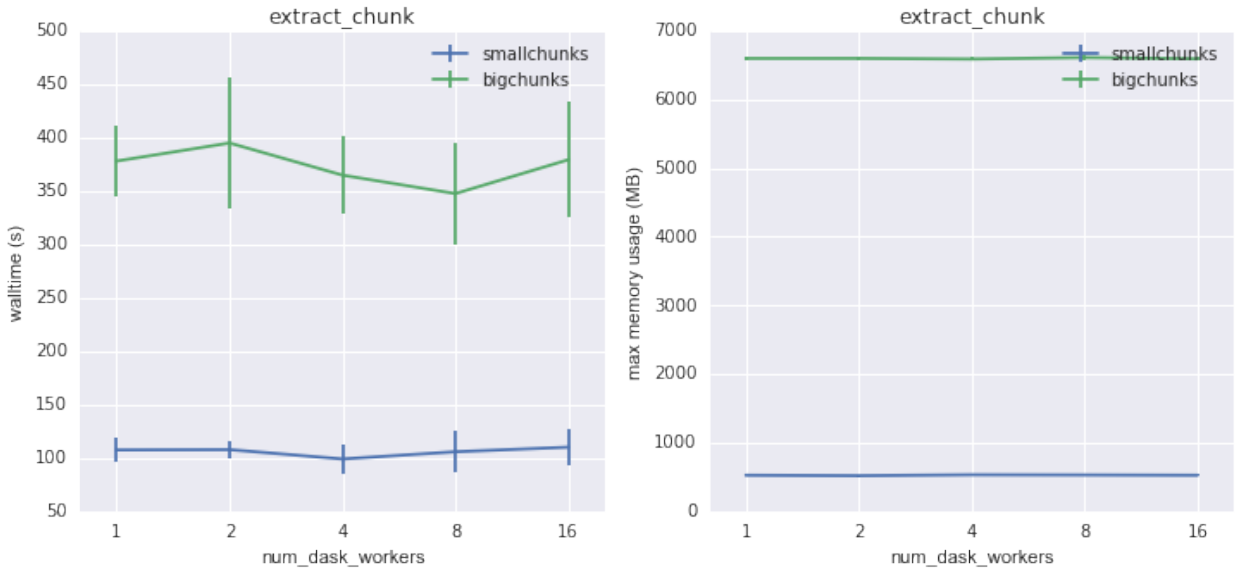

Fig. 5: Walltime and memory usage for `extract_chunk` on 3D data as a function of number of dask workers

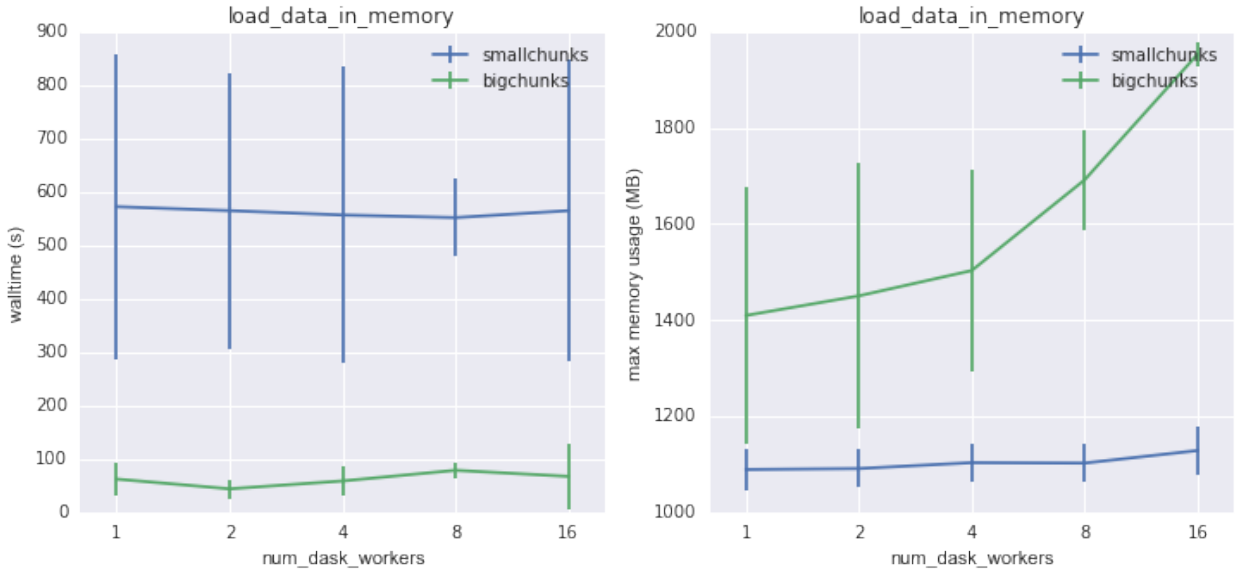

Fig. 6: Walltime and memory usage for `load_data_in_memory` on 2D data as a function of number of dask workers

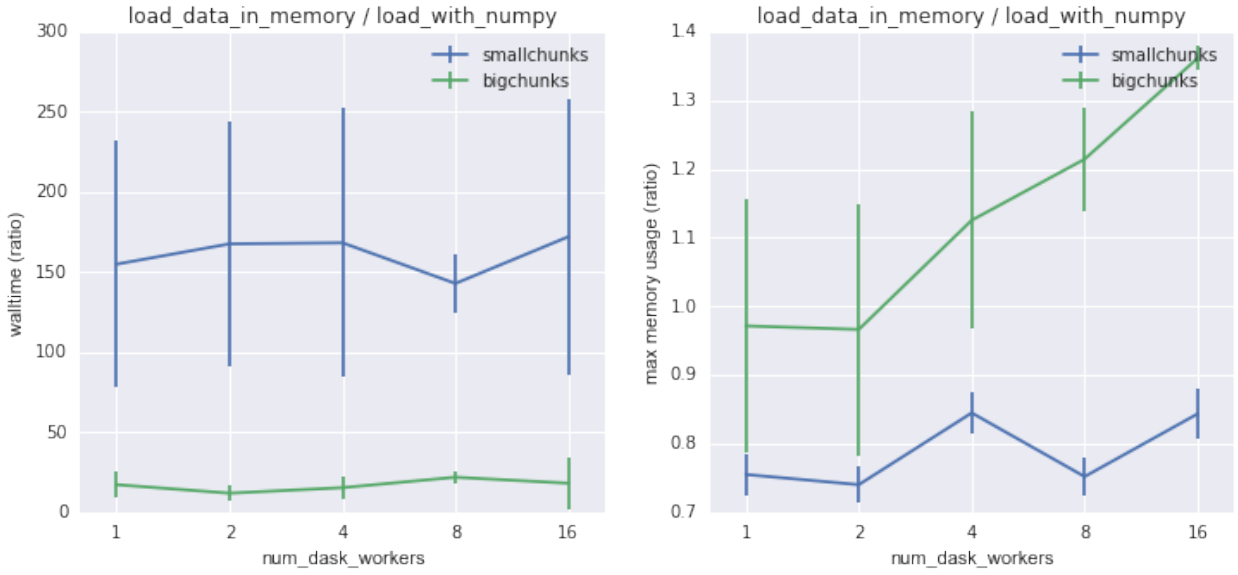

Fig. 7: Walltime and memory usage for `load_data_in_memory` on 2D data as a function of number of dask workers, normalized against loading the same data directly using `numpy.fromfile`.

```
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -O␣
↪miniconda.sh
```

or for python 2.7:

```
wget https://repo.continuum.io/miniconda/Miniconda2-latest-Linux-x86_64.sh -O␣
↪miniconda.sh
```

5.2.2 Step 2: Run Miniconda

Now you actually run miniconda to install the package manager. The trick is to specify the install directory within your home directory, rather in the default system-wide installation (which you won't have permissions to do). You then have to add this directory to your path:

```
bash miniconda.sh -b -p $HOME/miniconda export PATH="$HOME/miniconda/bin:$PATH"
```

5.2.3 Step 3: Create a custom conda environment specification

You now have to define what packages you actually want to install. A good way to do this is with a custom conda environment file. The contents of this file will differ for each project. Below is an environment.yml suitable for xmitgcm:

```
name: xmitgcm
dependencies:
  - numpy
  - scipy
  - xarray
  - netcdf4
  - dask
  - jupyter
  - matplotlib
  - pip:
    - pytest
    - xmitgcm
```

Create a similar file and save it as environment.yml.

5.2.4 Step 4: Create the conda environment

You should now be able to run the following command:

```
conda env create --file environment.yml
```

This will download and install all the packages and their dependencies.

5.2.5 Step 5: Activate The environment

The environment you created needs to be activated before you can actually use it. To do this, you call:

```
source activate xmitgcm
```

This step needs to be repeated whenever you want to use the environment (i.e. every time you launch an interactive job or call python from within a batch job).

5.2.6 Step 6: Use xmitgcm

You can now call ipython on the command line or launch a jupyter notebook and import xmitgcm. This should be done from a compute node, rather than the head node.

5.3 Example Pleiades Scripts

Below is an example python which extracts a subset from the LLC4320 simulation on Pleiades and saves it a sequence of netCDF files.

```
import os
import sys
import numpy as np
import xarray as xr
import dask
from multiprocessing.pool import ThreadPool
from xmitgcm import open_mdsdataset

# By default, dask will use one worker for each core available.
# This can be changed by uncommenting below
#dask.set_options(pool=ThreadPool(4))

# where the data lives
data_dir = '/u/dmenemen/llc_4320/MITgcm/run/'
grid_dir = '/u/dmenemen/llc_4320/grid/'
# where to save the subsets
outdir_base = '/nobackup/rpaberna/LLC/tile_data/'

dtype = np.dtype('>f4')

# can complete 300 files in < 12 hours
nfiles = 300

# the first available iteration is iter0=10368
# we start from an iteration number specified on the command line
iter0 = int(sys.argv[1])

delta = 144 # iters
delta_t = 25. # seconds
all_iters = iter0 + delta*np.arange(nfiles)

region_name = 'agulhas'
region_slice = {'face': 1,
                'i': slice(1080,3240), 'i_g': slice(1080,3240),
                'j': slice(0,2160), 'j_g': slice(0,2160)}

fprefix = 'llc_4320_%s' % region_name
outdir = os.path.join(outdir_base, fprefix)

ds = open_mdsdataset(data_dir, grid_dir=grid_dir,
                     iters=list(all_iters), geometry='llc', read_grid=False,
```

(continues on next page)

(continued from previous page)

```
                    default_dtype=np.dtype('>f4'), delta_t=delta_t,
                    ignore_unknown_vars=True)

region = ds.isel(**region_slice)

# group for writing
iters, datasets = zip(*region.groupby('iter'))
paths = [os.path.join(outdir, '%s.%010d.nc' % (fprefix, d))
         for d in iters]

# write the data...takes a long time and executes in parallel
xr.save_mfdataset(datasets, paths, engine='netcdf4')
```

Here is a batch job which calls the script

```
#!/bin/bash
#PBS -N read_llc
#PBS -l select=1:ncpus=28:model=bro
#PBS -l walltime=12:00:00

source activate xmitgcm
cd $PBS_O_WORKDIR
# the first available iteration
iter0=10368
python -u write_by_iternum.py $iter0
```

CHAPTER 6

Low Level Utilities

These functions can be imported from `xmitgcm.utils`. They are meant for internal use in xmitgcm, and users are recommended to use `open_mdsdataset` wherever possible (see *Reading MDS Data*).

Utility functions for reading MITgcm mds files (.meta / .data)

`xmitgcm.utils.`**`get_extra_metadata`**(*domain='llc'*, *nx=90*)
Return the extra_metadata dictionay for selected domains

`xmitgcm.utils.`**`parse_available_diagnostics`**(*fname*, *layers={}*)
Examine the available_diagnostics.log file and translate it into useful variable metadata.

Parameters **fname** : str or buffer

the path to the diagnostics file or a file buffer

layers : dict (optional)

dictionary mapping layers names to dimension sizes

Returns **all_diags** : a dictionary keyed by variable names with values

(coords, description, units)

`xmitgcm.utils.`**`parse_meta_file`**(*fname*)
Get the metadata as a dict out of the MITgcm mds .meta file.

Parameters **fname** : str

Path to the .meta file

Returns **flds** : dict

Metadata in dictionary form.

`xmitgcm.utils.`**`read_2D_chunks`**(*variable*, *file_metadata*, *use_mmap=False*, *use_dask=False*)
Return dask array for variable, from the file described by file_metadata, reading 2D chunks.

Parameters **variable** : string

name of the variable to read

file_metadata : dict

internal file_metadata for binary file

use_mmap : bool, optional

Whether to read the data using a numpy.memmap

use_dask : bool, optional

collect the data lazily or eagerly

Returns dask array for variable, with 2d (ny, nx) chunks

or numpy.ndarray or memmap, depending on input args

xmitgcm.utils.**read_3D_chunks**(*variable*, *file_metadata*, *use_mmap=False*, *use_dask=False*)

Return dask array for variable, from the file described by file_metadata, reading 3D chunks. Not suitable for llc data.

Parameters **variable** : string

name of the variable to read

file_metadata : dict

internal file_metadata for binary file

use_mmap : bool, optional

Whether to read the data using a numpy.memmap

use_dask : bool, optional

collect the data lazily or eagerly

Returns dask array for variable, with 3d (nz, ny, nx) chunks

or numpy.ndarray or memmap, depending on input args

xmitgcm.utils.**read_3d_llc_data**(*fname*, *nz*, *nx*, *dtype='>f8'*, *memmap=True*, *nrecs=1*, *method='smallchunks'*)

Read a three-dimensional LLC file using a custom dask graph.

Parameters **fname** : string

Path to the file on disk

nz : int

Number of vertical levels

nx : int

Size of each face side dimension

dtype : np.dtype, optional

Datatype of the data

memmap : bool, optional

Whether to read the data using np.memmap. Forced to be `False` for `method="smallchunks"`.

nrecs : int, optional

The number of records in a multi-record file

method : {"smallchunks", "bigchunks"}, optional

Which routine to use for reading raw LLC. "smallchunks" splits the file into a individual dask chunk of size (nx x nx) for each face of each level (i.e. the total number of chunks is 13 * nz). "bigchunks" loads the whole raw data file (either into memory or as a numpy.memmap), splits it into faces, and concatenates those faces together using `dask.array.concatenate`. The different methods will have different memory and i/o performance depending on the details of the system configuration.

Returns **data** : dask.array.Array

The data

`xmitgcm.utils.`**`read_all_variables`**(*variable_list*, *file_metadata*, *use_mmap=False*, *use_dask=False*, *chunks='3D'*)

Return a dictionary of dask arrays for variables in a MDS file

Parameters **variable_list** : list

list of MITgcm variables, from fldList in .meta

file_metadata : dict

internal metadata for binary file

use_mmap : bool, optional

Whether to read the data using a numpy.memmap

chunks : str, optional

Whether to read 2D (default) or 3D chunks 2D chunks are reading (x,y) levels and 3D chunks are reading the a (x,y,z) field

Returns

———-

list of data arrays (dask.array, numpy.ndarray or memmap)

corresponding to variables from given list in the file

described by file_metadata

`xmitgcm.utils.`**`read_mds`**(*fname*, *iternum=None*, *use_mmap=True*, *endian='>'*, *shape=None*, *dtype=None*, *use_dask=True*, *extra_metadata=None*, *chunks='3D'*, *llc=False*, *llc_method='smallchunks'*, *legacy=True*)

Read an MITgcm .meta / .data file pair

Parameters **fname** : str

The base name of the data file pair (without a .data or .meta suffix)

iternum : int, optional

The iteration number suffix

use_mmap : bool, optional

Whether to read the data using a numpy.memmap

endian : {'>', '<', '|'}, optional

Dndianness of the data

dtype : numpy.dtype, optional

Data type of the data (will be inferred from the .meta file by default)

shape : tuple, optional

Shape of the data (will be inferred from the .meta file by default)

use_dask : bool, optional

Whether wrap the reading of the raw data in a `dask.delayed` object

extra_metadata : dict, optional

Dictionary containing some extra metadata that will be appended to content of MITgcm meta file to create the file_metadata. This is needed for llc type configurations (global or regional). In this case the extra metadata used is of the form :

aste = {'has_faces': True, 'ny': 1350, 'nx': 270, 'ny_facets': [450,0,270,180,450], 'pad_before_y': [90,0,0,0,0], 'pad_after_y': [0,0,0,90,90], 'face_facets': [0, 0, 2, 3, 4, 4], 'facet_orders' : ['C', 'C', 'C', 'F', 'F'], 'face_offsets' : [0, 1, 0, 0, 0, 1], 'transpose_face' : [False, False, False,

True, True, True]}

llc90 = {'has_faces': True, 'ny': 13*90, 'nx': 90, 'ny_facets': [3*90, 3*90, 90, 3*90, 3*90], 'face_facets': [0, 0, 0, 1, 1, 1, 2, 3, 3, 3, 4, 4, 4], 'facet_orders': ['C', 'C', 'C', 'F', 'F'], 'face_offsets': [0, 1, 2, 0, 1, 2, 0, 0, 1, 2, 0, 1, 2], 'transpose_face' : [False, False, False,

False, False, False, False, True, True, True, True, True, True]}

llc grids have typically 5 rectangular facets and will be mapped onto N (=13 for llc, =6 for aste) square faces. Keys for the extra_metadata dictionary can be of different types and length:

- bool:

1. has_faces : True if domain is combination of connected grids

- list of len=nfacets:

1. **ny_facets** [number of points in y direction of each facet] (usually n * nx)
2. **pad_before_y (Regional configuration)** [pad data with N zeros] before array
3. **pad_after_y (Regional configuration)** [pad data with N zeros] after array
4. facet_order : row/column major order of this facet

- list of len=nfaces:

1. face_facets : facet of origin for this face
2. face_offsets : position of the face in the facet (0 = start)
3. transpose_face : transpose the data for this face

chunks : {'3D', '2D'}

Which routine to use for chunking data. '2D' splits the file into a individual dask chunk of size (nx x nx) for each face (if llc) of each record of each level. '3D' loads the whole raw data file (either into memory or as a numpy.memmap) and is not suitable for llc configurations. The different methods will have different memory and i/o performance depending on the details of the system configuration.

obsolete : llc and llc_methods, kept for testing

Returns **data** : dict

The keys correspond to the variable names of the different variables in the data file. The values are the data itself, either as an `numpy.ndarray`, `numpy.memmap`, or `dask.array.Array` depending on the options selected.

xmitgcm.utils.**read_raw_data**(*datafile*, *dtype*, *shape*, *use_mmap=False*, *offset=0*, *order='C'*, *partial_read=False*)

Read a raw binary file and shape it.

Parameters **datafile** : str

Path to a .data file

dtype : numpy.dtype

Data type of the data

shape : tuple

Shape of the data

use_memmap : bool, optional

Whether to read the data using a numpy.memmap

offset : int, optional

Offset (in bytes) to apply on read

order : str, optional

Row/Column Major = 'C' or 'F'

partial_read : bool, optional

If reading part of the file

Returns **data** : numpy.ndarray

The data (or a memmap to it)

CHAPTER 7

Development

7.1 Release History

7.1.1 v.0.2.2 (2018-07-18)

- Extend capabilities of read_raw_data (GH84)
- Fix the problem with testing type of prefix (GH83)
- Cast prefix to list if it isn't already one (GH79)
- Generalizes _get_all_iternums in order to handle compressed data (GH77)
- Extract version number from git tag (GH72)
- Adding .stickler.yml (GH70)
- Added functionality to read PTRtave files (GH63)
- Update examples.rst (GH65)
- fix time encoding (GH61)
- Fix llc chunking (GH60)
- Test refactor (GH54)
- Kpp added properly (GH55)
- Tests for ref_date issue (GH53)
- Add python 3.6 testing (GH52)
- Added layers axis attribute (GH47)

7.1.2 v.0.2.1 (2017-05-31)

- Fix to ensure that grid indices are always interger dtype.

- Fix to keep proper Comodo metadata when swapping dimensions.

7.1.3 v0.2.0 (2017-02-14)

This release contains the following feature enhancements:

- Files are not read until the data are accessed. This helps overcome a common "too many open files issue" (GH11).
- A workaround for missing `.meta` files (GH12).
- Option for a separate `grid_dir` in case it is different from `data_dir` (GH13).
- Refactor of the way LLC data is read which allows for more efficient chunking and lower memory usage (GH20)
- Bug fix related to the handling of *default_dtype* parameter (GH34). By Guillaume Sérazin.
- Support for older MITgcm versions that write a different lenght `drC` variable (GH8). By Liam Brannigan.
- Support for cartesian curvilinear grids. By Andrea Cimatoribus.
- Expanded and improved documentation.

Unless otherwise noted, all updates are by Ryan Abernathey.

7.1.4 v0.1.0 (2016-10-15)

Initial release.

7.2 Develpment Workflow

Anyone interested in helping to develop xmitgcm needs to create their own fork of our *git repository*. (Follow the github forking instructions. You will need a github account.)

Clone your fork on your local machine.

```
$ git clone git@github.com:USERNAME/xmitgcm
```

(In the above, replace USERNAME with your github user name.)

Then set your fork to track the upstream xmitgcm repo.

```
$ cd xmitgcm
$ git remote add upstream git://github.com/xgcm/xmitgcm.git
```

You will want to periodically sync your master branch with the upstream master.

```
$ git fetch upstream
$ git rebase upstream/master
```

Never make any commits on your local master branch. Instead open a feature branch for every new development task.

```
$ git checkout -b cool_new_feature
```

(Replace *cool_new_feature* with an appropriate description of your feature.) At this point you work on your new feature, using *git add* to add your changes. When your feature is complete and well tested, commit your changes

```
$ git commit -m 'did a bunch of great work'
```

and push your branch to github.

```
$ git push origin cool_new_feature
```

At this point, you go find your fork on github.com and create a pull request. Clearly describe what you have done in the comments. If your pull request fixes an issue or adds a useful new feature, the team will gladly merge it.

After your pull request is merged, you can switch back to the master branch, rebase, and delete your feature branch. You will find your new feature incorporated into xmitgcm.

```
$ git checkout master
$ git fetch upstream
$ git rebase upstream/master
$ git branch -d cool_new_feature
```

7.3 Virtual Environment

This is how to create a virtual environment into which to test-install xmitgcm, install it, check the version, and tear down the virtual environment.

```
$ conda create --yes -n test_env python=3.5 xarray dask numpy pytest future
$ source activate test_env
$ pip install xmitgcm
$ python -c 'import xmitgcm; print(xmitgcm.__version__);'
$ source deactivate
$ conda env remove --yes -n test_env
```

Bibliography

[R1] http://cfconventions.org/Data/cf-conventions/cf-conventions-1.7/build/ch04s04.html

Python Module Index

x

Index