DASK Training

Shaojun Ni

Why Use Dask

- EASY TO PARALLEL
- SMOOTH HPC INTEGRATION
- HANDLE DATA LARGER THAN RAM
- NO COST

PARALLEL PROCESSING

Parallel in DASK is easy by using 2/3 keywords

- delayed (method decoration)

  Make the code be able to run parallel
- compute

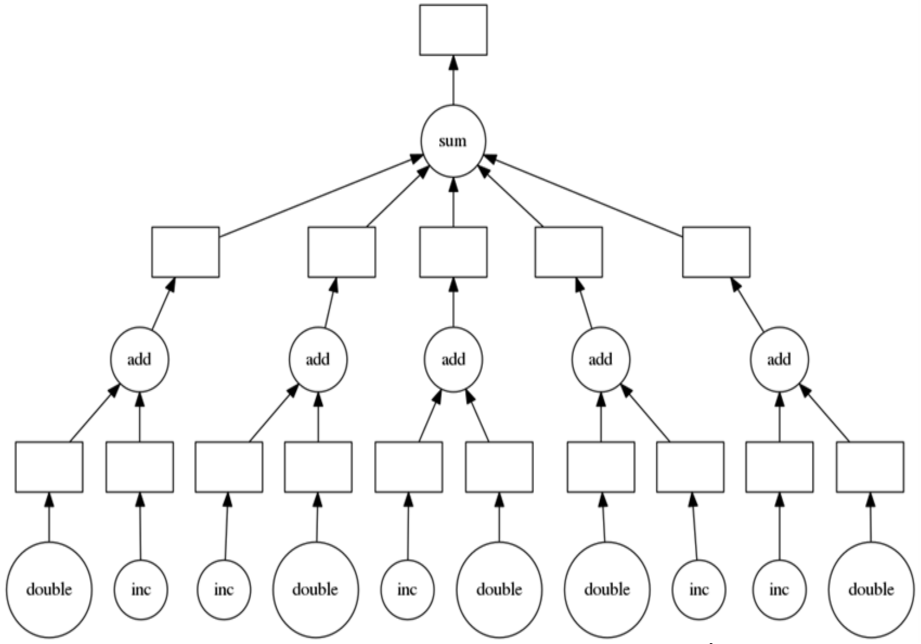
  Start delayed tasks parallelly
- visualize

  Create graph to visualize the parallel

DASK Array, Bag, Dataframe automatically parallel the computation by divide the big data into smaller pieces for parallel.

Parallel Code II:

- dask.delayed(func1(a)) is another way to make function parallel if you don’t want to use @delayed decoration.
- compute(scheduler=‘processses’) tells the code to run parallel in processes. By default it is run parallel in threads.
- dask.config.set(pool=Pool(5)) limits the number of processes.



- Use jupyter notebook to exercise parallel. Code is at /glb/data/cdis_projects/users/ussnis/attribute_engine/src/Dask/test/test_parallel.py

COMPUTATION

In DASK the computation is done at the block level. We use map_blocks function in the attribute engine.

dask.array.map_blocks(x, func, *args, dtype, chuncks)

Map a function across all blocks of a dask array.

x: dask array data

func: the attribute compute function

dtype: The data type of output array.

chunks: Chunk shape of resulting blocks if the function does not preserve shape. If not provided, the resulting array is assumed to have the same block structure as the first input array.

We set the chunk size to be the same as zarr file chunk size.

OVERLAP COMPUTATION

Some array operations require communication of borders between neighboring blocks. Dask Array supports these operations by creating a new array where each block is slightly expanded by the borders of its neighbors. This feature is extremely useful when the computation requires cross multiple traces.

map_overlap(x, func, depth[, boundary=None, trim=True])

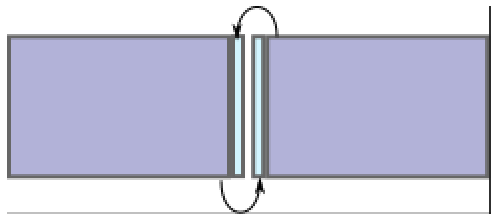
x: the dask array

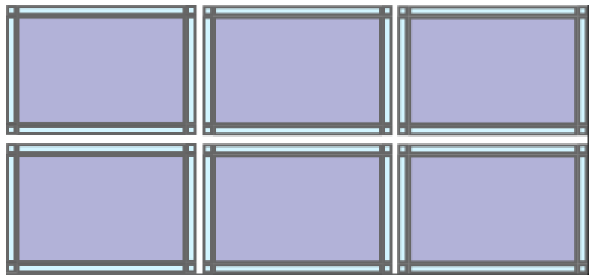
func: compute function

depth: int, tuple, or dict

boundary: How to handle the boundaries. Values include ‘reflect’, ‘periodic’, ‘nearest’, ‘none’, or any constant value like 0

trim: bool, Whether or not to trim depth elements from each block after calling the map function.





OVERLAP COMPUTATION

Depth:

It defines how much extra data needs to be added into the current block to compute. The bigger number means the array gets larger. The depth can be defined in each dimension of the array. Ex: (0:1, 1:2, 2:3)

Boundary:

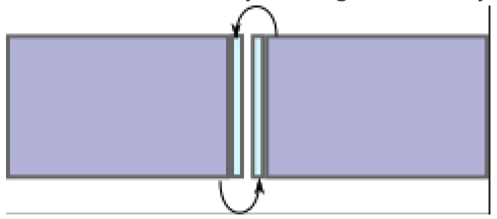
It defines how to expend the array at the end of array.

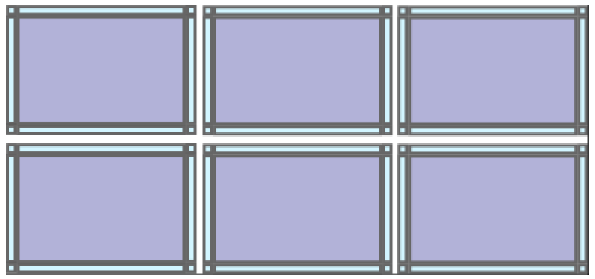
Reflect: reflect each border outwards

Periodic: wrap borders around to the other side

Constant Number: just constant number

Extend each block by trading thin nearby slices between arrays:





- *Use jupyter notebook to exercise the map function.*

HPC Integration

DASK has built-in library to support HPC. Ex: LSF, use LSFCluster to allocate HPC nodes. DaskScheduler will send the request to the HPC nodes.

The code stays the same for HPC, only needs to add some infrastructure code to enable HPC. Following code is to create Dask LSFCluster.

```
from dask.distributed import Client
from dask_jobqueue import LSFCluster

  job_extra_setting = ''
  if use_gpu:
    job_extra_setting = ['-gpu "num=2"']
  cluster = LSFCluster(queue=queue_name, project=hpc_project, walltime='{0}:00'.format(hpc_time),
                       cores=hpc_core, processes=processes_per_node, local_directory='dask-worker-space',
                       memory='{0}GB'.format(hpc_memory), job_extra=job_extra_setting, log_directory='scheduler_log',
                       dashboard_address=':{0}'.format(dashboard_port))
  cluster.scale(number_of_nodes * processes_per_node)
```

CLIENT

Client sends request to the scheduler. When create a client, the scheduler address has to provide.

```
client = Client()  // no HPC
client = Client(cluster.scheduler_address, timeout=60)  // HPC mode.
```

x = client.submit(inc, 10)

DASK collection uses the distributed system by default, don't have to use client to submit the request to workers.

```
dask.array.map_overlap(data, func....)
```

Regular functions needs client to submit the request.

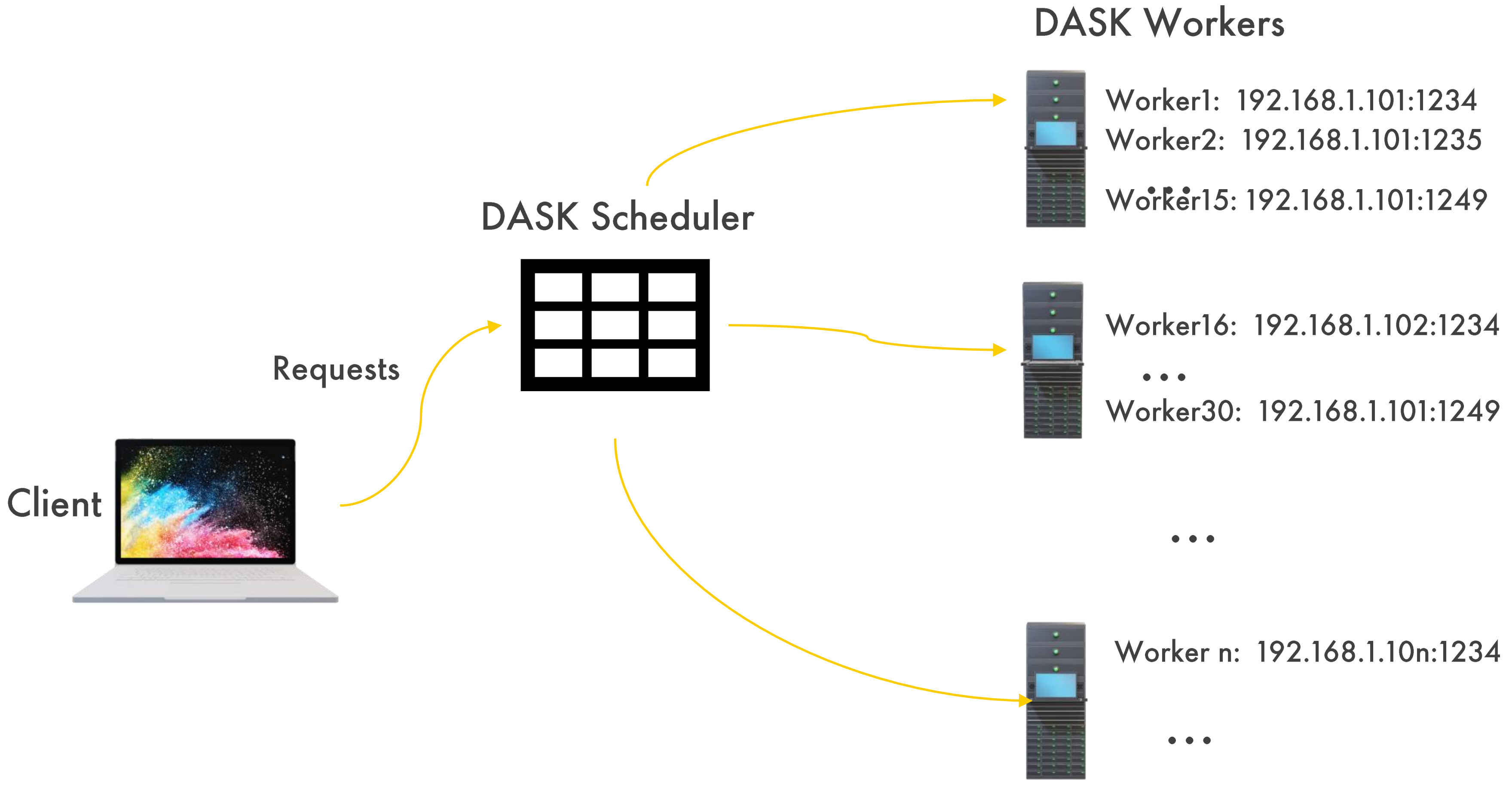
```
def inc(x):
  return x + 1
x = client.submit(inc, 10)
```

Dask preprocessor, kmean support.

```
import dask_ml.preprocessing as dl
import dask_ml.cluster as dl_k
```

Reference /glb/data/CDIS5/users/ussnis/ml

DASK WITH HPC


DASK Workers
Worker1: 192.168.1.101:1234
Worker2: 192.168.1.101:1235
• • •
Worker15: 192.168.1.101:1249
DASK Scheduler
Worker16: 192.168.1.102:1234
• • •
Worker30: 192.168.1.101:1249
Requests
Client
• • •
Worker n: 192.168.1.10n:1234
• • •

ZARR FILE

- Zarr is a python package providing an implementation of chunked, compressed, n-dimensional arrays.
- Zarr supports chunking of arrays along multiple dimensions, enabling good performance for multiple data access patterns.
- ZARR can be used in parallel computations, supporting concurrent reads and writes in either a multi-threaded or multi-process context.
- Unlike hdf5, zarr does not have GIL. It makes parallel/distribution computing very efficient.
- ZARR support multiple data store, such as directory, zip, Amazon S3…

Example:

```
ls seismic_0.zarr -al
drwxr-sr-x. 2 ussnis g_cdis00      209 Aug 23 15:57 .
drwxr-sr-x. 5 ussnis g_cdis00       96 Aug 23 15:57 ..
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 0.0.0
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 0.0.1
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 0.1.0
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 0.1.1
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 1.0.0
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 1.0.1
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 1.1.0
-rw——. 1 ussnis g_cdis00 36000000 Aug 23 15:57 1.1.1
-rw——. 1 ussnis g_cdis00      251 Aug 23 15:57 .zarray
```

Handle Large Data

Dask introduces 3 parallel collections that are able to store data that is larger than ram, namely Dataframes, Bags and Arrays. Ex:

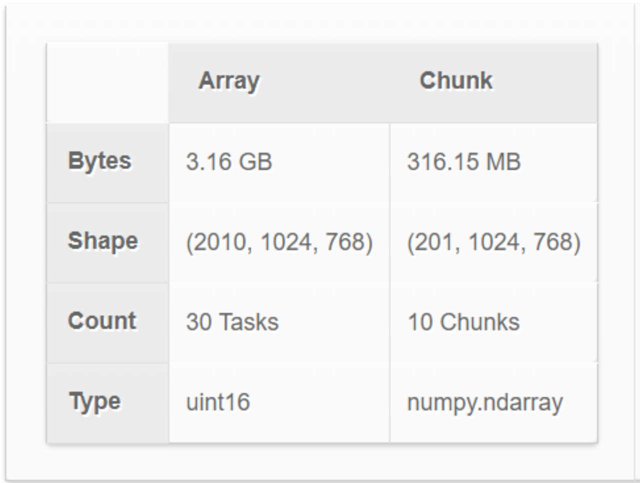
```
import dask.dataframe as dd
df = dd.read_csv('logs/2018-*.*.csv', parse_dates=['timestamp'])
df.groupby(df.timestamp.dt.hour).value.mean().compute()
```

DASK prefer use zarr file to store big data, therefore the data can be parallelly computed by HPC nodes efficiently. DASK Array provides to_zarr(), from_zarr() function.

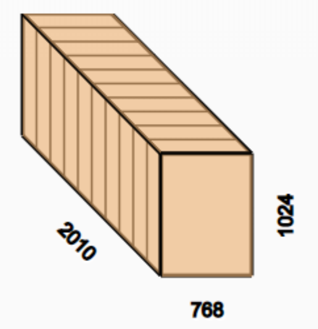
If the source data is bigger than memory, lazy loading can be used to convert big than memory data into zarr format. During the conversion, the array computations would work fine, and would run in low memory, but we'll save actual computation for future posts.

```
import dask
import dask.array as da

img_arrays = [dask.delayed(imageio.imread)(fn) for fn in filenames]
lazy_arrays = [da.from_delayed(x, shape=sample.shape, dtype=sample.dtype)
            for x in img_arrays]
x = da.concatenate(lazy_arrays)
x.to_zarr(filename).compute()
```



	Array	Chunk
Bytes	3.16 GB	316.15 MB
Shape	(2010, 1024, 768)	(201, 1024, 768)
Count	30 Tasks	10 Chunks
Type	uint16	numpy.ndarray



Current DL Project

Amplitude Extraction.ipynb, run from hougdc interactive node.
/glb/data/cdis_projects/users/ussnis/ml

Prestack
/glb/data/cdis_projects/users/ussnis/ml/prestack, can run from hougdc interactive node.

Fault Crawler
/glb/data/cdis_projects/users/ussnis/crawler3.6/geocrawler, run from cdis interactive node.

Wbsddl
/glb/data/cdis_projects/users/ussnis/ml/geocrawler_wbsdDL-master

Questions and Answers

Q&A