Big Data in Parallel

Originally by Vladimir Ulman, 1st Apr 2020

Github repository: https://github.com/fiji-hpc/hpc-datastore

The DataStore

- Titled (inaccurately) as the Dataset in the PDF
- Permanent storage with read-write capabilities, collection of **Datasets** (e.g. SPIM video)
- Stores data in one format, but smartly
    - In blocks/chunks/ROIs (synonyms here), and
    - in copies at different spatial resolution, and
    - in versions/revisions (synonyms)
- **DataStore** has
    - one **DatasetsRegisterService**, runs 24/7
    - possibly many, on-demand started **DatasetServers**

The whole layout of dataset chunks (here called blocks) follows essentially the N5 philosophy: https://github.com/saalfeldlab/n5/#file-system-specification
It is assumed that we will have N5 on the backend side, because it offers multiple simultaneous access (at the level of filesystem requests -> can be even processes from different OSes) and flexibility to replace pieces of it (e.g. own compression schema). We can even plugin our own bytes storing backend (e.g., if filesystem architecture changes). N5 can hold sparsely filled data.

Dataset

- Stores multiple versions of image data, e.g., data acquired from microscope, instance segmentation images; each version is stored as a (possibly spatially sparse) separate N5
- For the BigDataOnHPC idea: One processes a Dataset at version v (in ro regime), the outcome is stored into the same Dataset (in wo regime) at version v+1
- Stores image data at multiple resolutions, that is in multiple smaller and smaller copies; this is an optional, though strongly suggested feature of a dataset that utilizes (mainly) smoother viewing of the image data

- Dataset is referenced with a unique but arbitrary string, e.g., UUID or Java package naming scheme (projects can choose own naming schemes)

- Dataset is stored in multiple copies, each at different resolution level Rx,Ry,Rz
- Resolution levels are defined by user (a default suggestion will be offered in client GUI)
- R*i* defines how many highest-res (full res.) pixels are squashed into this-level single pixel along axis *i*, it is integer downscale factor per axis of the original geometry

- Rx,Ry,Rz is a triplet, must always be used as a whole; resolution levels is a list of such triplets (it is not any combination from available separate Rx, Ry, Rz)

- Datasets are versioned, version is 0-based integer, greater version means newer

- A dataset at some version is stored alone in its own N5 (or any other backend), however all (sub)resolutions of the dataset are stored together inside this N5 (or any other backend), that said, versions live under different paths, resolutions under the same path as its full res data
- Every dataset's N5 is constructed so that BigDataViewer can read it
- One can remove intermediate versions, or move old versions to slow (but large) drivers; one can inspect versions directly with BDV if one has direct access to the N5 files.
- Every version, since it maps to own full N5, can have own metadata (as part of N5), including, e.g., time-stamp when this version was created and the author (who uploaded)

- Dataset is completely addressed as UUID/Rx/Ry/Rz/V ...as three params:
  UUID - string that identifies the particular Dataset
  (Rx,Ry,Rz) - some resolution level from the available ones
  V - version of the data from the available ones

- Dataset at resolution 1,1,1 (the original resolution) is always 6 dimensional scalar tensor of size [X,Y,Z,T,Ch,Ang] (Ch = channel, Ang = angle), fixed semantics to the dimensions
- Scalars are of the same types that N5 natively supports
  (only *Byte-multiple* types, e.g. gray8, gray16, long, float)
- Dataset at Rx,Ry,Rz is tensor of [*down*(X/Rx), *down*(Y/Ry), *down*(Z/Rz), T,Ch,Ang]
  where *down*(x) is the largest integer not larger than x (floor-rounding), and
  where *up*(x) is the smallest integer not smaller than x (ceil-rounding)

- Dataset at Rx,Ry,Rz is stored as multiple blocks,
  each block's geometry/size is [Bx,By,Bz,1,1,1], "right edge" blocks may be smaller
  (which is why the block size (3x 4 B little-endian integers) is transferred at the beginning of every transferred block's data)
- Must hold: Bx*By*Bz*voxelSize < 2^30 B = 1 GB
- Block size (and thus their number) is a function of resolution level
- Block size should be intelligently set
  - not too small = not too many blocks (comm. overhead)
  - not exceedingly large = not to reach HW limits during processing
  - lower resolution copies are expected to have fewer blocks because the block size is not really expected to change between res levels
- Dataset [X,Y,Z,T,Ch,Ang] at Rx,Ry,Rz res. level is made of
  up(*down*(X/Rx)/Bx) * *up*(*down*(Y/Ry)/By) * *up*(*down*(Z/Rz)/Bz) *T*Ch*Ang blocks
- Blocks are numbered as 0-based 6-element vector

- Block(s) are finally addressed as UUID/Rx/Ry/Rz/V[/x/y/z/t/c/a]+
- Within DataStore path to a Dataset
- Within Dataset path to a block = N5 path to a chunk,
  [AB,]+ denotes sequence AB,AB,...AB, that lists at least one AB
- Cannot manipulate anything larger than one block, always exactly block(s)
  (note again, 12 B block size is prefetched before the actual block data)
- Can request multiple blocks in one request

Terminology recap:

- **Dataset dimension** = tells how many voxels are there along every axis in the 1/1/1 (full) Dataset resolution, there are always 6 axes in every dataset, axes are X,Y,Z,T,Ch,Ang
- **Dataset dimensions** = Dataset dimension, treated as synonyms
- **Resolution level** Rx/Ry/Rz = defines downsampling rates for X,Y,Z axes **from** the same dataset at **full resolution** (not from previous resolution); it is absolute (not relative)
- **Dataset at** Rx/Ry/Rz **resolution** = Downsampled version of the same dataset at full resolution
- **Voxel resolution** = "physical size" of a voxel in the given dimension X,Y,Z
- **Time/Angle/Channel resolution** = “physical distance/separation” in time, in viewing angle, and in displayed content, respectively

DatasetsRegisterService

- Runs 24/7
- Maintains a list of known Datasets (list of UUIDs), and for each
  - it has some common metadata (additional descriptive note)
    (content’s format not defined, can be even empty; JSON;
    every N5/Dataset has also own optional detailed metadata field)
  - Original dimension [X,Y,Z,T,Ch,Ang] (size in each axis) at full res. and voxelType
  - List of available versions V, for each path to own N5
  - List of available resolution levels (Rx,Ry,Rz), for each
    - Block size: Bx,By,Bz
    - Value (scalar) for each dimension per 1 pixel, Unit (string) for pixels, timepoints, channels, angles

- Internally, it calls for a database with
  - A table DatasetInfo with the columns (Dataset name and common metadata):
    - DID (int shortcut/ID to the dataset’s name) - primary key
    - Dataset’s identifying string, e.g., UUID or Java naming scheme
    - Dataset’s common metadata
    - voxelType (e.g. uint128)
    - X,Y,Z,T,Ch,Ang - full res original size
    - 4x Value, Unit (four pairs of columns)

    - a table DatasetResolutions with the columns:
        - DID (to ref Dataset in the table above); DID = Dataset ID
        - Rx,Ry,Rz (three columns) - primary key together with DID column
        - Bx,By,Bz (three columns)
    - a table DatasetVersions with the columns:
        - DID, V (two columns) - primary key
        - Filesystem path to the data in N5
- Needs connection to a (remote, shared) database
- Needs to be able to access N5 files on (remote, shared) filesystem

Offers the following services:

- Query UUID, replies with *select all (minus DID) from DatasetInfo where UUID*=UUID
- Get/Set the common metadata for UUID

- Delete UUID, deletes all for that UUID, deletes this dataset completely
- Delete UUID/[V]+, deletes all listed Vs
- *Deleting or adding particular res. level is currently not supported*

- Create empty UUID,X,Y,Z,T,Ch,Ang,6xDouble,4xString [,Rx,Ry,Rz,Bx,By,Bz]+
  at least one resolution level must be given with metadata, not necessarily Rx=Ry=Rz=1, and creates corresponding N5 data where all chunks are initiated with "*data-not-available*" status, params Rx (also Ry,Rz) should appear as non-decreasing sequence, returns a status, e.g. "OK", "some error report"
- Example of Create:
  1000,1000,1,10,3,5, //initial dimensions: x,y,z,T,Ch,Ang
  0.4, 0.4, 1, 0,0,0, "um","min","null","null", //units: 0.4,0.4,1 um, 0 min, 0 null, 0 null
  1,1,1,50,50,1, //first res level
  2,2,1,50,50,1 //next res level (in which x,y is 2x downsampled)
  Passed as JSON Object:

```
{
        "voxelType":"uint64",
        "dimensions":["1000","1000","1"],
        "timepoints":"10",
        "channels":"3",
        "angles":"5",
        "voxelUnit": "um",
        "voxelResolution": ["0.4", "0.4", "1"],
        "timepointResolution": {"value":"1","unit":"min"},
        "channelResolution": {"value":"0","unit":null},
        "angleResolution": {"value":"0","unit":null},
        "storageCompression": "raw",
        "resolutionLevels":
                [
                        {"resolutions":["1","1","1"],"blockDimensions":["50","50","1"]},
                        {"resolutions":["2","2","1"],"blockDimensions":["50","50","1"]}
                ],
        "versions": ["0","1","5"]
}
```

- Resolution is a pair: double Value and string Unit,
  e.g., 2.4 “microns”, 3.0 “mins”, 180 “seconds”, 567 “nm”, 567 “wave length”, 20 “deg”

- Start serving requests for datasets/*UUID/Rx/Ry/Rz/V/read[?timeout=<value>]* with TIMEOUT or datasets/*UUID/Rx/Ry/Rz/write[?timeout=<value>]* with integer TIMEOUT in minutes
  - Has to find unoccupied IP:PORT
  - Spawns (fire-and-forget) a DatasetServer at that **IP:PORT** for reading-only or for writing-only to serve the corresponding N5 (from the table)
  - When writing, supplied **V** can be keyword “**new**”, it creates a next new version and spawns a server to serve it, tells back the new number (int)
  - When reading, supplied **V** can be keyword “**latest**”, it spawns a server to serve a dataset at the latest version, tells back the number (int)
  - When reading, supplied **V** can be keyword “**mixedLatest**”, it spawns a special server that opens all available versions of the dataset (all N5s) and serves the appropriate blocks from the highest version where the asked block exists
  - Returns with http redirect to http://IP:PORT/
    - Client updates the URL to fetch the data:
      http://IP:PORT/UUID/Rx/Ry/Rz/V/mode[read|write]
    - Client consequently attempts to connect the DatasetServer on that URL
    - DatasetServer on that URL confirms the connection back to the client (this way, client is sure it could reach the DatasetServer)
- This is meant for processing Dataset at particular resolution level (one could read “latest” and write “new” versions), or update particular existing version of given res level
- If there’s no traffic within TIMEOUT, server closes automatically

- Start serving requests for UUID/1/1/1[/Rx/Ry/Rz]+*/write[?timeout=<value>]*
  - Checks that UUID exists at all specified resolutions and removes non-zero versions from them (only one version “0” will remain)
  - Spawns (fire-and-forget) a special DatasetServer with a DatasetBuilder that downscales simultaneously
  - Redirects to http://IP:PORT/
- This is meant to (re)build a Dataset and all its resolution levels, you upload 1/1/1 data, and server on-the-fly (via the DatasetBuilder) creates and (over)writes data at the other listed/requested resolutions
- We prefer to have all res levels of a Dataset at same version V, that’s why we reset it
- If there’s no traffic within TIMEOUT, server closes automatically

- Request server-side rebuild of resolution levels for UUID/1/1/1[/Rx/Ry/Rz]+/rebuild
  - Checks that UUID exists at all specified resolutions and removes non-zero versions from them (only one version “0” will remain)
  - Spawns (fire-and-forget) a special DatasetBuilder, but no server

    - DatasetBuilder reads 1/1/1 data and creates and (over)writes data at the other listed/requested resolutions

- Explicit stop serving requests at IP:PORT

DatasetServer (and **DatasetBuilder**)

- It is started and closed from DatasetsRegisterService (can be RPC'ed!),
  Closed when no client comes after TIMEOUT minutes since the last one
- Needs to be able to access the N5 files on filesystem
- Can be more DatasetServers (PORT:IP) serving the same N5
  (filesystem resolves data races)
- Back-end N5+filestorage with chunks gzipped (N5 can do all of that)
    - This suggest this will be Java application
- Blocks fit completely easily into RAM, just read&send it (get&write it)
- Front-end implements manyClients-to-oneServer communication:
    - Client request is fully solved until next client is processed
    - write-only regime: Client sends /x/y/z/t/c/a and data
    - read-only regime: Client sends /x/y/z/t/c/a and gets
      the wanted data, or "*data-not-available*" message
    - Chunk size is known here to all parties
    - Data transferred is already the gzipped chunk, the chunk format is explained in the N5 specs: File-system specification, point 9
    - Should be implemented such that we can have clients in any programming language! This affect mainly the communication network protocols
    - read/write both in push/pull flavours… differ in who initiates the connection to fight NAT; we provide no solution if both parties are behind NAT - not our problem here

Client

- Gets from user pointer on DatasetsRegisterService
- Gets inUUID/Rx/Ry/Rz from user (or inUUID alone and offers choices…)
- Gets OP to be executed
- Gets ParallelParadigm instance..

- Requests to create outUUID/Rx/Ry/Rz and requests M DatasetServers for writing it
- Requests N DatasetServers for reading inUUID/Rx/Ry/Rz
- ParallelParadigm starts spawning Jobs
    - Job is to populate outUUID/Rx/Ry/Rz/x/y/z/t/c/a using round-robin-from-M for writing, and round-robin-from-N for reading whatever needs to be read to complete the job
    - Job can also be to process the full frame/image, i.e. not per-chunks here,
      In this case, the job is given only outUUID/Rx/Ry/Rz/t/c/a - **fullFrameForm** (**FFF**)

Executor

- For the OP, we have to provide our own mean to reach the data:
  - Either **API for OP** to use (OP is pimped for this story, OP is aware)
  - Path to input and output TIFF files (in Ramdisk?) (OP is unaware)
    (we can build larger TIFF to have the convolution computed, but
    extract only the relevant portion of it for upload of the result;
    we can collect all chunks and build the full frame image if we are given FFF)
  - If the reading of the inUUID/Rx-Ry-Rz/x/y/z/t/c/a finds "*data-not-available*", OP is stopped and nothing is written to the corresponding outUUID
    (aware OP detects this state via API, unaware OP is not started at all)
- Monitoring of data availability and potential further notifications will be done in Executor. It is out of the scope of the Dataset managing ensemble. (You are probably watching for new data only when you know that you have started their modifications, its built up somewhere - so it is relevant to the Executor session.)

FAQ:

Q: How to upload the vanila data?
A: Anything that can end up as Fiji's Dataset (with axes that have defined semantics) can be uploaded to the DatasetBuilder, we have to provide code for this.

Q: Do I need to upload the full image? What if there is noisy nothing in the corners?
A: You either upload the full dataset and design OP to detect it and to save anything,
or detect this already before uploading the relevant chunk and just don't upload it.

Q: Imglib2 anywhere in here?
A: Shouldn't be difficult to have DiskCachedCellImg where the disk loading/saving would actually use the API to read/write chunks to the Dataset; the CellImg needs to be created with the same cell geometry as is in the Dataset, which is easily possible. Then you can BDV it, or just do anything that is already imglib2. This is the **API for OP**.
Example:
https://github.com/tpietzsch/n5-rest/blob/master/src/main/java/tpietzsch/n5/rest/N5RestServer.java
http://undertow.io/

Q: Who will implement it?
A: Jan + Vlado + maybe some students (for satellite topics...like the DB, authorization, authentication)