# MeerKAT Online Data Storage

*CALIM 2010*

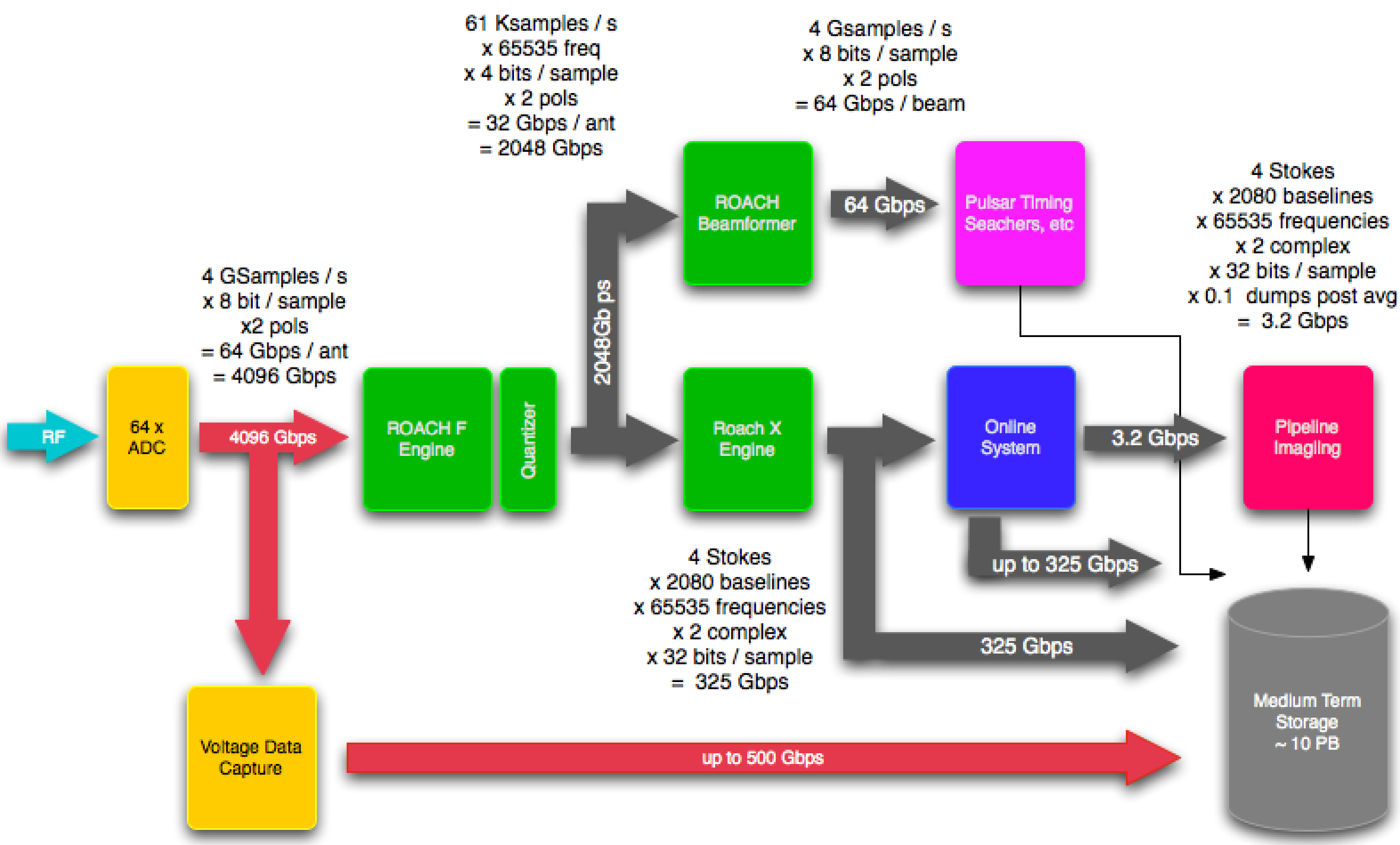
SKA SOUTH AFRICA
SQUARE KILOMETRE ARRAY

*Thomas Bennett*

# Overview of Current Effort

- Experimenting with HDF5 and CASA Tables Python tools.
- Understand how efficient they are by doing some comparisons based on speed and usability.
- Getting data to disk: start using and understanding underlying tools – such as MPI-IO and parallel file systems.
- Need to make sure that we can meet the MeerKAT requirements for the online data store.
- KAT-7 telescope is a good opportunity to test out some of these technologies while data rates are low.

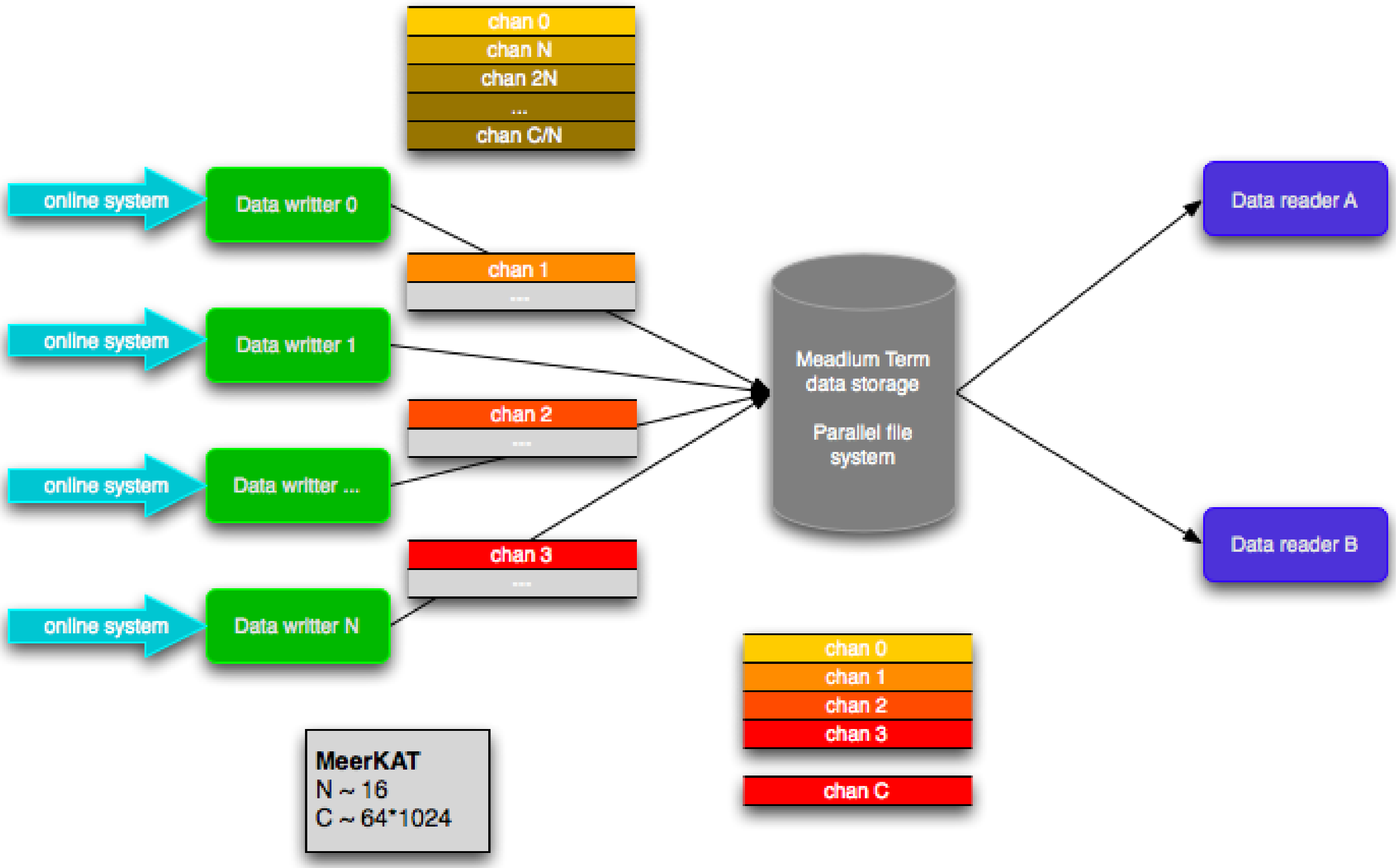
# MeerKAT Data Rates



# Online Storage Data Rates

- Expected data rates to online data storage:
  - 65536 frequency channels
  - 2080 baselines (incl auto correlation data)
  - 4 Stokes
  - complex data type (8 bytes)
  - ~ 4160 MB / correlator dump
- Predicted data storage requirements:
  - 10 PB / year for the intermediate data product from the online system

# MeerKAT Online Data Storage



# Parallel HDF5

- First cut of an implementation of a parallel write to an HDF5 dataset, written in C.
- This implementation has been tested on a basic cluster, consisting of 2 nodes which which write data to a Lustre parallel file system.
- Although this has been a good learning experience, there is still a lot that needs to be understood in terms of 'tuning' at the software layers.

# Parallel HDF5

Example code:

```
plist_id = H5Pcreate(H5P_DATASET_XFER);

H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

status = H5Dwrite(dset_id, corr_id, memspace, filespace,
  plist_id, data);

H5Pclose(plist_id);
```

# Python Tools: Overview

- Python implementations investigated so far:
    - pyrap python wrapper for casacore
    - h5py  python wrapper for hdf5 (pytables to follow)
    - netCDF4 python wrapper for netCDF4
- Since most of the heavy lifting should be done in wrapped libraries, python overhead should be minimal.

# Python Tools: Examples

## Accessing data from h5py high level API:

```
dset[0:ts, chan, 0:blines]
```

## Accessing data from h5py low level API:

```
dataspace.select_hyperslab(start=(0,chan,0,),
 count=(ts,1,blines,))

memspace = h5s.create_simple((ts,blines))

hd = np.empty((ts,blines,), dtype=np.complex64)

dset.id.read(memspace, dataspace, hd, tid)
```

# Python Tools: Test Descriptions

- Small row access (~ 700 MB): all frequency channels and all baselines for one time stamp (contiguous read).
- Large row access (~ 6 GB): all frequency channels and all baselines for one time stamp (contiguous read).
- Column access (~ 80 MB): 1 freq channel and all baselines for all time stamps (strided read)

# Python Tools: Results

| | raw | h5py high API | h5py low API |
|---|---|---|---|
| small contiguous data read | 215 - 250 MBps | 28 MBps | 150 MBps |
| large contiguous data read | 215 - 250 MBps | 24 MBps | 155 MBps |
| column access | 35 - 140 MBps | 0.8 MBps | 2 MBps |

# High Level Tools: Observations

- When opening HDF5 data group in a file of significant size can take up to 10 minutes.
- High level numpy slicing interface to h5py data slow and unoptimised for large data sets. Rather use low lever interface.
- None of the Python HDF5 interfaces currently support the MPI-IO driver.
- HDF5 - chunk size set to 8 bytes (size of np.complex64)

# Future Work

- High level tools:
  - Compare pyrap and h5py. Also compare with C level implementations.
  - Investigate pytables HDF5 implementation and compare to h5py.
- Lustre scaling tests on CHPC Sun cluster attached to a Lustre file system and performance tuning – HDF5/MPI-IO/Lustre
- Continue communications open with ASTRON people involved in the data storage and archiving.
- Keep an eye on pNFS as it matures.