# MPI Parallelization



MPI is a world-standard for large-scale parallel computing, and is supported by every major academic computer batch system. It allows for parallelization across multiple nodes, and also provides tools for gathering the results from different CPUs together. It not only allows you to add more CPU power to a problem, but can also be used to add:

- memory (by distributing a large memory-bound problem over multiple nodes)
- I/O (by allowing multiple network connections between data senders/receivers)

The recommended simplest way of running parallel analysis is to use the "MPIDataSource" pattern. This allows you to write code as if it was running only on one processor and store **small** per-event information (numbers and small arrays) as well as "end of run" summary data. This data can optionally be saved to a small HDF5 file, which can be copied, for example, to a laptop computer for analysis with any software that can read that format. This script can be found in /sdf/group/lcls/ds/ana/tutorials/psana1_examples/mpiDataSource.py

This script can be run in **real-time** while data is being taken, and will typically complete a few minutes after the run ends. **NOTE**: when running in parallel, the standard python "break" statement can cause hangs. Use the "break_after" command here to terminate data processing early.

```
from psana import *

dsource = MPIDataSource('exp=xpptut15:run=54:smd')
cspaddet = Detector('cspad')
smldata = dsource.small_data('run54.h5',gather_interval=100)

dsource.break_after(3) # stop iteration after 3 events (break statements do not work reliably with
MPIDataSource).
partial_run_sum = None
for nevt,evt in enumerate(dsource.events()):
    calib = cspaddet.calib(evt)
    if calib is None:
        continue
    cspad_sum = calib.sum()      # number
    cspad_roi = calib[0][0][3:5] # array
    if partial_run_sum is None:
        partial_run_sum = cspad_roi
    else:
        partial_run_sum += cspad_roi

   # save per-event data
    smldata.event(cspad_sum=cspad_sum,cspad_roi=cspad_roi)

# get (optional) "summary" data
run_sum = smldata.sum(partial_run_sum)
# save HDF5 file, including summary data
smldata.save(run_sum=run_sum)
```

"smd" mode (given to the MPIDataSource object) stands for "small data". It tells the software to use an identical copy of the full xtc data, but with all the large data (default >1kB) removed. Objects like camera images are replaced with a small "pointer" to the big data in the full xtc file. This allows all CPU cores to read through all the small-data quickly, and psana only fetches the large data when requested by the user with python methods like det.calib(evt) and det.image(evt). This is the mechanism used to allow for real-time MPI parallelization.

Run the script on 2 cores with this command to produce a "run54.h5" file:

```
mpirun -n 2 python mpiDataSource.py
```

It is also possible to specify a hierarchy in the output hdf5 file by passing a hierarchy of dictionaries/values to smldata.event. This would produce hdf5 groups "cspad/calib", "cspad/raw", and "my_other_data".

```
calib = det.calib(evt)
raw = det.raw(evt)
d = {'cspad' : {'calib': calib, 'raw': raw}, my_other_data = 3}
smldata.event(d)
```

In addition to running offline, these parallel scripts can be run in real time while the data is being taken and can complete within a few minutes of the end of the run (you can see how to submit MPI psana-python batch jobs here). Note that this interface does not currently work with the shared-memory analysis mode.

This software automatically saves per-event commonly-used information to the HDF5 file:

- The timestamps of each event, which can be used to quickly "jump" to the big data (e.g. camera images) of interesting events using small python scripts like this.
- All timing system "EVR event codes". For example, these can be used to tell whether or not a pump-laser was on or off, or a shutter was open or closed on a particular event
- FEE gas detector information
- EBEAM per-shot information from the accelerator (e.g. photon energy)
- Phase-cavity detector timing information

It is important to emphasize that this code is optimized for producing **SMALL** HDF5 files. For example, it will not run quickly if you save large images for every event. This may also cause the machines to run out of memory.

This pattern does not provide a solution for all possible LCLS analyses, so it is also possible to call MPI directly from python in these advanced examples.

## Missing Detectors: How Not To Crash

The default behavior of the Detector class is to crash when a requested Detector is not present (so users will be alerted if they make a typo, for example). But this is often not desired behavior during production running when Detectors are being added/removed from the data stream on a run-to-run basis. To change that behavior use a constructor like Detector('MyDetectorName',accept_missing=True). With this flag, all methods of the Detector will return None, just as it would if it was missing in every event.

## Accessing Data While Running

If one wishes to examine the data gathered while the analysis is in progress, users can register a "monitor" function will will be called every time data is gathered by all the cores (set with the "gather_interval" parameter in the above example). The monitor function will be passed a dictionary of all gathered values, and can be registered like this:

```
def my_monitor(results):
    # process results dictionary here

smldata.add_monitor_function(my_monitor)
```

Note that after each gather interval the data is removed from memory, so it is the user's responsibility to "remember" any data from previous callbacks that they want to keep (be careful not to use up all machine memory).

## Ragged Arrays

MPIDataSource supports "ragged" 1D arrays (also called variable-length, or vlen, arrays). An example would be an array of photon energies (or positions) whose length changed for every LCLS shot. If you have such an array you must start the HDF5 dataset name with the string "ragged_".

An alternative to ragged arrays are "variable" arrays. These are not limited to 1D, but only first dimension is variable, and the other dimensions must be fixed sizes. An HDF5 dataset name that starts with the string "var_" will generate a variable length array. A separate integer array ending with "_len" will be automatically generated with the number of elements of the variable array that belong to each event. (When reading such a file, a running count of the values from the "_len" array must be kept to locate the next event's data in this dataset.)

## Batch Job Submission

Examples showing how to submit SLURM batch jobs can be found here: Submitting SLURM Batch Jobs

# Saving Larger Data to HDF5 ("Translation")

The MPIDataSource pattern can be used to "translate" data from xtc to hdf5. It offers the following features:

- users can choose what data they want to store in HDF5 (e.g. raw image data, calibrated)
- users can use python algorithms (e.g. saving only part of a camera Image) to reduce the output data volume which dramatically speeds up translation. It also potentially allows the hdf5 files to be moved to a laptop for further analysis.
- can be run in parallel on many cores
- datasets are guaranteed to be "aligned"

**TIP**: If you save larger data to HDF5 (not recommended, for performance/space reasons) be sure to set the gather_interval parameter to 1 in order to avoid using up all machine memory.

## References

- Batch Nodes And Queues
- Detector documentation