CCTBX tools:
I. Parallelizing Python code
II. Analysis of unmerged intensities

Nathaniel Echols

DIALS workshop 3, February 2013

http://cci.lbl.gov/~nat/slides/dials_feb_2013.pdf

Parallelization methods in CCTBX

- **Multiprocessing**: our tool of choice, with some modifications for easier coding
- **Threading**: works poorly for pure-Python code due to Global Interpreter Lock (GIL), although this can be circumvented in C++ or by starting child processes; mostly used internally
- **OpenMP**: C++ directives enable automatic parallelization by compiler; easy to use, but problematic for us
- **CUDA/OpenCL**: GPU acceleration, potentially useful for some applications (e.g. direct summation) but of limited use for Phenix; difficult to distribute or support
- Other hybrid methods possible (e.g. threading + queuing system)

The `multiprocessing` module

- Introduced in Python 2.6; used extensively in CCTBX and Phenix GUI
- Cross-platform support for non-shared memory parallelization via separate processes, with communication via pipes and queues
- Basic API similar to `threading` module
- `Pool` class creates persistent process pool and farms out jobs with automatic load balancing
- Main limitation: target function and all input and output objects must be pickle-able*, which requires extra work for Boost-wrapped C++ classes

* pickle = Python object serialization format, represents objects as binary strings

A simple example from the Python manual*

- Except for the pickling restriction, this is very similar to the `threading` equivalent - but genuinely parallel

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print q.get()    # prints "[42, None, 'hello']"
    p.join()
```

Disadvantage: using the API this way requires explicit parallelization within application code

* *http://docs.python.org/2/library/multiprocessing.html*

`libtbx.easy_mp`: parallel `map()` implementations

- Many of the rate-limiting steps in MX are "embarrassingly parallel": multiple independent calls to the same function
  - equivalent to built-in function `map(func, iterable)`
  - examples in Phenix: refinement weight optimization, multiple MR searches, Rosetta building, ligand fitting
- In these cases an even simpler API is helpful
- Since much of the calling code was written to run in serial, parallelization may be difficult without extensive refactoring (e.g. to work around pickling limitation)
- Although these implementations provide parallelism, they can also be run in serial if multiprocessing is not desired or not available - no need for additional if/else logic in applications

`pool_map`: multiprocessing for the impatient

- Ralf's solution to pickling problem: hack the Pool class to take advantage of internal `fork()` calls on Unix-like systems
- The function may be specified in one of two ways:
  - `func` is used as in the Pool, and pickled
  - `fixed_func` will be saved as a reference in forked processes, avoiding pickling
    - usually this would be an object method, with the object holding most of the data (not passed as arguments!)
- In practice, copy-on-write behavior of `fork()` means that large objects (such as `scitbx.array_family` arrays) will essentially be in shared memory as long as they are not modified
- <u>This will not work on Windows, which does not have fork() and must start entirely new Python interpreter processes</u>

`pool_map` in action: before

Code written for serial execution:

```
class optimize_xyz_refinement_weight (object) :
  def __init__ (self, model, fmodel, params,
      out=sys.stdout) :
    self.model = model
    self.fmodel = fmodel
    self.params = params
    self.trial_results = []
    for weight in [0.1, 0.25, 0.5, 1.0, 2.0, 5.0] :
      self.trial_results.append(self.try_weight(weight))

  def try_weight (self, weight) :
    # function defined elsewhere; modifies objects in place
    out = StringIO()
    minimize_coordinates(
      model=self.model,
      fmodel=self.fmodel,
      weight=weight,
      log=out)
    sites_cart = self.fmodel.xray_structure.sites_cart()
    return (self.fmodel.r_free(), weight, sites_cart)
```

pool_map in action: after

The same code, parallelized:

```
class optimize_xyz_refinement_weight (object) :
  def __init__ (self, model, fmodel, params,
      out=sys.stdout, nproc=Auto) :
    self.model = model
    self.fmodel = fmodel
    self.params = params
    self.trial_results = libtbx.easy_mp.pool_map(
      fixed_func=self.try_weight,
      args=[0.1, 0.25, 0.5, 1.0, 2.0, 5.0],
      nproc=nproc)

  def try_weight (self, weight) :
    ...
```

No additional refactoring is required for this to work!

`parallel_map`: adding queuing systems

- Wrapper for modules written by Gabor Bunkoczi; currently supports SGE, PBS, LSF, and Condor, in addition to multiprocessing and threading
  - Mac and Windows limited to the latter two modes
- Communication handled by temporary files when a queuing system is used
  - note that NFS latency can be problematic here
- Common libtbx.phil parameter block can be embedded in end-user applications
- The target function needs to be pickled, but this means we can also get full parallelization on Windows

An example of parallel_map use

Run multiple MR searches with different models:

```
class phaser_manager (object) :
  def __init__ (self, data_file) :
    self.data_file = data_file

  def __call__ (self, model) :
    # the actual implementation is elsewhere
    return run_phaser(self.data_file, model)

def run_all (data_file, models, method="multiprocessing",
    processes=1, qsub_command=None, callback=None) :
  phaser = phaser_manager(data_file)
  from libtbx.easy_mp import parallel_map
  return parallel_map(
    func=phaser,
    iterable=models,
    method=method,
    processes=processes,
    callback=callback,
    qsub_command=qsub_command)
```

method could also be "sge", "pbs", "condor", or "lsf"

Limitations of multiprocessing

- I have found handling of exceptions in subprocesses problematic - at present it is better if the application code does this
    - `KeyboardInterrupt` often not handled properly*
- Avoid printing to stdout/stderr; `pool_map` can be called with `func_wrapper="buffer_stdout_stderr"` to intercept output
    - this will return tuples of results and output strings
    - the disadvantage is we can't see output for each task as it completes - optional callbacks can partially alleviate this

* *parallel_map does not have this limitation, but pool_map currently does - we will probably fix this in the near future*

More advanced parallelization tools

- See previous two issues of our newsletter*
- Gabor's implementation of parallel MR search uses the same API as `parallel_map`, but at a lower level
- Core modules are in `libtbx.queuing_system_utils` (although not strictly limited to queuing systems)
- Many more options available here, allowing for greater optimization for custom tasks where the assumptions made in `parallel_map` are inappropriate
- We would like all of these to be as robust and generally applicable as possible, so further improvements can and will be made

* http://www.phenix-online.org/newsletter

Other ideas we haven’t tried

- **Hadoop**: open-source MapReduce implementation, very scaleable and fault-tolerant, suitable for cloud computing; written in Java but supports Python
    - In theory Gabor’s library could be extended to support this, but it appears considerably more complex than simple queuing systems
- I believe **Condor** has additional capabilities beyond what we use right now
- **MPI**: message-passing for highly parallel, speed-optimized computations; very efficient but more difficult to program (and/or run)
- The optimal solution may depend on intended use: distributed applications have many more constraints than local setups such as beamline clusters

Part II: a few quick words about unmerged data

Unmerged data in CCTBX: current state

- Supported input formats include MTZ, Scalepack, XDS, SHELX, CIF
  - note that we do not do much with batch numbers and other experimental parameters
- Only CIF output is possible at present - could add MTZ
- phenix.merging_statistics will calculate intensity stats, R-factors, CC1/2, etc.
  - Xtriage will automatically call this if appropriate
- phenix.cc_star calculates CC* and related model-based statistics (Karplus & Diederichs 2012)
- In every other program we immediately merge redundant observations

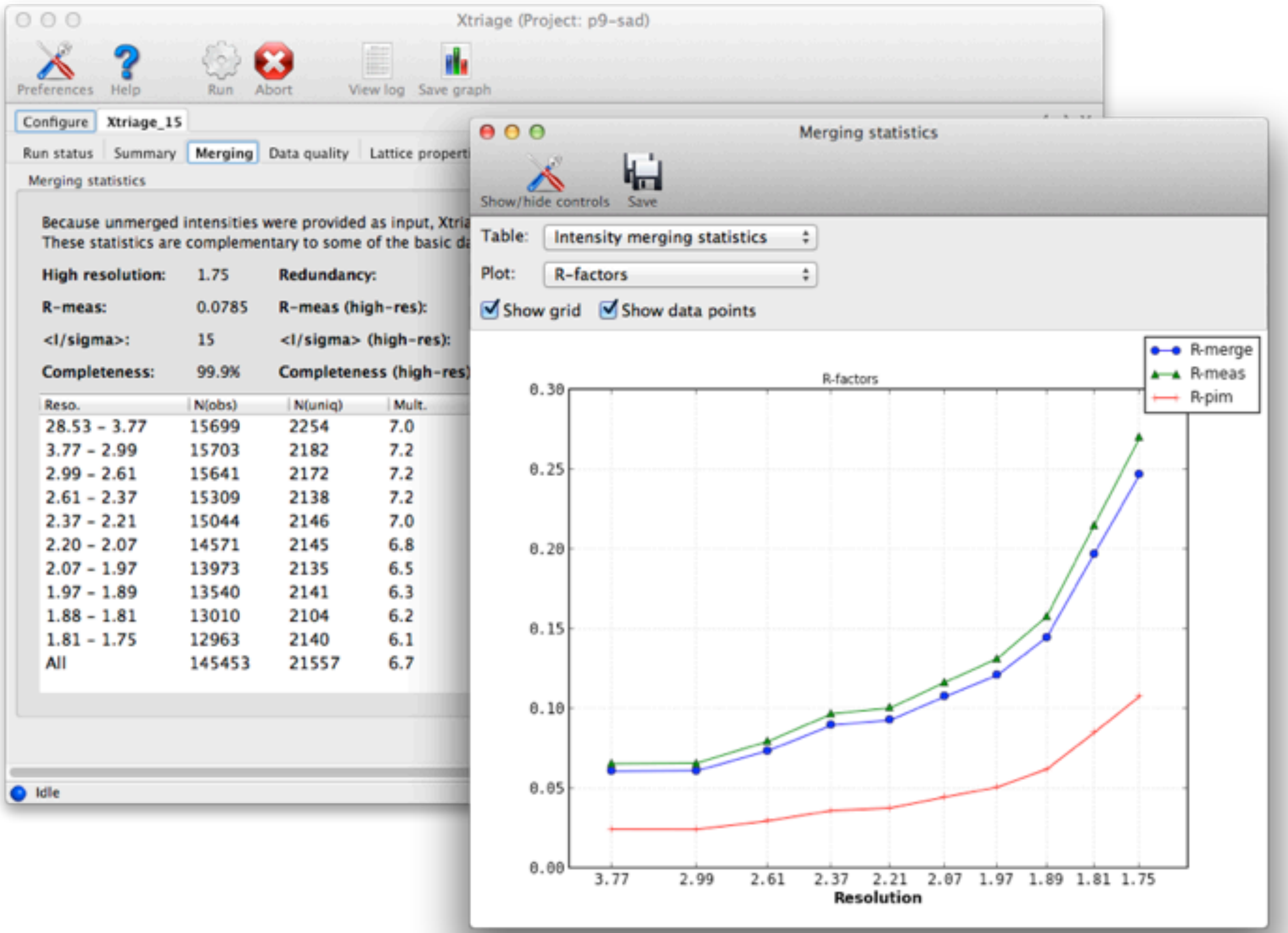
phenix.merging_statistics

- Accepts any unmerged data format we have parsers for
- Similar output to SCALA et al.; reports merging R-factors and basic intensity statistics
- We can easily add any number of other statistics (R_{ano}?) - most of these don't even require C++ code
- The only real limitation is how much we can display at once

```
 Statistics by resolution bin:
d_max  d_min   #obs  #uniq   mult.  %comp       <I>  <I/sI>  r_mrg r_meas  r_pim  cc1/2
28.53   3.77  15699   2254    6.96  99.87   78997.8    23.4  0.061  0.066  0.025  0.997
 3.77   2.99  15703   2182    7.20  99.95   47400.1    23.1  0.061  0.066  0.024  0.997
 2.99   2.61  15641   2172    7.20 100.00   17930.9    21.1  0.074  0.080  0.030  0.996
 2.61   2.37  15309   2138    7.16 100.00   10520.1    18.6  0.090  0.097  0.036  0.995
 2.37   2.21  15044   2146    7.01  99.95    9103.8    17.2  0.093  0.101  0.038  0.995
 2.20   2.07  14571   2145    6.79 100.00    6560.2    13.5  0.108  0.117  0.045  0.993
 2.07   1.97  13973   2135    6.54 100.00    5016.1    10.8  0.121  0.131  0.051  0.992
 1.97   1.89  13540   2141    6.32 100.00    3620.6     8.6  0.145  0.158  0.062  0.984
 1.88   1.81  13010   2104    6.18  99.95    2070.5     6.8  0.197  0.215  0.085  0.980
 1.81   1.75  12963   2140    6.06  99.49    1477.4     5.6  0.247  0.270  0.108  0.970
28.53   1.75 145453  21557    6.75  99.92   18672.0    14.9  0.073  0.079  0.030  0.998
```

phenix.merging_statistics: graphical display

- The actual GUI is part of Phenix, but nearly all of the building blocks (including plot window) are in CCTBX; can also output loggraph format



Merging statistics

Because unmerged intensities were provided as input, Xtria
These statistics are complementary to some of the basic da

High resolution:	1.75	**Redundancy:**	
R-meas:	0.0785	**R-meas (high-res):**	
<I/sigma>:	15	**<I/sigma> (high-res):**	
Completeness:	99.9%	**Completeness (high-res)**	

Reso.	N(obs)	N(uniq)	Mult.
28.53 - 3.77	15699	2254	7.0
3.77 - 2.99	15703	2182	7.2
2.99 - 2.61	15641	2172	7.2
2.61 - 2.37	15309	2138	7.2
2.37 - 2.21	15044	2146	7.0
2.20 - 2.07	14571	2145	6.8
2.07 - 1.97	13973	2135	6.5
1.97 - 1.89	13540	2141	6.3
1.88 - 1.81	13010	2104	6.2
1.81 - 1.75	12963	2140	6.1
All	145453	21557	6.7

Table: Intensity merging statistics
Plot: R-factors
✓ Show grid ✓ Show data points

Long-term goals

- Automatic estimation of resolution limit?
- Use unmerged data in preparation of PDB depositions, Table 1
  - this will also facilitate deposition of the unmerged intensities
- Add support for unmerged data output as MTZ
  - and better support for CIF
- Incorporate local scaling (T. Terwilliger)

- Scientific goals (as part of Phenix project): use unmerged data directly in phasing and refinement

Acknowledgments

Gabor Bunkoczi
Ralf Grosse-Kunstleve

Kay Diederichs
Keitaro Yamashita
Andy Karplus
Markus Rudolph
Phil Evans
Luc Bourhis
Tom Terwilliger