1 Step 0. Connect to IPython Cluster

```
from IPython import parallel
rc = parallel.Client(packer='pickle')
view = rc.load_balanced_view()
print "We have %i engines available" % len(rc.ids)
# skip cached results (for faster debugging)
rc[:]['skip_cache'] = False
```

```
We have 16 engines available
```

A utility for monitoring progress while waiting on an AsyncResult:

```
import sys
from datetime import datetime
from IPython.core.display import clear_output

def wait_on(ar):
    N = len(ar.msg_ids)
    rc = ar._client
    submitted = rc.metadata[ar.msg_ids[0]]['submitted']

    while not ar.ready():
        ar.wait(1)
        progress = sum([ msg_id not in rc.outstanding for msg_id in ar.msg_ids ])
        dt = (datetime.now()-submitted).total_seconds()
        clear_output()
        print "%3i/%3i tasks finished after %4i s" % (progress, N, dt),
        sys.stdout.flush()
    print
    print "done"
```

2 Step 1. Slice alignments

```
base_region_boundaries = [
      ('v2', 136, 1868), #27f-338r
      ('v2.v3', 136, 2232),
      ('v2.v4', 136, 4051),
      ('v2.v6', 136, 4932),
      ('v2.v8', 136, 6426),
      ('v2.v9', 136, 6791),
      ('v3', 1916, 2232), #349f-534r
      ('v3.v4', 1916, 4051),
      ('v3.v6', 1916, 4932),
      ('v3.v8', 1916, 6426),
      ('v3.v9', 1916, 6791),
      ('v4', 2263, 4051), #515f-806r
      ('v4.v6', 2263, 4932),
      ('v4.v8', 2263, 6426),
```

```
    ('v4.v9', 2263, 6791),
    ('v6', 4653, 4932), #967f-1048r
    ('v6.v8', 4653, 6426),
    ('v6.v9', 4653, 6791),
    ('v9', 6450, 6791), #1391f-1492r
    ('full.length', 0, 7682), # Start 150, 250, 400 base pair reads
    ('v2.150', 136, 702),
    ('v2.250', 136, 1752),
    ('v2.v3.400', 136, 2036), # Skips reads that are larger than amplicon size
    ('v3.v4.150', 1916, 2235),
    ('v3.v4.250', 1916, 2493),
    ('v3.v4.400', 1916, 4014),
    ('v4.150', 2263, 3794),
    ('v4.250', 2263, 4046),
    ('v4.v6.400', 2263, 4574),
    ('v6.v8.150', 4653, 5085),
    ('v6.v8.250', 4653, 5903),
    ('v6.v8.400', 4653, 6419)
]
print "%i regions, which we will break up into tasks to be done in parallel" % len(
    base_region_boundaries)
```

```
32 regions, which we will break up into tasks to be done in parallel
```

```
base_percentages = [82]

seq_file_base = '/home/ubuntu/qiime_software/gg_otus-4feb2011-release/rep_set/gg_%
    i_otus_4feb2011_aligned.fasta'
```

If we want to use multiple seq_files, that's a nested list:

```
sub_alignments = []
for seq_file in seq_files:
    for region_boundary in region_boundaries:
        sub_alignemnts.append(load_sub_alignment(seq_file, region_boundary)
```

This can actually be transformed into a flat list suitable for map with clever use of `itertools.product`:

```
import itertools
list_of_tuples = itertools.product(base_percentages, base_region_boundaries)
percentages, region_boundaries = zip(*list_of_tuples)
seq_files = [seq_file_base % i for i in percentages]
labels = [ "%i.%s" % (p,rb[0]) for p,rb in zip(percentages, region_boundaries) ]

ntasks = len(region_boundaries)
ntasks
```

```
32
```

Loading data is an expensive operation. This takes the most time, of any steps

```
def load_sub_alignment(seq_file, region_boundary):
    """load subregion of data into new file"""
    from cogent import LoadSeqs
    from cogent.core.alignment import DenseAlignment
    import os
    id_, start, end = region_boundary
    base, ext = os.path.splitext(os.path.basename(seq_file))
    sub_fname = '/home/ubuntu/data/' + base + "_%s" % id_ + ext
    if skip_cache and os.path.exists(sub_fname):
        # skip if we've already generated it
        return sub_fname
    aln = LoadSeqs(seq_file, aligned=DenseAlignment)
    sub_alignment = aln.takePositions(range(start, end))
    sub_alignment.writeToFile(sub_fname)
    return sub_fname
```

Submit the loads to be done in parallel

```
amr = load_amr = view.map_async(load_sub_alignment, seq_files, region_boundaries)
```

Submission is asynchronous, and returns immediately.

Now we wait for the computations to actually finish, returning the list of filenames for the subregions.

this will take time

```
wait_on(amr)
sub_aligns = amr.get()
sub_aligns
```

```
32/ 32 tasks finished after  236 s

done

['/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.v3.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.v4.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.v6.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.v8.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.v9.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.v4.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.v6.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.v8.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.v9.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v4.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v4.v6.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v4.v8.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v4.v9.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v6.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v6.v8.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v6.v9.fasta',
 '/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v9.fasta',
```

```
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_full.length.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.150.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.250.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v2.v3.400.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.v4.150.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.v4.250.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v3.v4.400.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v4.150.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v4.250.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v4.v6.400.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v6.v8.150.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v6.v8.250.fasta',
'/home/ubuntu/data/gg_82_otus_4feb2011_aligned_v6.v8.400.fasta']
```

Now let's take a quick peek at the overhead of performing this compuation with IPython

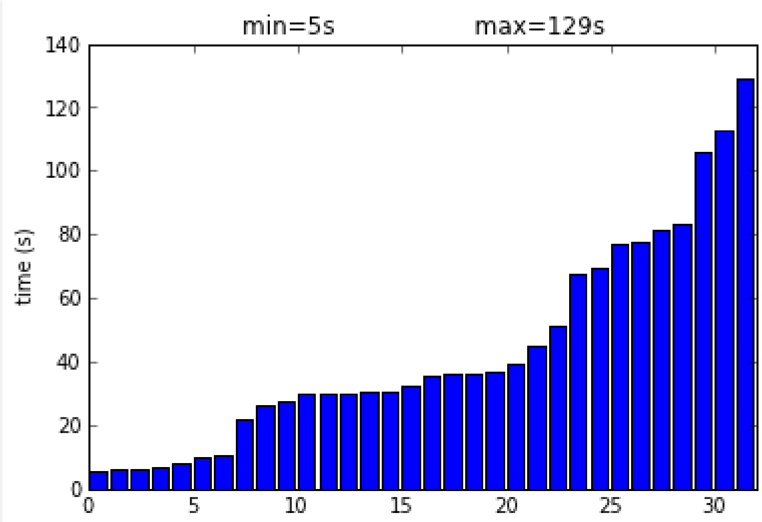
```
def print_parallel_stats(ar):
    """print some performance info for a given AsyncResult"""
    ar.wait()
    serial = 0.
    times = []
    for start,stop in zip(ar.started, ar.completed):
        elapsed = (stop-start).total_seconds()
        times.append(elapsed)
    longest = max(times)
    serial = sum(times)
    finished = max(ar.received)
    submitted = min(ar.submitted)
    wall = (finished - submitted).total_seconds()

    bar(range(len(times)), sorted(times))
    xlim(0, ntasks)
    ylabel("time (s)")
    title("min=%is max=%is" % (min(times), max(times)))

    print "ran %.1fs of work in %.1fs in %i tasks on %i engines" % (serial, wall, len(ar.
        msg_ids), len(rc.ids))
    print "for a speedup of %.1fx" % (serial/wall)
    print "longest task was %.1fs, which is the best we could hope to do." % (longest)
    print "IPython overhead: %ippm" % (1e6*(wall-longest)/longest)
```

```
print_parallel_stats(load_amr)
```

```
ran 1397.5s of work in 236.3s in 32 tasks on 16 engines
for a speedup of 5.9x
longest task was 129.3s, which is the best we could hope to do.
IPython overhead: 826950ppm
```



3 Step 2. Filter hypervariable positions and mostly gapped positions

```
def filter_alignment(fname):
    """call out to subcommand, which filters the positions"""
    import os
    import subprocess
    cmd = "filter_alignment.py -i %s -e 0.1 -g 0.8 --suppress_lane_mask_filter -o /home/
        ubuntu/data/" % fname
    # RUN cmd
    subprocess.call(cmd, shell=True)
    base, ext = os.path.splitext(fname)
    filtered = base + '_pfiltered' + ext
    return filtered
```

This one is quick, so we do it synchronously, but still in parallel:

```
%time filtered = view.map_sync(filter_alignment, sub_aligns)
```

```
CPU times: user 0.51 s, sys: 0.07 s, total: 0.58 s
Wall time: 5.37 s
```

4 Step 3. Build trees in parallel

```
def build_tree(filtered_aln_fp):
    """build tree from a filtered alignment"""
    import os
    from cogent import LoadSeqs
    from cogent.core.alignment import DenseAlignment
    from cogent.app.fasttree import build_tree_from_alignment
```

```
    from cogent import DNA

    tree_fp = '%s.tre' % os.path.splitext(filtered_aln_fp)[0]
    if skip_cache and os.path.exists(tree_fp):
        # skip already done
        return tree_fp
    tree = build_tree_from_alignment(LoadSeqs(filtered_aln_fp, aligned=DenseAlignment),
        moltype=DNA)
    tree.writeToFile(tree_fp,with_distances=True)
    return tree_fp
```

This is the other step that takes some real time.

this will take time

```
amr = tree_amr = view.map_async(build_tree, filtered)
```

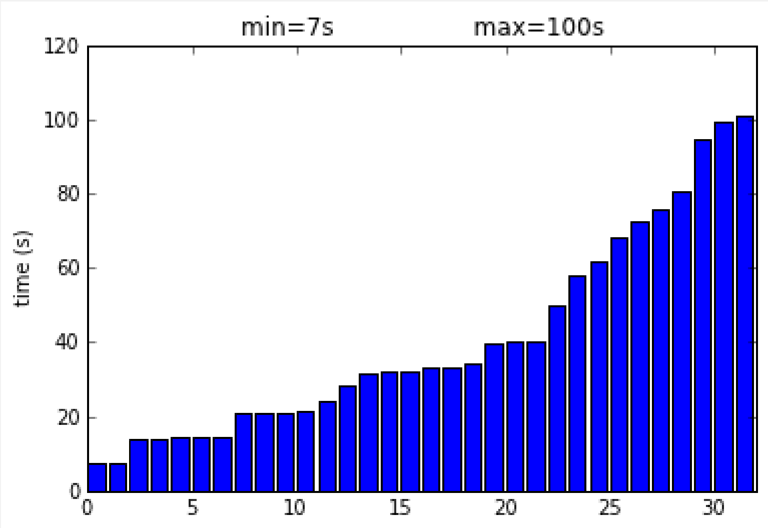
```
wait_on(amr)
trees = amr.get()
```

```
32/ 32 tasks finished after  200 s

done
```

```
print_parallel_stats(tree_amr)
```

```
ran 1299.5s of work in 200.1s in 32 tasks on 16 engines
for a speedup of 6.5x
longest task was 100.7s, which is the best we could hope to do.
IPython overhead: 986627ppm
```



5 Step 4. Compute distances between trees

`compare_trees()` computes the distance *submatrix* corresponding to a given tree.

```
def compare_trees(list_of_tree_files, i, nreps=50, sample_percent=0.1):
    """compute section of distance matrix for a single tree"""
    from cogent.parse.tree import DndParser
    from numpy import zeros, mean

    trees = [DndParser(open(f)) for f in list_of_tree_files]
    dist_mat = zeros((len(trees), len(trees)))
    t1 = trees[i]
    t1_ntips = len(t1.tips())
    for dj,t2 in enumerate(trees[i+1:]):
        j = i+dj+1
        sample_size = int(round((min(t1_ntips, len(t2.tips())) * sample_percent)))
        distances = [t1.compareByTipDistances(t2, sample=sample_size) for r in range(
            nreps)]
        dist_mat[i,j] = mean(distances)
        dist_mat[j,i] = mean(distances)
    return dist_mat
```

For instance, the distance elements for the third-to-last tree:

```
%precision 3
compare_trees(trees, ntasks-3)
```

```
array([[ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.   ,  0.   ],
       ...,
       [ 0.   ,  0.   ,  0.   , ...,  0.   ,  0.087,  0.132],
       [ 0.   ,  0.   ,  0.   , ...,  0.087,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   , ...,  0.132,  0.   ,  0.   ]])
```

We can then compute these submatrices in parallel

```
map_trees = [trees]*ntasks
amr = view.map_async(compare_trees, map_trees, range(ntasks)[::-1], ordered=False)
```

And compute the final distance matrix by peroming a sum (via builtin `reduce()`)
this will take the most time

```
def _print_progress(ar):
    N = len(ar.msg_ids)
    rc = ar._client
    submitted = rc.metadata[ar.msg_ids[0]]['submitted']
    progress = sum([ msg_id not in rc.outstanding for msg_id in ar.msg_ids ])
    dt = (datetime.now()-submitted).total_seconds()
    clear_output()
    print "%4i/%3i tasks finished after %4i s" % (progress, N, dt),
```

```
    sys.stdout.flush()

def progress_sum(a,b):
    c = a+b
    _print_progress(amr)
    return c

dist_mat = reduce(progress_sum, amr, 0)
dist_mat.tofile('/home/ubuntu/data/dist_mat_fast.np')
```

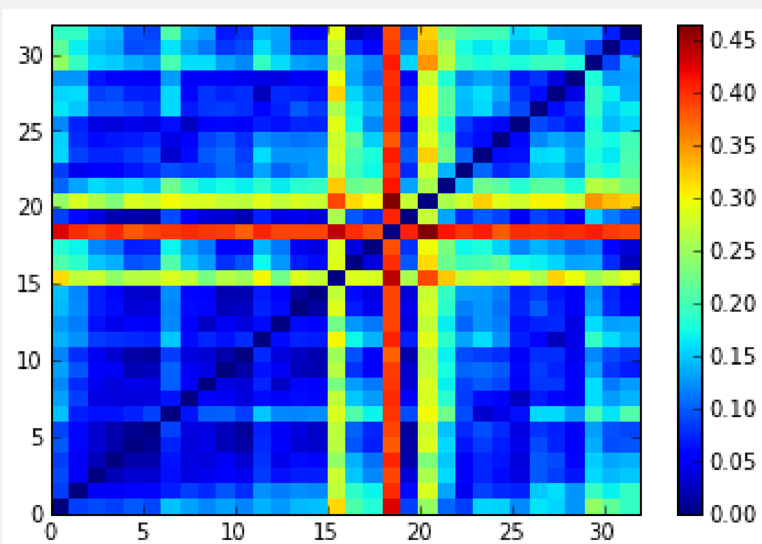
```
32/ 32 tasks finished after  174 s
```

Now we can peek at the distance matrix, to see if there is anything interesting.

```
# Uncomment here to load dist_mat from cache, to regenerate plots
# import numpy
# dist_mat = numpy.fromfile('/home/ubuntu/data/dist_mat.np').reshape(ntasks,ntasks)
```

```
pcolor(dist_mat)
xlim(0,ntasks)
ylim(0,ntasks)
colorbar()
```

```
<matplotlib.colorbar.Colorbar instance at 0x7fd8b47161b8>
```



Write QIIME's distance matrix format

```
from qiime.format import format_distance_matrix

open('/home/ubuntu/data/distance_matrix_fast.txt','w').write(format_distance_matrix(
    labels,dist_mat))
```

6 Step 5. Compute PCoA: QIIME/PyCogent

```
!principal_coordinates.py -i /home/ubuntu/data/distance_matrix_fast.txt -o /home/ubuntu/
    data/pc_fast.txt
```

7 Step 6. Display PCoA: QIIME

This generates an HTML file and java visualization for the data. To do this we need one additional file: tree_metadata.txt. You can view this file directly here.

```
!wget http://qiime.org/home_static/nih-cloud-apr2012/tree_metadata.txt
```

```
--2012-08-06 22:20:12--  http://qiime.org/home_static/nih-cloud-apr2012/tree_metadata.txt
Resolving qiime.org... 216.34.181.97
Connecting to qiime.org|216.34.181.97|:80... connected.
HTTP request sent, awaiting response...
200 OK
Length: 9313 (9.1K) [text/plain]
Saving to: 'tree_metadata.txt'


 0% [                                       ] 0           --.-K/s
100%[======================================>] 9,313       --.-K/s   in 0.02s

2012-08-06 22:20:13 (393 KB/s) - 'tree_metadata.txt' saved [9313/9313]
```

We can then generate 3D PCoA plots.

```
!make_3d_plots.py -i /home/ubuntu/data/pc_fast.txt -o /home/ubuntu/data/pcoa_plots/ -m /
    home/ubuntu/tree_metadata.txt
```

And the notebook simply serves these files up in `/files`, so we can visit the visualization directly **NOTE**: The above link is not static: to view the plot, you must run the notebook.