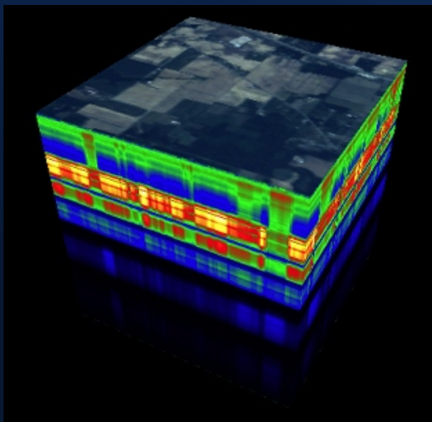
Parallel Implementation of the k-means Clustering Algorithm for Unsupervised Classification of Hyperspectral Imagery

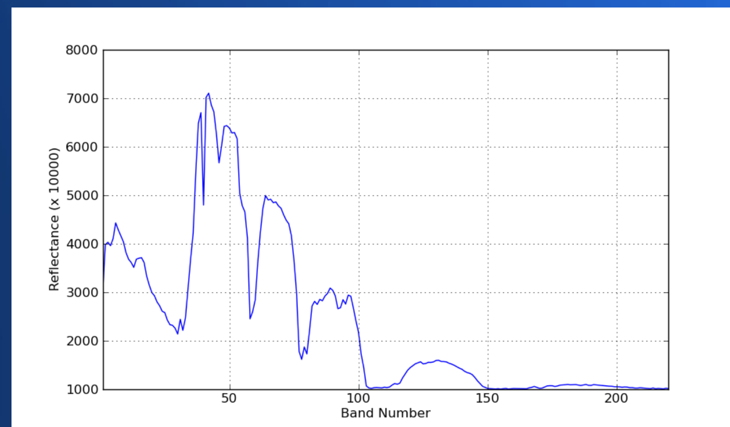
Thomas Boggs
CSI 702
6 May 2010

Hyperspectral Imaging (HSI)

- Hyperspectral imagers are specialized cameras – instead of producing 3-band (RGB) images, they create images at hundreds of wavelengths.
- Each pixel in a hyperspectral image contains an entire spectrum of data.



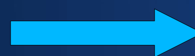
Pixel (100,100)



Clustering

- Clustering is an unsupervised classification technique that determines groups (clusters) of observations that are in some way similar.
- Instead of considering an HSI pixel as *N* samples from a spectrum, we can consider it as an *N*-dimensional vector.
- Then we can create clusters based on proximity of pixels in the *N*-dimensional space.

Hyperspectral Image



Cluster map, where each unique color corresponds to a specific cluster in N-space

k-means Algorithm

- k-means (a.k.a, “migrating means”), is an iterative clustering algorithm that assigns pixels to the cluster with the nearest center (mean).
- Basic algorithm:
  1. Select number of clusters (k) to create
  2. Initialize means for each cluster (e.g., random pixels from image)
  3. While <not done>:
     1. Assign each pixel to the cluster with the nearest mean
     2. Recompute the cluster means from their assigned pixels
- Typical stopping criteria:
  - Max number of iterations reached
  - Fewer than *R* pixels reassigned between iterations
  - Cluster means migrate less than *X* after updating

Serial Implementation

- Assigning pixels to nearest cluster:

```
for (i = 0; i < numPixels; i++)
{
    pixel = image + i * numBands;

    nearestCluster = 0;
    minDist = distance(pixel, centers[0], numBands);

    for (j = 1; j < numClusters; j++)
    {
        dist = distance(pixel, centers[j], numBands);
        if (dist < minDist)
        {
            minDist = dist;
            nearestCluster = j;
        }
    }
    clusterMap[i] = nearestCluster;
}
```

- Note: numPixels is typically $O(10^5)$
  numClusters is typically $O(10^1)$
  numBands is typically $O(10^2)$
- Therefore, using Euclidean distance, assigning pixels requires $O(10^8)$ multiply-accumulate operations (MACs)

Serial Implementation

- Updating cluster centers:

```
for (i = 0; i < (int) numPixels; i++)
{
    addTo(image + i * numBands, centers[clusterMap[i]], numBands);
    clusterCounts[clusterMap[i]] +=1;
}

for (i = 0; i < (int) numClusters; i++)
{
    for (j = 0; j < numBands; j++)
        centers[i][j] /= clusterCounts[i];
}
```

- Updating cluster centers requires
  - $O(10^7)$ additions
  - $O(10^3)$ divisions
- Order of magnitude fewer FLOPS than cluster assignments (with no multiplications)

MPI Parallelization Strategy

- Perform domain decomposition by dividing image into *numProcs* contiguous chunks.
- Each MPI node assigns pixels from its own chunk of the image to appropriate clusters.
- Interprocess communication is required to update the cluster means

OpenMP Software Implementation

```
#pragma omp parallel default(shared) private(pixel, nearestCluster, minDist, j, dist)
#pragma omp for reduction(+:numChanged) schedule(runtime)

for (i = 0; i < numPixels; i++)
{
   pixel = image + i * numBands;

   nearestCluster = 0;
   minDist = distance(pixel, centers[0], numBands);

   for (j = 1; j < numClusters; j++)
   {
       dist = distance(pixel, centers[j], numBands);
       if (dist < minDist)
       {
          minDist = dist;
          nearestCluster = j;
       }
   }
   clusterMap[i] = nearestCluster;
   if (clusterMap[i] != nearestCluster)
       ++numChanged;
}
```

- Since the number of computations is fixed for a given number of iterations, static thread scheduling was chosen with OMP_SCHEDULE=static,1000

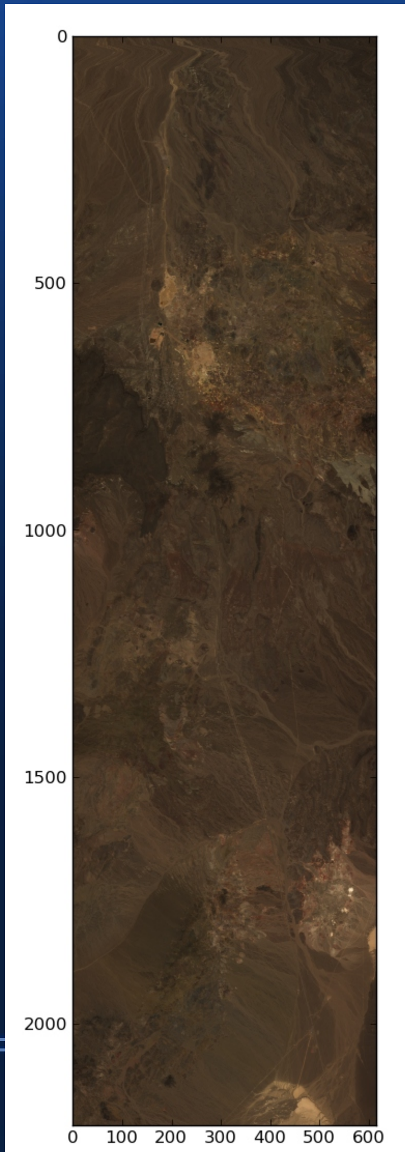
MPI Software Implementation

- Initialize clusters from image diagonal (for each cluster, broadcast from owning node to all others).
- Calculate initial cluster map on each node, then:

```
for (i = iStart; i < parser.maxIterations; i++)
{
    sumCenters(image, clusterMap, centers, numClusters, myNumPixels, numBands, clusterCounts);

    MPI::COMM_WORLD.Allreduce(MPI::IN_PLACE, centersBuffer, numClusters * numBands, MPI::FLOAT, MPI::SUM);
    MPI::COMM_WORLD.Allreduce(MPI::IN_PLACE, clusterCounts, numClusters, MPI::UNSIGNED_LONG, MPI::SUM);

    for (j = 0; j < numClusters; j++)
    {
        for (k = 0; k < numBands; k++)
        {
            centers[j][k] /= clusterCounts[j];
        }
    }

    numChanged = assignPixels(image, clusterMap, centers, numClusters, myNumPixels, numBands);
    MPI::COMM_WORLD.Allreduce(MPI::IN_PLACE, &numChanged, 1, MPI::UNSIGNED_LONG, MPI::SUM);
}
```

- Finally, MPI::Send final cluster map from each node to the root node.

AVIRIS Test Data

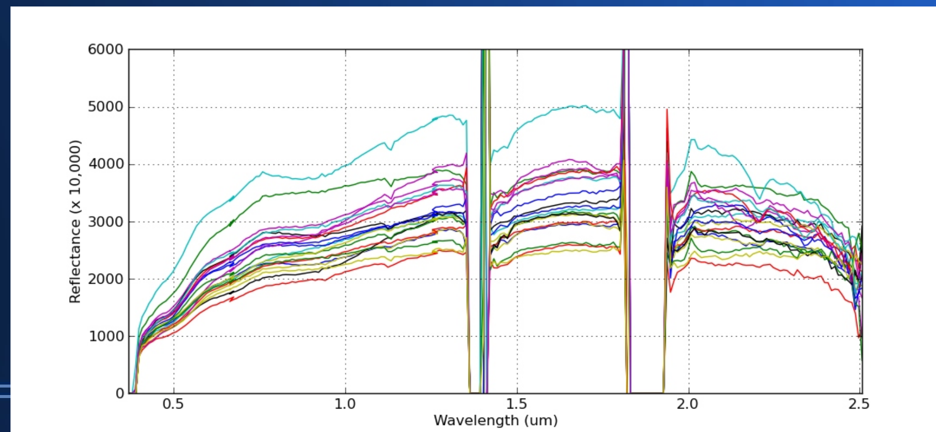


AVIRIS = Airborne Visible & InfraRed Imaging Spectrometer

- 224 spectral bands (0.400 – 2.5 um)
- 16 bit quantization

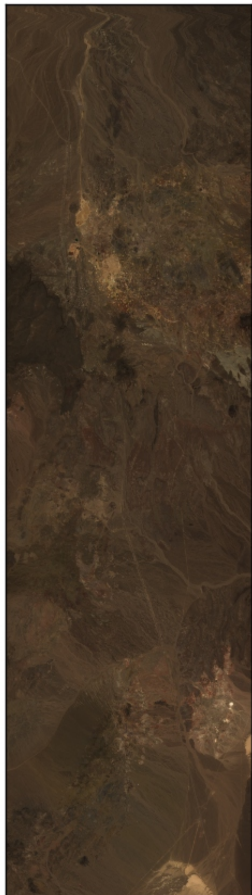
Test image is a flightline collected over Cuprite, NV

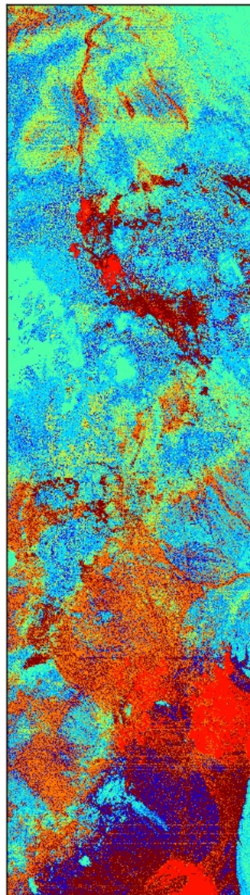
2206 rows x 614 cols x 224 bands x 16 bits/sample = ~608MB

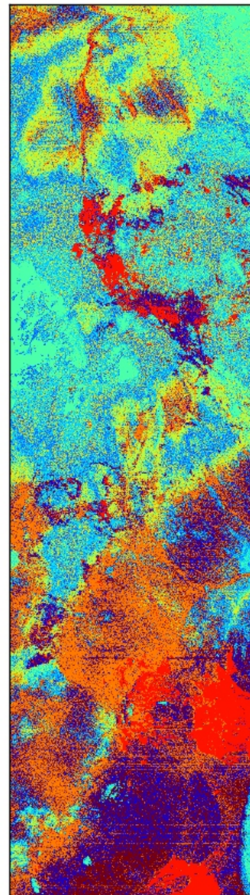


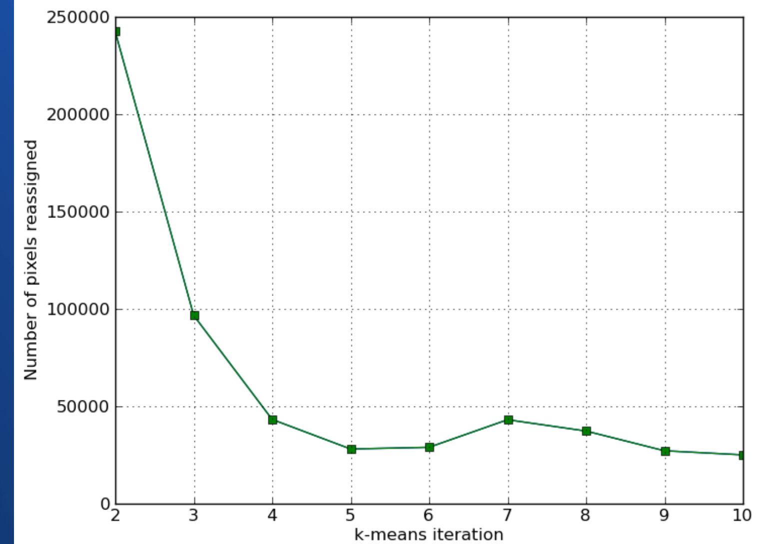
Sample pixels

Clustering Results


RGB View


1 Iteration


10 Iterations



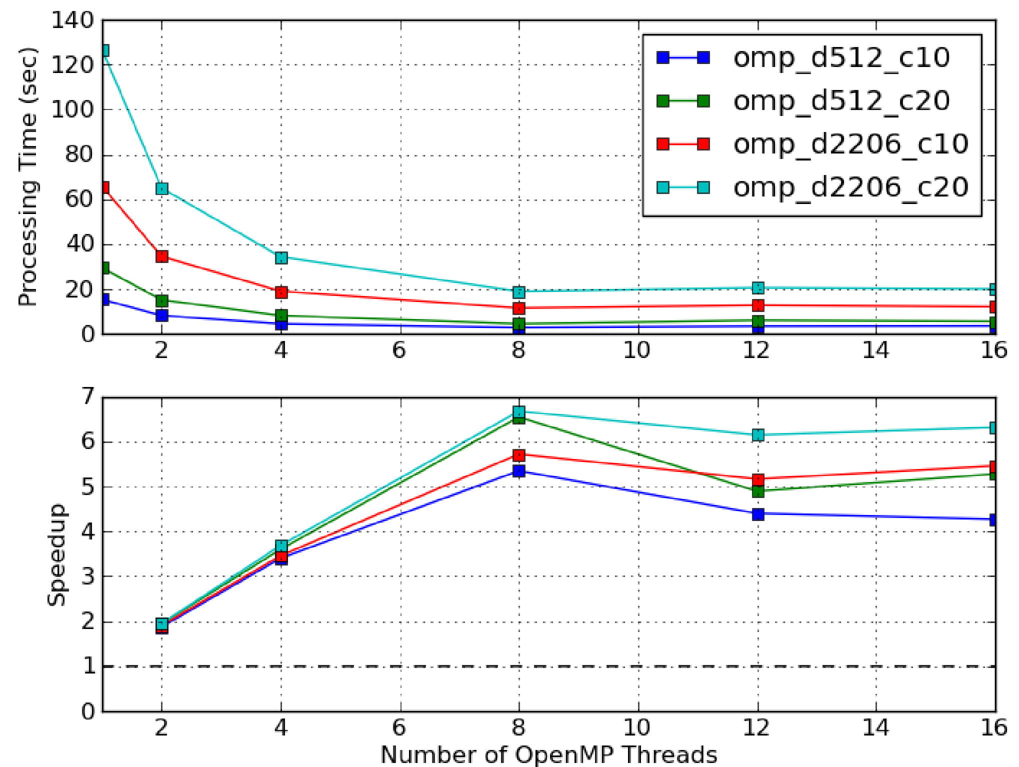
Parallel implementations may have minor deviations from serial cluster map due to finite floating point precision.

Performance Evaluation

- Versions of program tested
  - Serial, OpenMP, MPI, & hybrid OpenMP/MPI
- Parameters varied
  - 2206x614 & 512x614 pixel images
  - 10 & 20 k-means clusters
- Computing systems used to evaluate performance
  - OpenMP on gmice with 1 8-core system
  - MPI on CDS cluster with up to 2 processes per dual-core system
  - Hybrid OpenMP/MPI on CDS with one dual-threaded MPI node per dual-core system

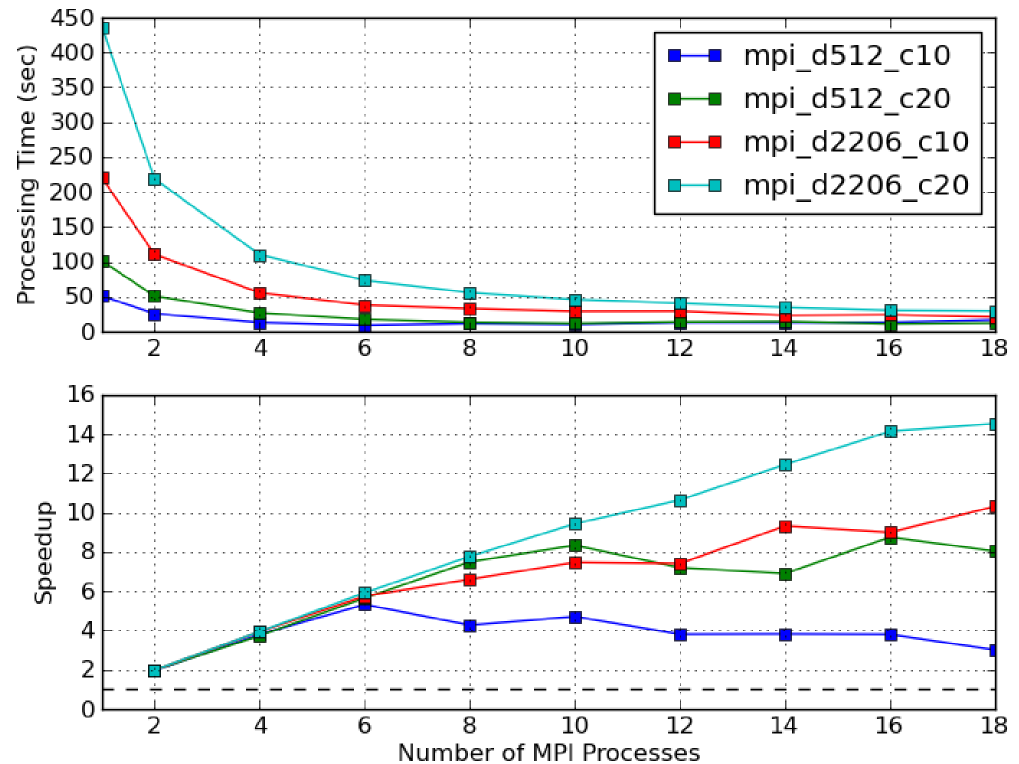
OpenMP Performance

- Test performed on an 8-CPU node on the gmice cluster.
- Speedup scales better for cases with 20 clusters than 10 clusters, due to increased computation for each parallel (outer) loop iteration.
- Speedup drops for more than 8 threads because CPUs must run multiple threads simultaneously.



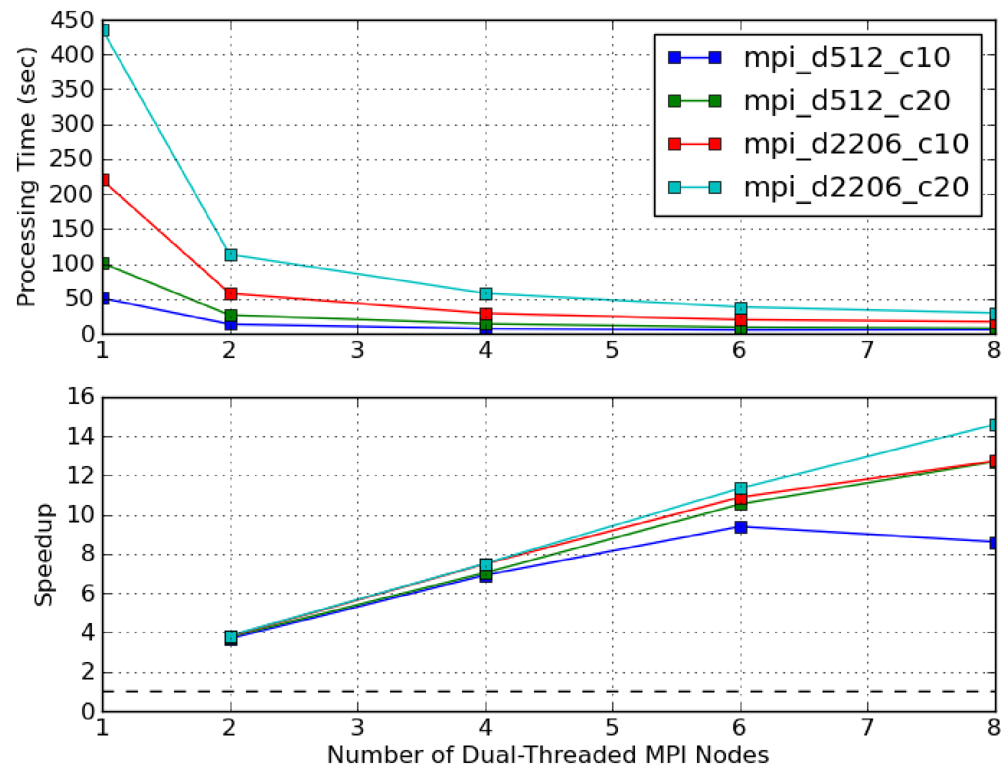
MPI Performance

- Test performed on CDS cluster using up to 9 dual-core nodes.
- Speedup scales nearly linearly up to 16 nodes for most computationally intensive case.
- Speedup for case with lowest computation drops after more than 6 processes.



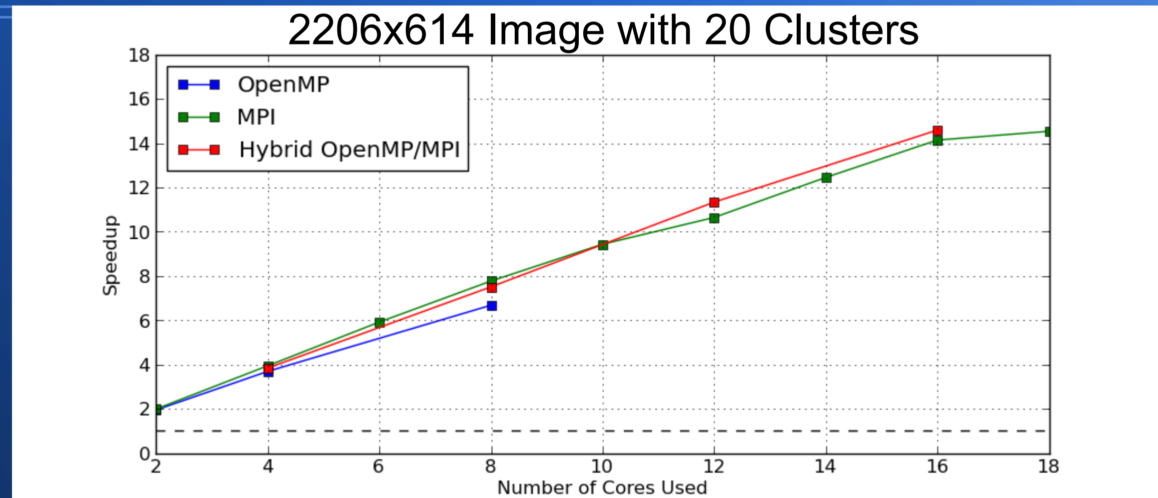
Hybrid OpenMP/MPI Performance

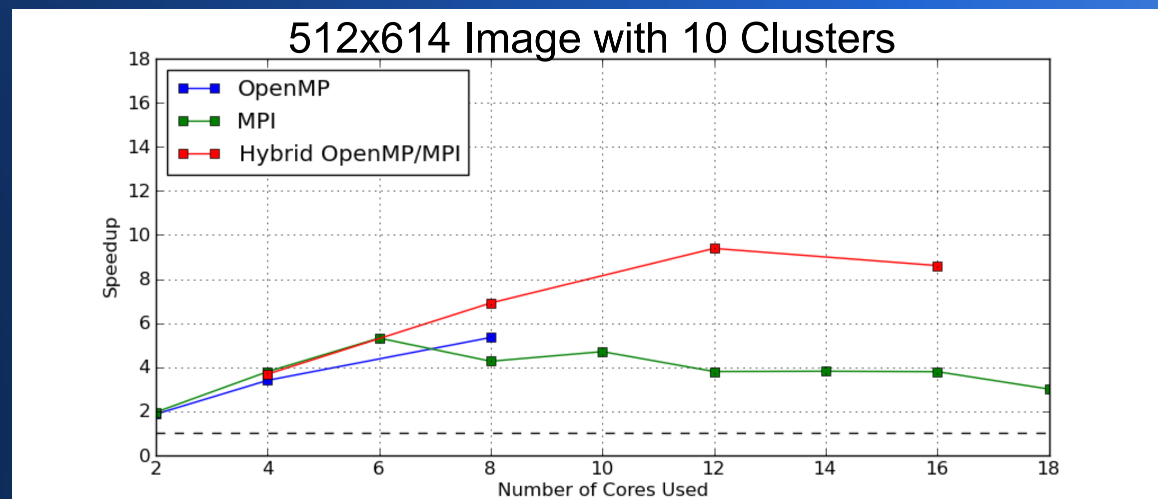
- Test performed on CDS cluster of dual-core workstations, using 2 OpenMP threads per MPI node.
- Speedup is approximately twice that of MPI-only for 3 of the 4 cases.



Comparing Parallel Programs

- Note:
  - MPI & Hybrid run on CDS cluster
  - OpenMP run on gmice cluster
  - Hybrid uses 2 threads per process
- MPI & Hybrid performance near identical up to point where MPI performance begins to drop.
- Lower speedup for OpenMP is possibly due to running on different cluster.
- For both MPI & Hybrid, speedup declines for greater than 6 MPI processes, suggesting that inter-process communications overhead is preventing benefit of additional parallelism.





Issues with gmice

- Zombie PBS jobs
- Software incompatibility between nodes?

```
You can't run mpdboot on ['r1i1n5'] version of python must be >= 2.4, current ['']
You can't run mpdboot on ['r1i1n4'] version of python must be >= 2.4, current ['']
You can't run mpdboot on ['r1i1n6'] version of python must be >= 2.4, current ['']
mpiexec: unable to start all procs; may have invalid machine names
    remaining specified hosts:
        10.148.1.177 (r1i1n5-ib0.ice.gmu.edu)
        10.148.1.178 (r1i1n6-ib0.ice.gmu.edu)
        10.148.1.176 (r1i1n4-ib0.ice.gmu.edu)
```

- gmice inaccessible:

```
tboggs@CDS05:~> ssh gmice.gmu.edu
ssh: connect to host gmice.gmu.edu port 22: Connection refused
```

Conclusion

- Parallelizing the k-means algorithm can clearly speed up computation:
  - Speedup > 14 for large problem using
    - MPI on 16 CPUs or
    - Dual-threaded MPI on 8 dual-core CPUs
  - Speedup > 9 for typical problem size using dual-threaded MPI on 6 dual-core CPUs
- Over-parallelizing for a given problem size can <u>reduce</u> performance.
- Should try running these problems on larger cluster or use more threads per MPI node to determine maximum possible speedup.

Tips & Lessons Learned

- Use MPI::IN_PLACE for in-place MPI reductions
  - Not required for OpenMP+gcc but Intel compiler causes run-time error without it.
  - `MPI::COMM_WORLD.Allreduce(MPI::IN_PLACE, buffer, count, ...`
- For hybrid MPI/OpenMP
  - With OpenMPI, must use special compiler flag syntax:
    - CXX → OMPI_CXX
    - LDFLAGS → OMPI_LDFLAGS
    - etc.
  - With Intel compilers, to see gain from threads, must set I_MPI_PIN_DOMAIN=omp