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

Release 1.0

Thomas Boggs

May 05, 2010

Contents

1 Introduction ii
1.1 Hyperspectral Imaging ii
1.2 Clustering ii
1.3 k-means Algorithm ii

2 Serial and Parallel Code iii
2.1 Serial Code iv
  Computational Burden iv
2.2 OpenMP v
2.3 MPI v
  Cluster Mean Initialization vi
  k-means Iterations vi
  Cluster Map Consolidation vii

3 Performance Testing viii
3.1 Test Data ix
3.2 Test Platforms ix
3.3 Clustering Results x
3.4 Test Parameters x
3.5 OpenMP x
3.6 MPI x
3.7 Hybrid OpenMP/MPI xi
3.8 Comparison of Parallel Implementations xi

4 Conclusion xiii

Bibliography xv
1 Introduction

1.1 Hyperspectral Imaging

Hyperspectral imaging (HSI) sensors (a.k.a., imaging spectrometers) are essentially specialized digital cameras. When a commodity digital camera collects an image, it near-simultaneously collects images corresponding to red, green, and blue wavelengths of light because those wavelengths correspond to color receptors in the human eye. By contrast, hyperspectral sensors image hundreds of wavelengths (possibly including red, green, and blue) such that each pixel in the image represents and entire spectrum of information. Numerous types of HSI sensors exist that measure various parts of the optical spectrum, from ultraviolet (UV) through long-wave infrared (LWIR).

![Figure 1: Hypercube view of a hyperspectral image and spectrum of a single pixel](image)

1.2 Clustering

A common first step when analyzing a hyperspectral image is to apply a clustering algorithm to attempt to identify image pixels with similar spectral content. Clustering algorithms attempt to divide a set of observations into groups of observations that are in some way similar. Rather than considering an HSI pixel to be a signal (or spectrum) sampled at \( N \) discrete points (as depicted in the figure above), we can instead consider each of the \( N \) values to represent an independent variable. From this perspective, the pixel can then be considered a vector defining a point in an \( N \)-dimensional space. Pixels that have similar spectra will be located closer to each other in \( N \)-space than pixels that are dissimilar. Thus, distance in \( N \)-space can be used as a similarity measure for clustering HSI pixels.

The figure below shows an RGB view of a hyperspectral image, along with a cluster map which is the result of applying a clustering algorithm to the image. In the cluster map image, all pixels of the same color belong to the same cluster in \( N \)-space. The particular colors used in the cluster map are arbitrary and serve only to distinguish one cluster from another. Note that pixels that appear similar in the RGB image may not belong to the same clusters since they may differ significantly at other wavelengths.

1.3 k-means Algorithm

The k-means algorithm (a.k.a. the “migrating means” algorithm) is one commonly used clustering algorithm [Schowengerdt]. With this algorithm, the user starts by selecting the number of clusters \( (k) \) to generate. The means of the \( k \) clusters (\( N \)-dimensional vectors) are then initialized by some method, such as

- Selecting the first \( k \) pixels in the image or selecting them evenly along the diagonal of the image.
Figure 2: RGB view of an HSI image and corresponding cluster map

- Randomly selecting unique pixels from the image
- Uniformly or randomly selecting means from the actual or theoretical N-dimensional bounding box of the image pixels.

After the means have been initialized, each pixel in the image is assigned to the cluster with the nearest mean. “Nearest” is defined in the context of the distance metric used. Euclidean (L2) distance is often used but Manhattan (L1) distance is sometimes used to reduce the computational complexity of the distance calculation. Once all pixels have been assigned, each cluster mean is updated by setting it to the average value of all pixels assigned to it. The process of assigning pixels and updating means is then repeated until a stopping criterion is met. After means are updated, some pixels will likely be assigned to different clusters during the next assignment phase. As iterations proceed, fewer pixels will tend to be reassigned during each iteration and the means will shift (migrate) less during successive iterations. Numerous stopping criteria can be used for the algorithm, including:

- A maximum number of iterations has been performed.
- Fewer than a threshold number of pixels are reassigned during an iteration.
- All means migrate less than a threshold distance during an update cycle.

Stopping criteria can also be combined. For example, a maximum number of iterations is often used in addition other criteria to prevent pathological situations that could result in prohibitively many iterations.

2 Serial and Parallel Code

2.1 Serial Code

Before addressing the parallel implementations of the algorithm, it is useful to first examine the serial implementation from which the parallel implementation is derived. The code is written in C++ for images where the spectral bands are interleaved by pixel. That is, the data file begins with the data for bands 1 through $N$ for the first pixel, followed by bands $q$ through $N$ for the second pixel, etc. Cluster means are initialized by selecting evenly spaced pixels from along the diagonal of the image being processed. The bulk of the computation for the algorithm is devoted to the pixel assignment and cluster mean updates phases, which will be addressed in turn. The code for pixel assignment (simplified here, for clarity) is as follows:
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 = numChanged;
    clusterMap[i] = nearestCluster;
}

While only two nested loops are shown in the code listing above, there is a third nested loop which occurs in the distance function call, which iterates over all bands to compute the distance between two N-dimensional vectors.

Updating the cluster means consists of two main loops: one to sum the pixels assigned to each cluster and another to divide by the numbers of pixels assigned to each clusters to determine the update mean:

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];
}

In the first for loop, the addTo function accumulates the first argument (pixel vector) in the second argument to sum all the pixels associated with each cluster. The second loop performs the division necessary to produce the updated cluster means.

**Computational Burden**

Before parallelizing the algorithm, it is useful to assess the computational burden of the serial algorithm. For the pixel assignment phase, the computational burden is driven by the 3 nested loops, which iterate over all pixels, all clusters, and all bands, respectively. The orders of magnitude for these quantities are as follows for typical hyperspectral applications:
Table 1: Parameter Magnitudes

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Magnitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>numPixels</td>
<td>$O(10^5)$</td>
</tr>
<tr>
<td>numClusters (k)</td>
<td>$O(10^4)$</td>
</tr>
<tr>
<td>numBands</td>
<td>$O(10^2)$</td>
</tr>
</tbody>
</table>

For clustering with Euclidean distance calculations, this results in $O(10^8)$ multiply & accumulate (MAC) operations for the pixel assignment phase of a k-means iteration. By comparison, the cluster means update phase only requires $O(10^3)$ additions to sum contributions (with no multiplications) and $O(10^3)$ divisions to finish computing the updated means. Thus, there is an order of magnitude more calculations required during the cluster assignment phase than the means update phase.

### 2.2 OpenMP

Since there is an order of magnitude more calculations required for the pixel assignment phase of the algorithm, parallelization with OpenMP focused on that phase of the algorithm. Parallelization was achieved by placing OpenMP `pragma` statements before the outermost loop, as follows:

```c
#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;
    clusterMap[i] = nearestCluster;
}
```

Parallelizing the outermost loop maximizes the amount of computation that is performed in a thread chunk. More computation could be achieved by reordering the loops such that iteration over pixels occurs in an inner loop; however, that would result in frequent cache misses so the loops were kept as is. Since the amount of computation performed per pixel is constant, static thread scheduling was chosen by setting the environment variable `OMP_SCHEDULE=static,1000`.

### 2.3 MPI

Since most of the computation occurs during the pixel assignment phase of a k-means iteration and all pixels are processed independently during that phase, domain decomposition was chosen for the MPI parallelization strategy.
Because MPI is a multi-process library (as opposed to multithreaded), modification of the source code for parallelization with MPI is significantly more invasive than OpenMP. For `numPixels` pixels in the hyperspectral image and `numProcs` MPI processes, each process is assigned a block of `numPixels/numProcs` pixels. The basic strategy is that each MPI node independently assigns its own block of pixels to the appropriate clusters. Then, interprocess communication is necessary to determine the global set of updated cluster locations and distribute them to all nodes for each k-means iteration.

**Cluster Mean Initialization**

Even though each process will have access to the full HSI image while processing, it was decided to have each pixel used to initialize a cluster be read from the data file only by the process owning the MPI block corresponding to that pixel. In fact, this would be necessary in a situation where the data file itself resides in a distributed archive such that different storage/processing nodes hold different chunks of the file. The cluster mean initialization pixels were distributed across the MPI nodes as follows:

```c
unsigned long pixelID;
int pixelRank; // <-- rank of node responsible for the pixel
for (i = 0; i < numClusters; i++)
{
    pixelID = (i * numRows / numClusters) * numCols + (i * numCols / numClusters);
    pixelRank = pixelID * numProcs / numPixels;
    if (rank == pixelRank)
    {
        // I own this pixel so copy it into my cluster centers buffer.
        copy(image + (pixelID - pStart) * numBands, centers[i], numBands);
    }
    MPI::COMM_WORLD.Bcast(centers[i], numBands, MPI::FLOAT, pixelRank);
}
```

Since each MPI node can determine which node is the owner of any pixel, only the owner of the block containing a particular initialization pixel copies it into its `centers` buffer. Then, all nodes perform an identical `Bcast` operation to copy it into the appropriate location in its local `centers` buffer.

After cluster centers have been initialized, each node performs an initial assignment of all its pixels to the appropriate clusters as follows:

```c
numChanged = assignPixels(image, clusterMap, centers, numClusters, myNumPixels, numBands);
```

The `assignPixels` function performs the same function as the pixel assignment loop in the serial code.

**k-means Iterations**

Each outermost loop of the k-means algorithm consists of an update of cluster means (centers), followed by a reassignment of all pixels to the shifted clusters. Interprocess communication is required to update the cluster centers because the mean position of the cluster requires summing pixels assigned to a given cluster across all MPI nodes.

```c
for (i = iStart; i < parser.maxIterations; i++)
{
    // Update cluster means (centers)
    sumCenters(image, clusterMap, centers, numClusters, myNumPixels, numBands, clusterCounts);
    MPI::COMM_WORLD.Allreduce(MPI::IN_PLACE, centers, 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++)
```
In the code above, centers is an array of length numClusters*numBands, which contains the means for each of the clusters. The sumCenters function replaces each of the cluster means in centers with the sum of all pixels (for the MPI node’s block of pixels) assigned to that cluster. To determine the updated cluster means, centers must be summed across all MPI nodes and each vector in centers must be divided by the global number of pixels assigned to the corresponding cluster. Since all of the sums are in one contiguous array, the global sum can easily be computed by performing an MPI Allreduce operation. The MPI::IN_PLACE value for the first argument indicates that the sums will be written to the same array as the source values.

To complete the update of the cluster means, the global counts of pixels assigned to each cluster is still required. Since the local counts are held in an array (clusterCounts) on each node, this can also be achieved by an MPI Allreduce operation. Once this has been accomplished, all that remains to update the means is to divide by the cluster counts. Since the total number of divisions is only a few thousand, it was decided to simply perform identical loops on all nodes rather than to parallelize that part of the calculation.

The last communication required in the iteration is to add the number of pixels reassigned across all nodes. This is accomplished by a third Allreduce call, this time with only a single value (the total number of pixels reassigned on each node).

**Cluster Map Consolidation**

The final step in the process is consolidating the partial cluster maps into a single cluster map for the entire image. This is accomplished by all non-root nodes performing a Send operation to the root node and the root node performing an Irecv to put each block of values into the appropriate region of the consolidated cluster map. All nodes then meet at an MPI Barrier.

if (rank == 0)
{
    for (i = 1; i < (unsigned int) numProcs; i++)
    {
        unsigned long iStart = i * numPixels / numProcs;
        unsigned long iStop = (i + 1) * numPixels / numProcs;
        if (i == (unsigned long) numProcs - 1)
            iStop = numPixels;
        MPI::COMM_WORLD.Irecv(clusterMap + iStart, iStop - iStart, MPI::SHORT, i, i);
    }
else
{
    MPI::COMM_WORLD.Send(clusterMap, myNumPixels, MPI::SHORT, 0, rank);
}
MPI::COMM_WORLD.Barrier();
3 Performance Testing

3.1 Test Data

The hyperspectral imagery used to evaluate the parallel algorithms is a flight-line collected by the Airborne Visible & InfraRed Imaging Spectrometer (AVIRIS) over Cuprite, NV. The AVIRIS sensor images 224 spectral bands, spanning 0.4-2.5 um. AVIRIS flight-lines are usually broken into segments of 512 rows; however, to test performance of the parallel k-means algorithms, the segments were recombined into an entire flight-line with 2206 rows and 614 columns. With 224 bands and 16-bit quantization, the entire image produces a file of over 606 MB.

Sample pixels from the image are plotted below. The image contains atmospherically corrected reflectance values (multiplied by 10,000). Deep atmospheric absorption bands are apparent near 1.4 and 1.8 um, as well as several saturated bands near the edges of the absorption bands. A persistent null or saturated value throughout an image does not adversely affect clustering; rather, it has the effect of reducing the dimensionality of the data by one.

![Image of RGB view of AVIRIS flight-line over Cuprite, NV](http://aviris.jpl.nasa.gov/html/aviris.freedata.html)
3.2 Test Platforms

The original plan for testing performance of the serial and parallel applications was to use the GMU gmice.gmu.edu computing cluster which provides hundreds of computing cores via the Portable Batch System (PBS). However, the gmice cluster experienced technical problems during the testing phase of the project. Therefore, gmice was only used for testing the OpenMP application on a single 8-core node.

The MPI and hybrid OpenMP/MPI applications were tested on the CDS cluster, which consists of 10 dual-core workstations. Since all workstations on the CDS cluster are available to multiple users at any time, the number of nodes used for a particular test depended on which nodes had significant CPU usage by other users.

3.3 Clustering Results

Results of clustering are shown in Figure 5. The figure shows the RGB channels for the original image, the cluster map after one iteration and the cluster map after 10 iterations. For the test image, the appearance of the cluster map after 1 and 10 iterations is not radically different. This is primarily due to the fact that the region covered by the image consists of a relatively small number of spectrally similar rocks and minerals. Figure 6 shows the number of pixels that are assigned to a new cluster, as a function of the k-means iteration number. Convergence of the clusters is apparent from the dwindling number of migrating pixels.

Figure 5: Results of k-means algorithm for Cuprite flight-line
3.4 Test Parameters

Performance of the parallel versions of the code were tested for the parameters in the table below. The hybrid MPI/OpenMP program is the same as the MPI program but with the OpenMP pragmas also enabled. In the case of the 512x614 image, it is actually the same Cuprite AVIRIS file; however, only the first 512 rows of data are read from the file. For all cases, the stopping criterion was to stop after 10 iterations.

Table 2: Testing Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Type</td>
<td>Serial, OpenMP, MPI, Hybrid MPI/OpenMP</td>
</tr>
<tr>
<td>Image Size</td>
<td>512x614 (140 MB), 2206x614 (607 MB)</td>
</tr>
<tr>
<td># Clusters (k)</td>
<td>10, 20</td>
</tr>
</tbody>
</table>

3.5 OpenMP

The OpenMP application was tested on the gnice.gmu.edu cluster using a single 8-core system. Figure 7 shows the execution time of the program as a function of number of threads (upper plot), as well as speedup (lower plot). Note that the data points for number of threads equal to one refers to execution time for the serial application. Speedup increases with increasing number of threads up until 8 threads, after which speedup declines. This is expected since the program was executed on an 8-core system, which means that with more than 8 threads, cores will have to run multiple threads, which will compete for cpu cycles and resources. For a given number of threads, speedup increased more by using more clusters than by increasing image size. This is likely due to the fact that increasing image size merely increases the total number of iterations in the parallelized loop, whereas increasing the number of clusters increases the amount of processing performed per parallel loop iteration, reducing the likelihood of threads being suspended due to locking shared resources.

3.6 MPI

The MPI application was tested on the CDS cluster, using up to 9 dual-core workstations. Performance is plotted in Figure 8. For the most computationally intensive case (2206x614 image with 20 clusters), speedup scales nearly
3.7 Hybrid OpenMP/MPI

The hybrid OpenMP/MPI application was tested on the CDS cluster using dual-threaded MPI processes. This was done because the CDS workstations have dual core CPUs. Thus, each CDS workstation will run no more than one MPI process with exactly two threads. Eight CDS nodes were used (as opposed to nine for the MPI test) because other nodes were executing CPU-intensive applications for other users. Note that in Figure 9, the x-axis indicates number of MPI nodes - the number of CPU cores used is twice that amount.

3.8 Comparison of Parallel Implementations

Figures 10 and 11 compare performance of the three parallelized applications for the greatest and least computationally intensive cases, respectively. It should be noted that the MPI and hybrid OpenMP/MPI applications were run on the CDS cluster, whereas the OpenMP application was run on the gmice cluster, which must be considered when comparing absolute speedup values. In the linear ranges of the speedup plots, MPI and hybrid OpenMP/MPI perform very similarly. Note also that the x-axis in these plots indicates cores used; thus, a value of 12 for MPI refers to 12 MPI processes run on 9 CDS nodes, whereas a value of 12 for hybrid OpenMP/MPI refers to 6 dual-threaded MPI processes run on 6 CDS nodes.

Note that in Figure 11, speedup for MPI begins to decrease with more than 6 cores and speedup for hybrid OpenMP/MPI begins to decrease with more than 12 cores. Both of these cases correspond to having more than 6
Figure 8: MPI performance on 9 dual-core CDS nodes

Figure 9: Hybrid MPI/OpenMP performance on 8 CDS Nodes
MPI processes running, which is consistent with a situation where inter-process communication begins to prevent additional speedup.

![Graph showing speedup of parallel algorithms for 2206x614 image with 20 clusters](image1)

**Figure 10:** Speedup of parallel algorithms for 2206x614 image with 20 clusters

![Graph showing speedup of parallel algorithms for 512x614 image with 10 clusters](image2)

**Figure 11:** Speedup of parallel algorithms for 512x614 image with 10 clusters

### 4 Conclusion

The tests performed clearly demonstrate that significant speedups can be achieved by using parallel implementations of the k-means algorithm for hyperspectral imagery. For a large file (608 MB), speedups of over 14 were achieved with MPI and dual-threaded MPI implementations of the algorithm. For a file closer to typical size (140 MB), a speedup of greater than 9 was achieved using dual-threaded MPI processes. For all cases evaluated, OpenMP never demonstrated a decrease in speedup with increased parallelism. The limitation on parallelism with OpenMP was that a system with only 8-cores was available to test OpenMP. For the typical file size, MPI yielded decreasing speedup when more than 6 MPI processes were used. With adequate resources available, it would be instructive to test the hybrid algorithm using 6 MPI processes across 6 8-core systems with 8 threads per MPI process to determine if speedup continues to increase up to 48 cores.
References