A High Performance Computing Approach to Nonnegative Matrix Factorization for Hyperspectral Data

Master's Project
Department of Computer Science
Montclair State University

Daniel Ricart

Fall 2012

Advisor: Dr. Robila

Table of Contents

Abstract

The ability to examine and extract the sources of data from hyperspectral images has become more and more important as the amount of data collected increases. Recent research has yielded better and better algorithms for un-mixing this data to provide more accuracy. One such algorithm is Nonnegative Matrix Factorization which aims to approximate the sources of the known end result. An issue with current approaches is they are designed to be run sequentially and can be very computationally expensive.

In this report, we examine ways of improving the performance of a known Nonnegative Matrix Factorization algorithm by utilizing parallelism over a cluster of computers. The goal of the project is to find ways of maximizing the throughput of a known algorithm without worrying about the accuracy of the algorithm itself (as this is shown through separate investigation). This is accomplished by testing out technologies such as MPI, POSIX threads and the OpenMP library. The aim is to compare and contrast different methods and find out what might be the optimal solution to allow for large data sets. The work on the project was completed on copou.csam.montclair.edu, a cluster available in the Department.

Acknowledgment

This work was supported by the National Science Foundation's MRI Program under Grant Number CNS-0922644. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

List of Figures

1. Introduction

The use of hyperspectral data has seen an increase in recent years as the amount of data captured has grown. Along with the increase in use comes an increase in the need to process and make sense of the data. Hyperspectral data is data collected from special sensors that take images of objects at different portions of the electro-magnetic spectrum[1]. Each pixel represents a spectral response to a band of light, indicating the materials that make up the objects surface. The end result is a set of images that gives scientists and others information about objects that are not available to the human eye. This data is commonly used for environmental monitoring, mineral exploration, vegetation mapping, water quality control and many others [2]. A major challenge, however, is determining the source components. Any given pixel can correspond to several different source materials of varying strengths.

To help solve the problem, several methods have been introduced over the years. The methods generally involve a process called linear unmixing which aims to take the observed values and decompose them in to sources that when combined would recreate the observed data. Common techniques for this include Independent Component Analysis, Principal Component Analysis and Nonnegative Matrix Factorization (NMF) [5]. These methods generally deal with approximating the sources as best as possible and each present specific advantages (in setting specific conditions for their models, in improving convergence speeds, etc.). Yet, no method has been provided as the preferred solution. As a result, we still have not reached the right level of confidence that we can fully recreate the sources for any given image.

1.1 Project Goal

While algorithmic accuracy is an important problem, another problem also exists. Regardless of the method used, scientists must deal with large data sets and find ways to process them quickly. Most implementations of unmixing have been done linearly which is often time consuming [4]. For any given algorithm, there would need to be a way to improve the process so that it can be applied on a much larger scale. As a result, we decided it would be beneficial to tackle this issue head on. We propose to pick a method of linear unmixing we thought would be suitable for speed up and looked at ways to make it more efficient. As algorithms improve in the future, it would be easy to adapt them to these ideas as well for a good combination of speed and accuracy.

To help achieve our project goal, we chose to design and implement Nonnegative Matrix Factorization for a High Performance Computing environment. The process employed included first, finding a good version of the algorithm to implement, and second, applying parallel computing to make it as fast as possible. The scope of this project included learning about hyperspectral data, learning about a parallel environment, and learning what technologies can be used in such an environment. As a result, this project covered the use of the cluster (copou.montclair.edu) located at Montclair State University along with a study of common libraries for both inter-node communication and individual processor speed up. This includes taking a look at MPI, POSIX threads and OpenMP.

The project did not aim to investigate the accuracy of an algorithm itself or what algorithm might be considered best as published results already exist in these areas [6]. Our goal instead was to examine performance and optimize it the best we can, The main challenge of this project was learning the technologies in a short amount of time and discovering how they can be used together. Using a combination of things might yield positive results and we hope overall the results will provide a good base for further experiments with clusters.

2. High Performance Computing

High performance computing is an area of computer science that involves techniques that allow faster and more efficient computations than what a standard computer processor is capable. The idea started in the 1960s when the first super computer was architected by Seymour Cray. His initial computer design used techniques such as time sharing of a single processor and using parallel program units. This allowed the computer to attempt parallel operations on a single processor. The next design evolved to introduce pipelining allowing different stages of execution to occur at once, thus resulting in large speedups [12]. Many other advances have occurred since, including faster memories and advanced techniques for creating faster processors with smaller transistors and faster clock speeds.

While individual computers have come a long way, the biggest gains, however, have come from utilizing multiple processors and cores at the same time in one system. These are commonly called clusters. Clusters combine the power of many simple processors to form a supercomputer capable of massive parallelism. Common examples of supercomputers like this include IBM's BlueGene (Fig. 1) and Cray's Titan [19] which bring together many thousands of core across many processors to form one machine. Adding to the shared power of the systems, individual computers involved can utilize their own parallelism through multi-threading or use of additional cores provided by graphics cards. The end result is that today's supercomputers are able to do what was once unthinkable thanks to modern HPC techniques. In the following, we discuss four HPC directions of high impact to improving applications' computational performance: Multi-core, multithreading, GPUs, and clusters.

2.1 Multi-core computing

An important advancement of the last decade has been the increased use of multi-core computing (see Fig. 2). Multi-core computing involves placing multiple processing units, called cores, on a CPU chip, allowing for hardware level parallelism. Creating CPUs with two or more cores allows the processor to run a task in each core completely independent of each other. This gives powerful performance gains in execution time. IBM first introduced a multi-core design with the PowerPC 970 in 2002 [18] . AMD and Intel soon followed with dual and then quad core computers which have since become the standard for household computers. Multiple core processors work by allowing each core to act like a single processor, capable of executing its own threads and having its own local memory. Intel's design has private level 1 caches and a shared level 2 cache and uses a bus controller to regulate the movement of data in memory. AMD processors use private L2 caches and a System Request Interface to allow sharing of data between processors. The end result of the multi-core designs is that programmers see a shared memory image [18]. Manufacturers continue to work to increase the number of cores supported for both personal use and business server use which has provided a big boost to the high performance computing sector.

Figure 1. Image of IBM BlueGene/P [19]

2.2 Multi-threading

A common technique used in high performance computing for individual processors is multi-threading. A thread is an independent stream of instructions that can be scheduled by an operating system (see Fig. 2a) [16]. Multi-threading involves allowing an operating system to run two or more threads at the same time. This can be done on the hardware level through redundant memory registers and other components or through software in the operating system itself. While multi-threading provides advantages even when used on single core systems, the introduction of multi-core has only increased its attractiveness. A system can take advantage of multiple cores to schedule its parallel tasks or even redundancies in single processors. Operations such as memory fetches can be done by one thread while another continues to push instructions though a pipeline. A core can then task switch between the workloads as best as it can to push the work through. On the software level, libraries such as POSIX Threads have been created to allow developers to easily take advantage of such capabilities [16].

2.3 GPUs

One of the most recent developments is the use of Graphics Processing Units. GPUs are processors contained on video graphics cards that handle the work of performing transformations and rendering pixels to a display. Originally, GPUs contained simple processors capable of doing math and basic operations for the video card. Modern GPUs are now equipped with processors that have hundreds or thousands of cores capable of a lot of parallelization. Each core is capable of running many threads in parallel meaning its computational capabilities can sometimes beat out that of CPUs. Software is used to send data and code to these cores so that algorithms can be run in parallel while the CPU continues to do other work [13].

The continued effort that companies have put in to GPU computing has led to several emerging possibilities for achieving parallelism. On the desktop consumer side, Nvidia's CUDA allows developers to take advantage of the many cores on NVidia's GPUs through a programming API. CUDA is both a platform and a progamming model that provides methods to directly access the memory of the video card and send work to as many or as few threads as you like. Combining CUDA with Nvidia's latest architecture called Kepler, cores can dynamically spawn threads and directly communicate with each without having to go back to the CPU[22]. Additionally, technologies like Microsoft's AMP C++

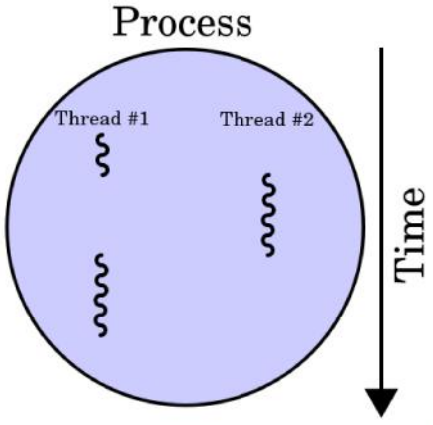


Figure 2a. Multithreading on a single process[20].

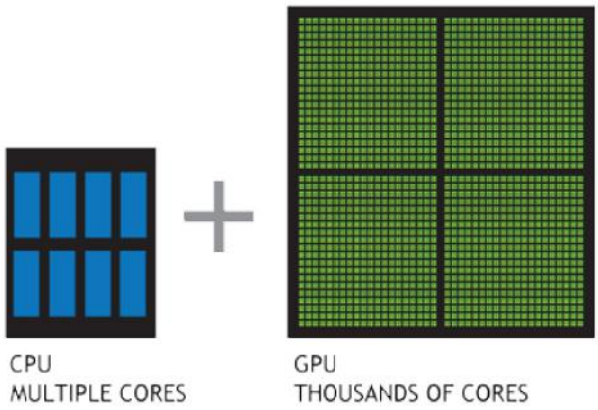


Figure 2b. Image of GPU cores from [13].

programming model are emerging to provide more device independent ways of running code on GPUs [24]. On the supercomputer side, both Nvidia and AMD continue to improve their high end offerings. Nvidia's latest Tesla GPUs offer up to 1.31 teraflops of double-precision peak performance while AMD's latest FirePro SM10000 deliver up to 1.48 teraflops of double-precision performance [23]. These cards are aimed at improving the performance of high end servers as both companies eye the cloud as the next place to make a big impact.

2.4 Clusters

While much has been done to increase the speed and efficiency of processors in individual machines, some of the most successful efforts have focused on bringing multiple computers together in to what is known as a cluster. Clusters are groups of computers brought together to act as a single computer for the purpose of some task or computation. They are attractive because they allow system builders to combine many machines built of cheaper components in to a system rather than one single more expensive system. The most popular type of cluster is the Beowulf cluster[14]. The project started with NASA in the early 90s from the requirement to build a supercomputer that cost under $50,000 [12]. The result is that Beowulf clusters are often built with commodity hardware that allows many cheap machines to come together to create a very powerful one. They are linked by network cards a common operating system, usually open source. As an example, IBM's BlueGene/L uses 65,536 nodes, each consisting of a 440 PowerPC and containing ports connecting to a 3-D torus interconnect for intra-node communication.

Most of the most powerful computers in use today are examples of clusters. Current examples of high end clusters can be seen on the top 500 list which lists the top 500 most powerful computers in the world [25]. Titan, which is a Cray XK7 system with 560,640 processors tops the most recent list. An IBM BlueGene/Q system called Sequoia from the Lawrence Livermore National Laboratory ranks second with an astounding 1.5 million cores. Two other machines in the top 5, Mira and JUQUUEEN, are also IBM BlueGene/Q machines [25]. Additionally, high end clusters have reached the cloud as Amazon's Elastic Compute Cloud service made the list. Their homegrown cluster runs on 17,024 cores and 66,000 GB of memory [26].

A practical example of a cluster and one that will be useful to this project is the one housed at Montclair State University. The cluster is composed of 65 nodes, one master and 64 slaves. Each node is running two 64 bit quad core AMD Opteron 2378's at 2.4GHZ and with 16GB of RAM. The cluster uses Red Hat Linux 5 and can be accessed via SSH [17] (See Fig. 3).

Figure 3. Image of the cluster from [17].

3. Introduction to Linear Unmixing

Linear unmixing assumes that the data we observe is formed as a linear mixing of original signals (often called sources) and is seeking a method to recover the sources. The same approach can be applied to the hyperspectral images, where the sources are abundances of the base materials for each pixel and the mixing is formed of the spectral signatures of the base materials.

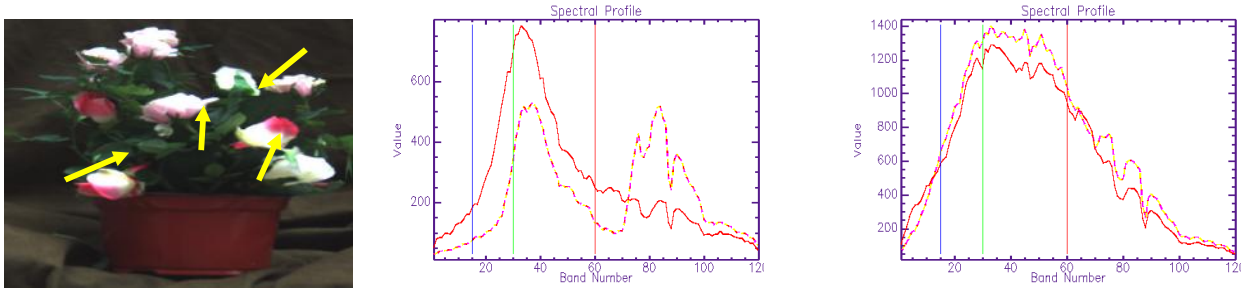
There are two types of methods that are currently in use. Supervised methods rely on prior knowledge about the reflectance patterns of the materials or domain knowledge and a series of semi-automatic steps. Supervised techniques involve a lot of user interaction and can easily lead to error. Unsupervised techniques on the other hand try to identify materials and mixtures directly from the observed data. When trying approaches like this, we typically have what is called Blind Source Separation problems (BSS) [2].

BSS means that the original components and their abundances are unknown. Several methods are currently used. One of the more basic is Independent Component Analysis. This method assumes that sources are considered to be statistically independent. This, however is limiting when using it on spectral data which does not meet that criteria. More recent methods have been developed targeting spectral data [10]. One such method is the N-FINDR method. This is a supervised, geometric pure pixel based algorithm. The algorithm works by randomly selecting a subset of pixels and evaluates them with the goal of refining the selected to find a group that maximizes the volume of the simplex defined by the selected pixels. The volume is calculated by replacing each pixel with a new one and calculating the new volume and when the volume increases, the new pixel is kept. The algorithm is repeated until there are no more replacements to be done [9].

Orthogonal subspace projection works by picking the pixel vector with the maximum length in a scene as the first endmember, then looks for the pixel vector with the maximum absolute projection in the space orthogonal to the space spanned by initial pixel for the second endmember. An orthogonal subspace projector is then applied to the original image where the signature has the maximum orthogonal projection in the space orthogonal to the space spanned by the first two endmembers forms the third endmember. This procedure is repeated until the desired number of endmembers is found [9].

Vertex component analysis makes use of the concept of OSP except it exploits the fact that the endmembers are the vertices of a simplex and that the affine transformation of a simplex is also a simplex. VCA models data using a positive cone, projecting it on to a hyperplane where the resulting simplex's vertices are the final endmembers. It projects all image pixels in a random direction and the pixel with the largest projection is the first endmember. It then iteratively projects data in a direction orthogonal to the subspace spanned by the endmembers already determined and the pixel with the extreme projection is the newest endmember [9].

Statistical methods also exist such as joint Bayesian endmember extraction and linear unmixing, the Bayesian analysis of spectral mixture data using Markov chain Monte Carlo methods or dependent component analysis. These methods are useful when geometric based methods return poor results because there are not enough spectral vectors in the simplex facets. These statistical methods make useful alternatives but have a higher computational complexity and could generally be slower [10].



a) b) c)

Figure 4. Hyperspectral image a) color composite image R-700nm, G-550nm, B-475nm, b) spectra of green vegetation (real (dashed) and artificial (solid)), c) spectra of flower (real (dashed) and artificial (solid)). Arrows show location of the spectra extraction. From [3].

4. Nonnegative Matrix Factorization

Nonnegative Matrix Factorization (NMF) is an un-mixing technique used to approximate the source matrices used in matrix multiplication. Given a matrix X, created from two matrices W and S representing the source materials and abundances we have:

$$X = W*S \tag{1}$$

where W and S are two non negative matrices. NMF involves minimizing the difference between x and W * s until we have reasonable approximations for W and S. This is done through the application of a gradient descent or multiplicative update that updates W and s repeatedly to bring us closer to the desired result. Unfortunately, NMF lacks a unique solution as different source matrices multiplied together could have the same end result. To help account for this, NMF uses constraints such as sparsity of the abundance matrix since most pixels are only comprised of a few source materials.

A typical NMF algorithm begins with an objective function to check the difference between the result and the source matrices. This is often done through a Euclidean Norm such as in [4]:

$$f(\mathbf{W}, \mathbf{s}) = \|\mathbf{x} - \mathbf{W}\mathbf{s}\|_F^2 \tag{2}$$

NMF algorithms generally begin by filling W and S with random data between 0 and 1. The function to minimize is then derived to get a gradient descent to ensure that the source matrices will converge toward the desired goal. This is accomplished by repeatedly updating the sources, W and s. This can be done through different approaches such as additive or multiplicative updates as described in [11].

5. NMF Algorithm with Sparsity Constraints

The algorithm we choose to implement is the L ½ Sparsity Constrained algorithm proposed by Y. Qian, and coauthors [6]. This algorithm is similar to the algorithm by Lee and Seung [11] but uses an L ½ sparsity constraint. In the implementation, we have source matrix X and randomized matrices A and S that we solve for. Using the L ½ regulizer, we get an objective function like this:

$$C(\mathbf{A}, \mathbf{S}) = \frac{1}{2}\|\mathbf{X} - \mathbf{A}\mathbf{S}\|_2^2 \tag{3}$$

Note that A and S correspond to W and s from the previous section. A sparsity constraint is then applied:

$$\mathcal{C}(\mathbf{A},\mathbf{S}) = \frac{1}{2}\|\mathbf{X}-\mathbf{AS}\|_2^2 + \lambda\|\mathbf{S}\|_{1/2} \tag{4}$$

Their sparsity function is then computed like this:

$$\|\mathbf{S}\|_{1/2} = \sum_{k,n=1}^{K,N} \mathbf{s}_n(k)^{1/2} \tag{5}$$

To minimize the above functions, multiplicative update algorithms were chosen. They are applied to achieve a gradient descent and look like this:

$$\begin{aligned} \mathbf{A} &\leftarrow \mathbf{A}.*\mathbf{X}\mathbf{S}^T./\mathbf{A}\mathbf{S}\mathbf{S}^T \\ \mathbf{S} &\leftarrow \mathbf{S}.*\mathbf{A}^T\mathbf{X}./(\mathbf{A}^T\mathbf{A}\mathbf{S}+\frac{\lambda}{2}\mathbf{S}^{-\frac{1}{2}}) \end{aligned} \tag{6, 7}$$

The final piece we need is to compute lambda to help with the sparsity constraint. This can be determined from the equation:

$$\lambda = \frac{1}{\sqrt{L}}\sum_l \frac{\sqrt{N}-\|\mathbf{x}_l\|_1/\|\mathbf{x}_l\|_2}{\sqrt{N}-1} \tag{8}$$

5.1 Implementation

To implement the algorithm, C code was used to write a program. Input data was supplied by Dr. Robila in the form of textual hyperspectral files. The foundation for the program is based on a program sample Dr. Robila created for the implementation of another NMF algorithm. The program starts by reading from the file in to matrix X where the rows are different bands and the columns are the pixels of each band. The program then allocates space for A and S where A is the number of bands times the number of pixels and S is the number of bands squared. The program then randomly fills in A and S and then normalizes S. An extra step also needs to be done to augment the X and A matrices (equation 9) with a constant value as one of their requirements which is done through simple loops.

$$\mathbf{X}_f = \begin{bmatrix} \mathbf{X} \\ \delta\mathbf{1}_N^T \end{bmatrix} \quad \mathbf{A}_f = \begin{bmatrix} \mathbf{A} \\ \delta\mathbf{1}_K^T \end{bmatrix} \tag{9}$$

These matrices are then used in the calculations to update S. Finally, we calculate the weight parameter lambda to use through the equation given in the previous section.

The matrix updates to minimize our objective function are done by looping a specified number of times, each time updating A and S and calculating the objective function at the end. We stop looping when we have looped the proper number of times and check our result to see that it headed toward

zero. The actual multiplication of matrices is done by looping through the bands and pixels to get the correct value for each point in the destination matrix. For example, to calculate A * S as part of the objective function, the result size would be bands by pixels so we loop through all the bands (rows) and then loop through all pixels (columns) for each row. To compute the value for each position, we loop through the bands again (this time representing columns of A) to get the proper source position of A (since it is of size bands * bands). We can multiply that value of A by value at the current position of S represented by the two inner loops as shown below.

```
for (i=0; i< bands+1; i++)
{
    for (int j=0;j<pixels; j++){
        sum = 0;
        for (int k=0; k< bands; k++){
            sum2 += A[i*bands+k]*s[k*pixels+j];
        }
        As[i*pixels+j]=sum;
    }
}
```

Many small matrix multiplication steps are carried out in each iteration to assist in the updates of A and S and this concept was repeated for each. For example, $X*S^T$ is used in the numerator of the calculation for updating the A matrix so it is calculated ahead of time using this looping scheme.

6. Parallel Implementation

6.1 MPI

With the basic algorithm written in C, we turned to the cluster at Montclair State described in Section 2 to run our program. The cluster is already set up and running and provided a stable platform for trying to improve performance. To actually utilize the machines available to us, we used the MPI library [15]. MPI stands for Message Passing Interface and is a language independent protocol for sending data between parallel machines. MPI uses a distributed memory programming model that allows for processes to be spread over more than one processor. The processes communicate with each other by sending messages across the processors or nodes they were distributed to. MPI programs are initialized to run a set number of instances by an external manager so they are automatically placed based on specified parameters. Each running instance of a program has its own address space so programs utilizing it need to be designed to operate on distributed data given to them locally [18].

Using MPI with C allows a program to spread its instances across several nodes of a cluster and allows each instance to send data to select other instances. It also implements operations such as Reduce for calculating single values from data spread across the nodes. MPI can send as many instances of a program to each node in a cluster as you want and the OS on each one will try its best to accommodate the request. If you have a cluster with two nodes, each with two quad core CPUs, you have eight cores per node and 16 cores total. You can have MPI run as many instances as you want but realistically only 16 will truly run in parallel.

To actually compile and run our application we made use of MPI's C++ compiler which allows for the compiling of basic C and C++ programs in addition to ones that use MPI's C library. With the original program in place we could now easily add MPI commands to our program to pass data between the instances of the program that would be running across the cluster.

The following code demonstrates a simple MPI C++ program.

```
int main(int argc, char *args[])
{
    int rank, size;
    rank = 0;
    size = 1;

    MPI::Init();
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    cout << "hello from node " << rank << " of " << size << endl;

    MPI_Finalize();
    return 0;
}
```

MPI programs are compiled with the mpicxx compiler (figure 5a). The compiler output is then run with the mpirun program (figure 5b), passing in the number of nodes to execute the program on, the name of the program and another arguments the program takes. We can attempt to give it as many nodes as we want. If you exceed the number of nodes in your cluster it will begin to pass multiple instances of the program to each node to make use of any parallelism on the node itself. The command also allows a user to specify which program to run. With this basic knowledge we were ready to modify the NMF algorithm to make use of MPI.

6.2 Parallelizing the Code

Using MPI allowed us to have fully parallel instances of the program running so each could operate on its own set of data. The objective then became how best to split the data so that the different nodes could calculate different sections of resulting matrices. The original source data and the calculated S matrix were large and seemed like good candidates. The way MPI splits data however made it awkward to use the source data matrix. There was no simple way to split down the rows or columns to evenly split multiplication duties. The S matrix, being randomly calculated on the other hand, was a great candidate. Each node could easily be given its own chunk of the S matrix without worrying about actually splitting the full matrix up in to equal rows or columns (since the data is random anyways). The small sub-matrices that each node would see are essentially the same as taking a fully created S matrix and rearranging the data in to smaller rectangles that can be broken apart. This is still accurate for all calculations because it is the S data we are modifying to seek the original source values.

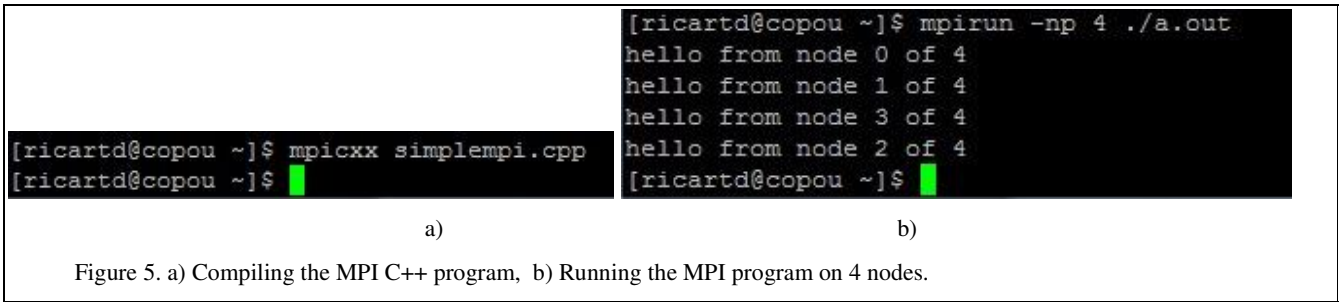


Figure 5. a) Compiling the MPI C++ program, b) Running the MPI program on 4 nodes.

With our idea set we were able to modify the program to support multiple nodes. When the program first starts, the main node loads the pixel data and creates the A matrix. It then passes those two full matrices to the other nodes. Next each node fills its own sub S matrix. To accomplish this, we divided the number of total pixels per image in to even chunks with the last chunk being a little larger than the rest if necessary. Each program instance calculates the chunk size and then creates its own small S matrix of chunk times bands and then normalizes it.

With S split up in to smaller sections, the calculations involving S performed by the loops in the original version no longer needed to loop through all columns of S. Each matrix calculation was altered to only loop through a subsection of the S matrix as demonstrated here:

```
for (i=0; i<bands+1; i++) {
    for (j=0; j<chunk; j++) {
        sum = 0;
        for (k=0; k<bands; k++) {
            sum += A[i*bands+k]*s[k*chunk+j];
        }
        As[i*chunk+j] = sum;
    }
}
```

For calculations where the main node needed to get the full end matrix, a reduce was done. A reduce is a parallel function where values from individual nodes are recursively compared against each other in a binary tree until one value is left at the top. A reduce can be done as a math function such as addition or subtraction or a simple comparison such as finding the maximum number. A reduce was used in the NMF program for cases when parts of a matrix were calculated in separate nodes and the final full matrix needed to be sent to all the nodes. The *reduce* function was used first to get the full matrix to the master node 0 such as in the following.

```
MPI::COMM_WORLD.Reduce(xst, xstreduce, bands*bands, MPI::DOUBLE,MPI::SUM,0);
```

After the reduce is done, the full matrix could then be sent to all nodes. This was handled with the broadcast command as demonstrated here.

```
MPI::COMM_WORLD.Bcast(A, bands*bands, MPI::DOUBLE, 0);
```

The last step was to parallelize the objective function calculation. Each node calculates its own value using its own section of the S matrix. The individual values were then summed together with a reduce and displayed for examination.

```
for (i=0; i< bands; i++) {
    objective[i] = 0;
    for (j=0; j<chunk; j++) {
        sum1 = 0;
        for (k=0; k< bands; k++){
            sum1 += A[i*bands+k]*s[k*chunk+j];
        }
        objective[i] += pow(data[i*pixels+startd+j]-sum1,2);
    }
}
```

6.3 Multi-threading

With the program running properly with MPI, we could explore adding a multi-threaded approach. The C++ compiler we were using supports multiple threads on a processor through the *Pthread* library. A thread is a block of code that runs inside a process but independent of other code blocks. By creating and running multiple threads at the same time, work can theoretically be done in parallel, as long as the blocks don't try to write to the same memory at the same time. The *Pthread* library is a C library implementing the POSIX thread standard [16]. The library was designed to speed up the creation of threads and has a host of ready made functions and structure for parallelizing code. Running code in a thread involves creating an instance of a thread structure and indicating it what function to execute as demonstrated here.

```
int rc;
void *status;
for (index=0; index<totalThreads; index++) {
    tInfo[index].index = index;
    tInfo[index].targetFunction = function;
    rc = pthread_create(&thread[index],&attr,compute,(void*)&tInfo[index]);
}
//join the threads
for (index=0; index<totalThreads; index++) {
    rc = pthread_join(thread[index], &status);
}
```

Even though MPI spreads instances of our program to individual processors in the cluster, we also wanted to investigate if those processors could see a gain from running on multiple threads. One limitation of the calculations we were performing was that just about all of them were dependent on the previous one being completed. As a result, we could not try task parallelism and run two different calculations in parallel. The other natural thought then was to chunk more of the looping across threads. Using the outer loops was the only route that made sense as placing them in inner loops meant excessive thread creation and joining which proves to be slow. If an outer loop was looping through all the bands, it could be chunked so that each thread would tackle a portion of the work and only calculate on part of the final matrix. Since the final result needed to be shared across the threads of a given node, the matrices that stored the calculation results were put in a container that was global with respect to the overall running instance. In the end, any source or destination component matrix or value used in a calculation was read from and written to the global structure.

To allow for the ease of throttling of threads, the calculations were placed in a special calculation function called by a wrapper that was in charge of creating and destroying the threads. The wrapper function took in the function type which was passed to threaded function that performed all the calculation logic for each step. Each calculation was placed in a code block that determines what chunk of work perform based on the threads index. The end code is similar to the original calculation except the outer loop was now chunked. For matrix calculations, since all threads were accessing the same global structure, there was no need for any further steps. When computing a single value such the final objective function however, the results of the threads had to quickly be summed together by the wrapper before it could be returned and reduced by the main thread code. An example of a threaded matrix function for computing A times local s is shown here.

```
int bandsPerThread = (int)ceil((double)(bands+1) / (double)totalThreads);
int startBand = index*bandsPerThread;
int endBand = startBand+bandsPerThread;
```

```
if (endBand > bands + 1)
    endBand = bands + 1;

for (i=startBand; i<endBand; i++) {
    for (j=0; j<chunk; j++) {
        double sum = 0;
        for (k=0; k<bands; k++) {
            sum += A[i*bands+k]*mys[k*chunk+j];
        }
        As[i*chunk+j] = sum;
    }
}
```

6.4 OpenMP

After testing that Pthreads provided a viable solution, we decided to try adding OpenMP to the code to compare the performance. OpenMP is an application programming interface designed to provide a shared memory model for parallel applications. It is defined by a group of hardware and software vendors and is supported by C/C++ and Fortran. The API is designed as a mix of compiler directives and runtime library calls. It was intended to provide a simple and standard way to provide parallelism for programs with as little code as possible. It does not guarantee that it will make the most efficient use of the shared memory but does make it much easier to achieve the model [21].

In use, OpenMP is similar to using Pthreads where memory is shared between the running threads. This contrasts with MPI which lets each process have its own memory and its own section of data. Using OpenMP can be accomplished by adding pragma statements to existing code. The statements can be used to specify sections of code that should be parallelized including loops or specific code blocks and control thread synchronizations. The following is an example of parallelizing a simple loop with an OpenMP pragma compiler directive.

```
int i;
#pragma omp parallel for
for (i=0; i<16; i++) {
    int id = omp_get_thread_num();
    #pragma omp critical
    cout << "hello from thread " << id << endl;
}
```

Compiling a program with OpenMP is done by linking to the library when compiling (figure 6). A nice side effect of using OpenMP is that it is that it is enabled and disabled by either linking or not linking the program to the library. No code changes are needed. As a result, the project could support both POSIX threads and OpenMP in one program without adding any special logic to the code.

To implement OpenMP in the code, we parallelized all the matrix calculation loops to split the work in the same way it was split with Pthreads. The only realistic performance gains were by parallelizing the outer loops and not the inner loops to avoid the thread joining from slowing down the program. A consideration that had to be made was to ensure that the variables and sums would be independent of each other in each thread so there would be no crosstalk. This was done by adding the private keyword

```
[ricartd@copou ~]$ mpicxx -fopenmp ricartmpi.cpp
[ricartd@copou ~]$
```

Figure 6. Compiling a simple OpenMP C++ program.

when defining the pragma statement. Here is an example of multiplying the A matrix by the local s matrix shown earlier but now with OpenMP.

```
#pragma omp parallel for private(i,j,k,sum2)
for (i=0; i< bands+1; i++)
{
    for (int j=0;j<chunk; j++){
        sum2 = 0;
        for (int k=0; k< bands; k++){
            sum2 += A[i*bands+k]*s[k*chunk+j];
        }
        As[i*chunk+j]=sum2;
    }
}
```

One issue with parallelizing sequential loops is that when computing a single sum, the program needs to ensure that the threads are not competing with each other to update the value at the same time. In OpenMP, this is handled by specifying a statement as atomic. The atomic keyword tells the computer that an incremental operation needs to be done that cannot be interfered with. This is effective for single line statements as it protects the underlying value without having to lock and unlock a section of code. The following is an example of computing the sum of the square roots in parallel.

```
#pragma omp parallel for private(i,j) shared(ssquare)
for (i=0; i<bands; i++) {
    for (j=0; j<chunk; j++) {
        #pragma omp atomic
        ssquare += sqrt(s[i*chunk + j]);
    }
}
```

7. Experiments

Prior to starting the experiments we repeatedly ran our code to ensure that the parallelization of the algorithm was functioning correctly. This meant that an objective function and sparsity was computed successively for each image band with the results getting smaller each iteration approaching 0. Since this project is focused on getting performance gains, the results and efficiency of the algorithm are not discussed here. The data used for the experiment was a 30 band set of hyperspectral data. Each band is a 640x640 image for a total of 409,600 pixels per band. The cluster used for the experiments is made up of 65 nodes, each with two AMD Opteron 2378's. These are quad core processors running at 2.4 GHZ. This gave us about 520 cores to use for our testing.

7.1 MPI

The first tests done were to see how MPI could speed things up on the cluster. We picked an iteration count of 50 for the calculations and performed it against a 30 band set of data. We felt this was a good starting point that took long enough to see nice speed gains but not so long that the smaller node count test runs would drag on for hours. The tests were run up to five times in a row to get consistent times which were then averaged together. Keeping the data set constant, we went from 1 node to 2 to 4 and kept doubling the number all the way up to 512. A slow down could be seen when adding the first node due to the extra communication cost counteracting the speed up due to the data split. With 4 or more nodes however the results improved. After 64 nodes, the times were too quick and did not improve

anymore since they were so quick. To counter this we did an additional experiment with more bands.

Despite the penalty for only two nodes, dividing up the work proved extremely efficient for yielding better times. Up to 64 nodes, a significant gain was observed for each doubling (Figure 7a). To examine the results further, we calculated the speed up for each increase in nodes. This is done by taking the original sequential time and dividing by each parallel time.

$$\text{Speedup} = T_{\text{sequential}} / T_{\text{parallel}} \quad (10)$$

The speed up results show a fairly linear increase in the gain we obtained as the node counts were doubled (Figure 7b). The next issue to tackle was testing for the higher number of nodes since they finished too quickly in the 50 iteration test. The new test we devised involved using a 60 band data sample provided by Dr. Robila and doing 200 iterations against it. The results showed the that there is an additional increase in speed gained from going up to 256 nodes (figure 8). Once past that mark we see efficiency drop as the individual processors get burdened with more work. Once nodes start to receive more than one execution of the program, the operating system has to manage how to spread the processes across itself. MPI also has to manage how to send the information around for the broadcast and reduce functions. At 128 nodes there would be at best one process running per processor in the cluster since each node has two processors. Past that point, the extra work across the cores will start to contend for resources more and more. Even when running against 64 nodes, some nodes could potentially be given more than one copy of a program to run and others nothing. Additionally, the overhead from performing broadcasts and reduces across the cluster grows with each node increase. The main theme of the results, however, was that they were mostly at the mercy of how well each node could schedule its given processes.

7.2 Threads

The next test to perform was to see how adding POSIX threads would improve or hurt performance. To do this, tests of 1, 2, 4, 8 and 16 threads were performed for each number of nodes tested in the previous section with 50 iterations done over 30 bands of data. The results show nice gains initially for 64 or fewer nodes but only up to 8 threads (figure 9a). After 8 there seemed to be no benefit, likely because the processors could not truly do more than that in parallel. Again we see that after 64 nodes, the tests ran too fast to provide useful data. While the speed ups obtained by threads were impressive, for any given number of nodes, the performance increase is less than what we get from increasing the

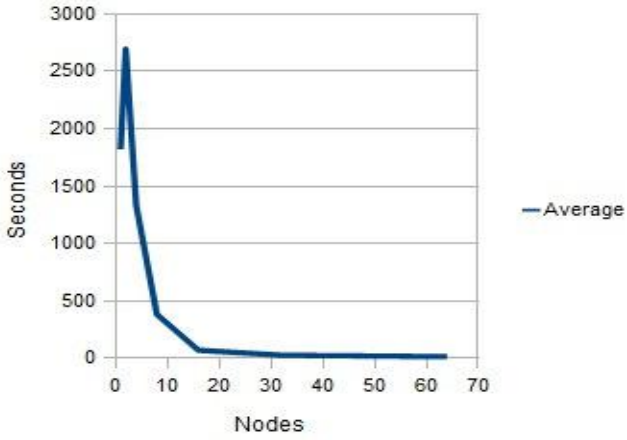


Figure 7a. Computation result times vs. number of nodes. 50 runs were done against 30 bands of images.

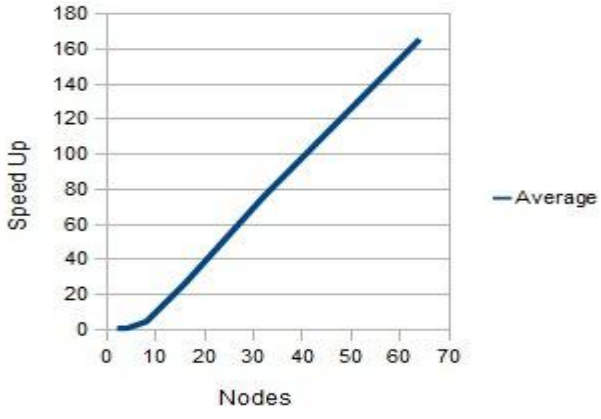


Figure 7b. MPI Speed up achieved per number of nodes

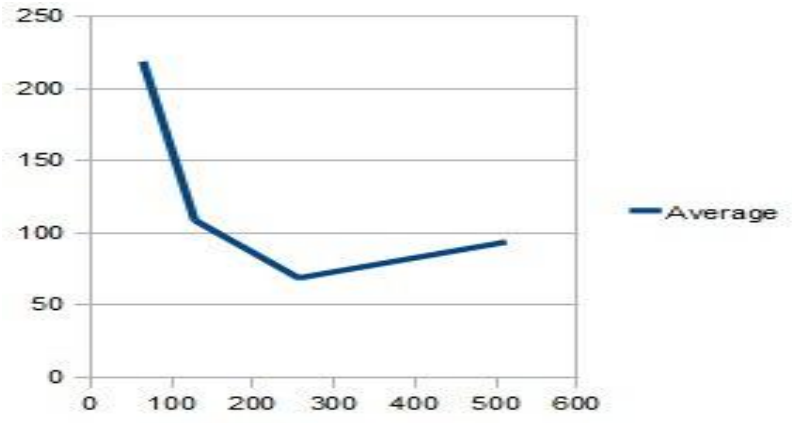


Figure 8. MPI results for 64 to 512 nodes with 60 bands of data.

number of nodes. Dividing up the data appears to be the stronger standalone strategy.

The next step was to compute the speed up of using threads for each node count. The results show that the speed up is roughly linear up to 8 threads and then little gain is seen after (figure 9b). This is due to the fact that each node has two processors for a total of 8 cores. The operating system will try to run as many threads as you tell it but the node can only truly do 8 processes in parallel. The results show benefits but are not as steep as the speedups gained from increasing the number of nodes alone.

Finally we tested the upper range of nodes against the 60 band data sample. The results continued the trend we had seen of nice gains being brought by adding threads, even when we tried to fully allocate the cluster's cores (figure 10). By adding up to 8 threads we almost halved some of the times. On 256 and 512 nodes however we started to see less of an increase. This was largely due to the load each processor in the cluster was already performing. Adding extra threads created more contention among the tasks on the node since MPI already gave them multiple tasks. It was interesting to see though that even 512 nodes could benefit from a little threading despite the fact it was slower than 256 nodes without threads.

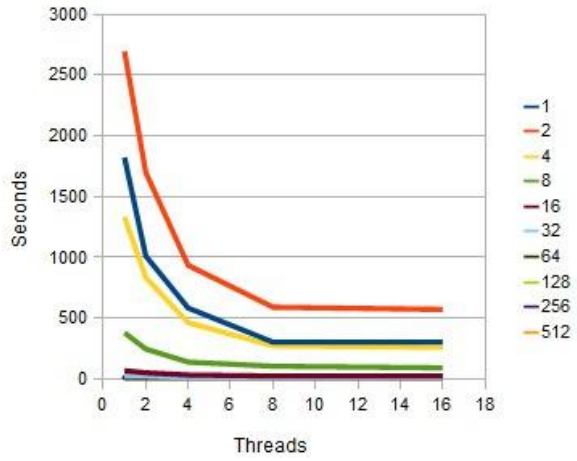


Figure 9a. Result times for increasing numbers of Pthreads. Node counts are represented in different colors.

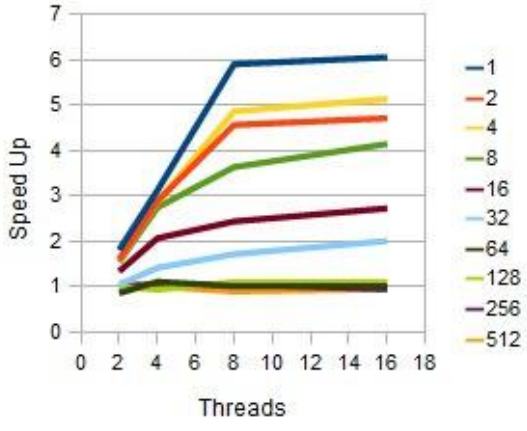


Figure 9b. Speed up results for nodes per thread count increase. Different node counts are represented in different colors.

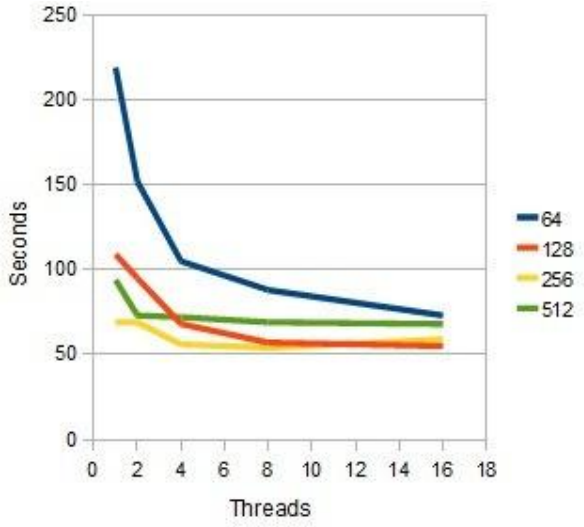


Figure 10. Result times for node counts 64 to 512 for 60 bands of images.

7.3 OpenMP

The last test was to see how OpenMP with MPI compared to everything done before. Switching on OpenMP was as simple as re-compiling the code with the OpenMP library flag set as described in section 6. This test worked similarly to the threading test where it was combined with MPI and tested with 2, 4, 8 or 16 threads per node. The results were similarly impressive to those of the POSIX threads, however, the overall times lagged in comparison (figure 11a). OpenMP's internal scheme for scheduling threads was a little slower than the custom routines used here, perhaps due to inefficiencies with the compiler or my own inexperience with best practices for using the library. Despite the interesting performance, the major limiter turned out to be the fact that MPI would not run with OpenMP on more than 16 nodes. Any attempts at spreading it out more resulting in MPI simply freezing the whole program and not attempting any runs. The speedup graph (figure 11b) shows the positive improvements from threading but reveals a lower speed up factor than the *Pthread* library.

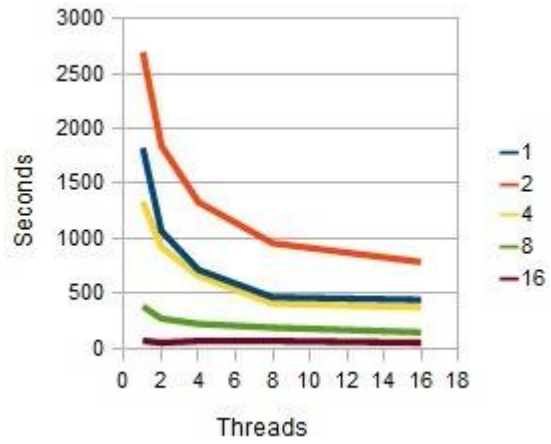


Figure 11a. OpenMP results for thread and node counts 1 to 16. Each color represents the times for a specific node count.

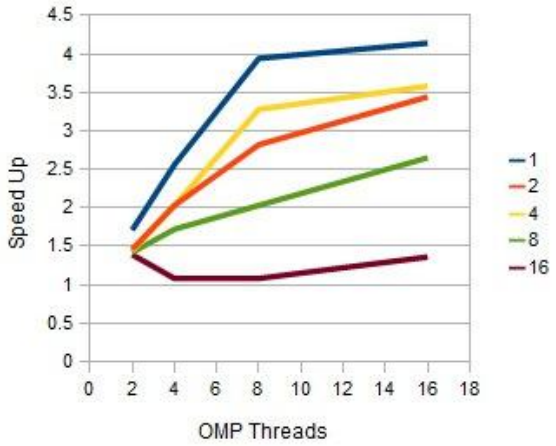


Figure 11b. Speed up results for OpenMP with increasing thread counts. Each color represents a specific node count.

8. Conclusions

This project yielded positive results on the whole for speeding up normally linear calculations such as Nonnegative Matrix Factorization. By utilizing MPI, we were able to divide up the work normally done by one processor in to many smaller pieces. Since the calculations involved matrices and had many loops involved, there was ample opportunity for enhancement. Even when allocating more processes than the 65 nodes available to us, we still saw gains up to 256 nodes. Once the work was divided up across different nodes in the cluster and performing nicely, we were able to further investigate how the work could be sped up on each individual node.

In addition to chunking with MPI, additional gains were found both by using custom *Pthreads* and by using OpenMP pragma statements to create the threads. This appears to be likely due to the multi-core nature of the processors on each node. When nodes were not fully allocated by MPI, such as using one process per node, the best gains were seen. Even with 512 nodes allocated, some amount of speed up was still seen by using *Pthreads* although it was a smaller gain than when it was used with fewer nodes. Unfortunately we were unable to get OpenMP to run on more than 16 nodes so full comparisons could not be made. For the runs it did work with, it provided adequate gains although it did not achieve the same speedup as the *Pthreads* scheme.

The main trend observed in the experiment was that the more processes we sent to a node, the more the potential for speed ups diminished. Any processor will be limited in how efficiently it can handle all the processes and threads thrown at it. Another point to make is that the threading scheme used was most likely not the most efficient one. Even better gains could be achieved in the future with a smart thread scheduling algorithm. Despite this, it was still impressive to see how much improvement can be found by maximizing the workload for each node.

While the overall result seems to indicate it is useful to mix threads with MPI, the best individual results definitely came from using MPI by itself. When scaling from 1 node upward while keeping it to 1 thread per node, the speed ups jumped from below 1 for the first increase to over 4, then 26 then 75 and up to a gain of 165. This meant that it ran in one hundred and sixty fifth of the original time with 64 nodes. This contrasts with testing thread count increases by itself on just one node where the gain only reached as high as 6, meaning it ran in one sixth the time. OpenMP similarly only gave a maximum speed up of about 4 to 4.5. The tests also proved that MPI's distribution of work to the individual servers to be passed on to the cores is a little more efficient than our scheme of trying to use threads on each node while the job spans all nodes. Results from using 256 nodes from MPI for the 60 band run gave slightly better results that using 64 nodes with 2, 4 or 8 threads.

In conclusion, which strategy to use for a cluster could depend on the exact task at hand and the characteristics of the cluster. For programs able to utilize the full list nodes, MPI by itself may prove to distribute the work to the servers in the most optimal way. When a smaller number of nodes makes sense, threads or OpenMP can be added for additional gains. As new advances are made in the ability to perform parallel tasks on individual machines these gains should become even greater.

8.1 Future Work

In addition to what was attempted here, the project could be extended to use CUDA or OpenCL to take advantage of CPUs and GPUs in parallel. CUDA is a library for C from Nvidia which allows

developers to take advantage of the many cores available on modern video cards. Adding the ability to make calculations on the GPU could easily enhance the gains already obtained from using OpenMP or Pthreads. Each individual node would be able to accomplish even more in parallel as the GPU work of a given node could be done without interruption from its processor. Even more promising is Nvidia's latest GPU architecture called Kepler that allows GPU cores to interact with each other and the CPU directly, allowing for extra flexibility of creating threads and allocating memory. It additionally allows the work distributed by a library like MPI to run directly on the GPU core and not be limited to just CPUs. This strategy unfortunately requires the cluster to have GPUs on each node but might be attainable on a smaller experimental cluster built by a student or faculty member.

References

[1] N. Keshava and J. F. Mustard, “Spectral unmixing,” *IEEE Signal Processing Magazine*, 2002, vol. 19, no. 1, pp. 44-57.

[2] P. Sajda, S. Du, and L. Parra, “Recovery of constituent spectra using non-negative matrix factorization,” in *Proceedings of SPIE*, 2003, vol. 5207, pp. 321-331.

[3] S. A. Robila, M. Chang, and N. B. D’Amico, “Face recognition using spectral and spatial information,” in *Proceedings of SPIE*, 2011, vol. 8135, p. 81351Q.

[4] S. Robila, “Linear Unmixing Based Feature Extraction for Hyperspectral Data in a High Performance Computing Environment,” in *Proceedings of SPIE Volume 8515*, 2012, p. 7 pgs.

[5] S. A. Robila and M. Butler, “Parallel unmixing of hyperspectral data using complexity pursuit,” in *Geoscience and Remote Sensing Symposium (IGARSS), 2010 IEEE International*, 2010, pp. 1035-1038.

[6] Y. Qian, S. Jia, J. Zhou, and A. Robles-Kelly, “Hyperspectral Unmixing via $ L_ {1/2} $ Sparsity-Constrained Nonnegative Matrix Factorization,” *Geoscience and Remote Sensing, IEEE Transactions on*, no. 99,, 2011, pp. 1-16.

[7] S. A. Robila and G. Busardo, “Hyperspectral Data Processing in a High Performance Computing Environment: A Parallel Best Band Selection Algorithm,” in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 1424-1431.

[8] S. A. Robila and L. G. Maciak, “Considerations on parallelizing nonnegative matrix factorization for hyperspectral data unmixing,” *Geoscience and Remote Sensing Letters, IEEE*, 2009, vol. 6, no. 1, pp. 57-61.

[9] A. Plaza, G. Martin, J. Plaza, M. Zortea, S. Sanchez. “Recent Developments in Endmember Extraction and Spectral Unmixing,” in *Optical Remote Sensing. Augmented Vision and Reality*, 2011. Volume 3, 235-267.

[10] J. Bioucas-Dias and A. Plaza. “An Overview of Hyperspectral Unmixing: Geometrical, Statistical, and Sparse Regression Based Approaches,” in *Geoscience and Remote Sensing Symposium (IGARSS), 2011 IEEE Internal*. 24-29 July 2011. Pages 1135-1138.

[11] D. Seung and L. Lee. “Algorithms for non-negative matrix factorization,” *Advances in neural information processing systems*, 2001, vol 13, 556-562.

[12] G. Bell. “A Brief History of Supercomputing: "the Crays", Clusters and Beowulfs, Centers. What Next?”. [Online]. http://research.microsoft.com/en-us/um/people/gbell/supers/supercomputing-a_brief_history_1965_2002.htm, Accessed November 14, 2012.

[13] NVidia. “What is GPU Computing”. [Online]. http://www.nvidia.com/object/what-is-gpu-computing.html, Accessed November 14, 2012.

[14] Wikipedia. Beowulf (computing). [Online]. http://en.wikipedia.org/wiki/Beowulf_cluster.,

Accessed November 14, 2012.

[15] B. Barney, Lawrence Livermore National Laboratory. “Message Passing Interface (MPI)”. [Online]. https://computing.llnl.gov/tutorials/mpi/, Accessed November 15, 2012.

[16] B. Barney, Lawrence Livermore National Laboratory. “POSIX Threads Programming”. [Online] https://computing.llnl.gov/tutorials/pthreads/, Accessed November 15, 2012.

[17] G. R. Busardo. "Procurement, Benchmarking and Usage of a New High Performance Computing Cluster", 2010, MS Project, Montclair State University. October 2010.

[18] C. Lin and L. Snyder. *Principles of Parallel Programming*, Addison Wesley, 2008

[19] Wikipedia. Supercomputer. [Online]. http://en.wikipedia.org/wiki/Supercomputer, Accessed November 15, 2012.

[20] Wikipedia. Multithreading. [Online]. http://en.wikipedia.org/wiki/File:Multithreaded_process.svg, Accessed Nov 30, 2012

[21] B. Barney, Lawrence Livermore National Laboratory. “OpenMP”. [Online]. https://computing.llnl.gov/tutorials/openMP/, Accessed December 3rd, 2012.

[22] Nvidia. “Nvidia Kepler Computer Architecture”. [Online] http://www.nvidia.com/object/nvidia-kepler.html. Accessed December 3rd, 2012.

[23] A.Kingsley-Hughes. “Nvidia and AMD unveil new supercomputer GPUs”. http://www.zdnet.com/nvidia-and-amd-unveil-new-supercomputer-gpus-7000007301/. Accessed December 3rd, 2012.

[24] Microsoft. “C++ AMP (C++ Accelerated Massive Parallelism)” [Online]. http://msdn.microsoft.com/en-us/library/hh265137.aspx. Accessed December 3rd, 2012.

[25] Top 500. “Top 500 Supercomputer Sites | November 2012”. [Online]. http://www.top500.org/lists/2012/11/. Accessed December 4th, 2012.

[26] J. Brodkin. “Amazon's cloud is the world's 42 fastest supercomputer.” [Online]. http://arstechnica.com/business/2011/11/amazons-cloud-is-the-worlds-42nd-fastest-supercomputer/. Accessed December 4th, 2012.

Appendix I – Compiling and Running on the Cluster

Compiling

To compile the program, we simple use the mpicxx command. This command can compile any C++ file whether it uses MPI in the code or not.

```
[ricartd@copou ~]$ mpicxx ricartmpi.cpp
[ricartd@copou ~]$
```

Compiling the program to use OpenMP involves giving the compiler the flag for the openmp library.

```
[ricartd@copou ~]$ mpicxx -fopenmp ricartmpi.cpp
[ricartd@copou ~]$
```

Running

Mpirun is used to execute programs. It takes the number of nodes, program name and arguments. The following runs 16 nodes and tells our program to iterate 10 times while using default image data.

```
[ricartd@copou ~]$ mpirun -np 16 ./a.out 10
Running 16 nodes!
Data to be read :FaceSOCRobila Quart Band
Dimensions :640*640*30
Each band has  409600 elements
No threading used
iterating 10 times
Round: 1 Objective: 1.58879e+08 sparsity: 29375.3
Round: 2 Objective: 5.30716e+07 sparsity: 24330.8
Round: 3 Objective: 5.18997e+07 sparsity: 22091
Round: 4 Objective: 4.6926e+07 sparsity: 20585.5
Round: 5 Objective: 4.1978e+07 sparsity: 19314
Round: 6 Objective: 3.77211e+07 sparsity: 18146.5
Round: 7 Objective: 3.43159e+07 sparsity: 17055.4
Round: 8 Objective: 3.18468e+07 sparsity: 16060.1
Round: 9 Objective: 3.00919e+07 sparsity: 15192.1
Round: 10 Objective: 2.87595e+07 sparsity: 14459
time: 17 seconds
```

The next example shows specifying the number of threads to run and the filename to use for the data. In this example we are running 10 times over 16 nodes with 4 threads and one of Robila's sample files.

```
[ricartd@copou ~]$ mpirun -np 16 ./a.out 10 4 SOCFull2_0.txt
Running 16 nodes!
Data to be read :FaceSOCRobila Quart Band
Dimensions :640*640*30
Each band has  409600 elements
Using 4 Pthreads
iterating 10 times
Round: 1 Objective: 1.58879e+08 sparsity: 29375.3
Round: 2 Objective: 5.30716e+07 sparsity: 24330.8
Round: 3 Objective: 5.18997e+07 sparsity: 22091
Round: 4 Objective: 4.6926e+07 sparsity: 20585.5
Round: 5 Objective: 4.1978e+07 sparsity: 19314
Round: 6 Objective: 3.77211e+07 sparsity: 18146.5
Round: 7 Objective: 3.43159e+07 sparsity: 17055.4
Round: 8 Objective: 3.18468e+07 sparsity: 16060.1
Round: 9 Objective: 3.00919e+07 sparsity: 15192.1
Round: 10 Objective: 2.87595e+07 sparsity: 14459
time: 15 seconds
```

Appendix II – Source Code

```
/*
File: ricartmpi.cpp

Nonnegative Matrix Factorization in Parallel
By Dan Ricart
Montclair State University
Master Project
Dr. Robila
*/

#include <iostream>
#include <istream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
#include <string>
#include <ctime>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include <omp.h>
#include "mpi.h"

#define endMembers 60

using namespace std;

enum TargetFunction {
  SSquareFunction,
  SparsityFunction,
  AsFunction,
  AtxFunction,
  xstAsstFunction,
  mysFunction,
  lambdaFunction,
  objectiveFunction
};

//structure to hold calculation data to be shared across threads
typedef struct {
  int *data;
  double *A;
  double *mys;
  double *xst;
  double *Asst;
  double *As;
  double *Atx;
  double *AtAs;
  double *xstreduce;
  double *Asstreduce;
  double lambda;
  double ssquaretotal;
  int pixels;
  int bands;
  int chunk;
  int startd;
} dataHolder;
```

```
//create an instance of the data holder
dataHolder holder;

//structure to store information to pass to threads
struct threadInfo {
  int index;
  double sum[60];
  TargetFunction targetFunction;
};

struct threadInfo threadInfos[16];
int totalThreads = 4;
pthread_t thread[16];
pthread_mutex_t parallelMutex;

//declarations for functions defined below
void NMF_Parallel(double delta, int timesToIterate, int useOpenMP);
void *computeInThread(void *threadArg);
double *computeCalcWithThreads(TargetFunction function, pthread_attr_t attr);

//entry poiint for the application
int main(int argc, char*args[])
{
    //flag to signal that something is wrong
    int shouldend = false;
    int val;
    int i,j;
    int provided;

    ifstream myfile;

    //filename can be provided as argument or as default value
    char *filename;

    //default times to iterate
    int timesToIterate = 30;
    int rows, cols, bands, pixels;
    string datadesc;
    double delta = 1.0;
    int useOpenMP = 0;

    int buflen = 512;
    char name[512];

    //get the name of this node
    gethostname(name, buflen);

    //MPI info
    int rank, size;
    rank = 0;
    size = 1;

    //initialize MPI to work with threading
    MPI_Init_thread(0, 0, MPI_THREAD_FUNNELED, &provided);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    //if we are the master node, check the arguments passed in to set defaults
```

```
if (rank == 0){
    cout << "Running "<<size<<" nodes!"<<endl;

    //basic file handling
    switch (argc){
        case 2:
            //user passed in # times to iterate only
            //use no threads and default 30 band file
            timesToIterate = atoi(args[1]);
            totalThreads = 0;
            filename = "SOCFull2_0.txt";
            break;
        case 3:
            //user passed in # times to iterate and the total threads to spawn
            //default file is used
            timesToIterate = atoi(args[1]);
            totalThreads = atoi(args[2]);
            filename = "SOCFull2_0.txt";
            break;
        case 4:
            //user passed in # times to iterate, total threads and the filename
            timesToIterate = atoi(args[1]);
            totalThreads = atoi(args[2]);
            filename = args[3];
            break;
        case 1:
            //no arguments passed, use all defaults
            totalThreads = 0;
            filename = "SOCFull2_0.txt";
            break;
        default:
            cout << "Incorrect number of arguments at entry... Exiting"<<endl;
         return 0;
    }

    //open the file for reading
    myfile.open (filename);

    if (myfile.is_open())
    {
        //read first three values
        myfile >> rows;
        myfile >> cols;
        myfile >> bands;

        getline(myfile,datadesc);
        getline(myfile,datadesc);

        cout << "Data to be read :"<<datadesc.c_str() << endl;
        cout << "Dimensions :" <<rows<<"*"<<cols<<"*"<<bands<<endl;

        //compute the pixels
        pixels = rows*cols;
        holder.data = new int[(bands+1)*pixels];
        if (holder.data == NULL){
            cout << "Failed memory allocation for bands. Done!\n";
            return 0;
        }
```

```
        cout<<"Each band has  "<<pixels<<" elements \n";

        for (i=0;i<bands*pixels;i++){
            myfile >> val;
            if (myfile.eof()){
                cout<<"Premature file end at :"<< i << "," << j;
                cout<< ". Exiting!\n";
                return 0;
            } else {
                holder.data[i]=val;
            }
        }
        myfile.close();
    }
    else
    {
        cout << "File "<<filename<<" could not be openned.\n";
        cout << "You may want to check its name. Read not performed.\n";
        shouldend = true;
    }

    //augment our data with one extra row of values to satisfy one of the
    //concerns in our algorithm
    for (i=0; i<pixels; i++)
    {
        holder.data[bands*pixels + i] = (int)delta;
    }

    //now create the A matrix with random values
    holder.A = new double [(bands+1)*bands];
    for (i=0; i<bands*bands; i++) {
        holder.A[i] = rand()/(float)RAND_MAX;
    }

    //augment the A matrix with an extra row of constant data
    for (i=0; i<bands; i++) {
        holder.A[bands*bands + i] = delta;
    }

} //end of rank = 0

//now initialize our constants that control how work is done across the nodes
//broadcast the total pixels and bands we read in and store them
MPI::COMM_WORLD.Bcast(&pixels, 1, MPI::INT, 0);
MPI::COMM_WORLD.Bcast(&bands, 1, MPI::INT, 0);
holder.pixels = pixels;
holder.bands = bands;

//compute our chunk size based on the number of nodes used
int chunk = pixels / size;
int startd = rank*chunk;
if (rank == size-1)
{
    if (startd+chunk<pixels)
    {
        chunk = pixels - startd;
    }
}
holder.chunk = chunk;
```

```
holder.startd = startd;
holder.mys = new double [chunk*bands];

//fill our s matrix with random data
for (i=0; i<bands*chunk; i++) {
    holder.mys[i]= rand()/(double)RAND_MAX;
}

//normalize the newly created s matrix
for (i=0; i<chunk; i++) {
    double sum = 0;
    for (j=0; j<bands; j++) {
      sum+= holder.mys[j*chunk+i];
    }
    for (j=0; j<bands; j++) {
      holder.mys[j*chunk+i]/=sum;
    }
}

//start the timer
time_t start;
time_t stop;
time(&start);
MPI::COMM_WORLD.Bcast(&timesToIterate, 1, MPI::INT, 0);
MPI::COMM_WORLD.Bcast(&totalThreads, 1, MPI::INT, 0);
MPI::COMM_WORLD.Bcast(&useOpenMP, 1, MPI::INT, 0);

//check if we compiled with openmp, if so tell it how many threads to use
#ifdef _OPENMP
    useOpenMP = 1;
    if (totalThreads == 0) {
        totalThreads = 1;
    }
    omp_set_num_threads(totalThreads);
#endif

//allocate data for our matrices if we are not the master node
if (rank > 0)
{
    holder.data = new int[(bands+1)*pixels];
    holder.A = new double[(bands+1)*bands];
}

//broadcast the full A and data matrices to all other nodes
MPI::COMM_WORLD.Bcast(holder.data, (bands+1)*pixels, MPI::INT, 0);
MPI::COMM_WORLD.Bcast(holder.A, (bands+1)*bands, MPI::DOUBLE, 0);

//Something is Wrong, Bye!
if (shouldend){
    MPI::Finalize();
    return 0;
}

//now we are ready to run our parallel computations
NMF_Parallel(delta, timesToIterate, useOpenMP);

//now compute how long the function took to run
if (rank == 0) {
    time(&stop);
```

```
        double diff = difftime(stop, start);
        cout << "time: " << diff << " seconds" << endl;
    }

    //finalize MPI and stop our threads
    MPI_Finalize();
    pthread_mutex_destroy(&parallelMutex);
    pthread_exit(NULL);
    return 0;
}

//this function hosts the logic and calculation steps for performing Nonnegative
//Matrix Factorization. the main loop will iterate as many times as indicated,
//performing the individual calculations for each loop using threads or openmp
void NMF_Parallel(double delta, int timesToIterate, int useOpenMP)
{
    int bands = holder.bands;
    int pixels = holder.pixels;
    int chunk = holder.chunk;
    int startd = holder.startd;

    pthread_t thread[16];
    pthread_attr_t attr;

    //create a thread attribute to allow threads to be joinable
    //this means that we can retrieve the data it was working on once finished
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    pthread_mutex_init(&parallelMutex, NULL);
    int rc;
    void *status;
    int i, j, k;
    int rounds;
    double epsilon = 0.0000001;
    double suma;

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    //display information about our chosen parallel scheme
    if (rank ==0) {
        if (useOpenMP > 0) {
            cout << "Using OpenMP with " << totalThreads << " threads" << endl;
        }
        else if (totalThreads > 0) {
            cout << "Using " << totalThreads << " Pthreads" << endl;
        }
        else {
            cout << "No threading used" << endl;
        }
    }

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        cout << "iterating " << timesToIterate << " times" << endl;
    }
```

```
    //allocate space for the temporary matrices we use to perform calculations
    holder.Atx = new double [bands*pixels]; //used to update s
    holder.As  = new double [(bands+1)*pixels]; //used to update A and s
    holder.xst = new double [(bands+1)*bands]; //used to update A
    holder.Asst= new double [(bands+1)*bands]; //used to update A
    holder.AtAs= new double [bands*pixels]; //used to update s

    //allocate space for our matrices to store reduction results
    holder.xstreduce = new double [bands*bands]; //used to update A
    holder.Asstreduce = new double [(bands+1)*bands]; //used to update A

    double sum1, sum2, sum3, sum4, sum5;

    //compute our lambda to use for the sparsity function
    if ((useOpenMP == 0) && (totalThreads > 0)) {
        //compute sum with pthreads
        double *lambdaSum = computeCalcWithThreads(lambdaFunction, attr);
        holder.lambda = lambdaSum[0] / sqrt((double)bands);
    }
    else {
        //we are using openmp or no threads
        double lambdaSum = 0;
        #pragma omp parallel for private (i,j)
        for (i=0; i< bands; i++)
        {
            double xl1temp = 0;
            double xl2temp = 0;
            for (j=0; j < pixels; j++)
            {
                double current = holder.data[i*pixels + j];
                xl1temp += current;
                xl2temp += current * current;
            }
            double xl1 = xl1temp;
            double xl2 = sqrt(xl2temp);
            double xdiv = xl1 / xl2;
            double numerator = sqrt((double)pixels) - xdiv;
            double denominator = sqrt((double)pixels) - 1;
            lambdaSum += (numerator / denominator);
        }
        holder.lambda = lambdaSum / sqrt((double)bands);
    }
    double oldObjective = 0;

    //loop as many times as the program tells us
    for (rounds = 1; rounds <=timesToIterate; rounds ++) {

        //if we are not using openmp and we have threads specified,
        //do calculations with pthreads
        if ((useOpenMP == 0) && (totalThreads > 0)) {
            //compute A*S with threads
            computeCalcWithThreads(AsFunction, attr);
            //compute x * sT and the AS result from above times sT
            computeCalcWithThreads(xstAsstFunction, attr);

            //reduce our calculations so node 0 gets the whole matrices
            MPI::COMM_WORLD.Reduce(holder.xst, holder.xstreduce, bands*bands,
MPI::DOUBLE,MPI::SUM,0);
```

```
            MPI::COMM_WORLD.Reduce(holder.Asst, holder.Asstreduce, bands*bands,
MPI::DOUBLE,MPI::SUM,0);

            //if we are the master node, update A quickly
            if (rank == 0) {
                for (i=0; i<bands; i++) {
                    for (j=0; j<bands; j++) {
                        holder.A[i*bands+j] *= holder.xstreduce[i*bands+j]/
(holder.Asstreduce[i*bands+j]);
                    }
                }
                for (i=0; i<bands; i++) {
                    holder.A[bands*bands + i] = delta;
                }
            }
            //now broadcast A to all nodes
            MPI::COMM_WORLD.Bcast(holder.A, (bands+1)*bands, MPI::DOUBLE, 0);

            //compute At * x with threads
            computeCalcWithThreads(AtxFunction, attr);
            //compute A*S again with threads
            computeCalcWithThreads(AsFunction, attr);

            //compute the square root of all values in s to use for sparsity
            double *ssquares = computeCalcWithThreads(SSquareFunction, attr);
            //reduce the computed values so they are all summed together
            //then node 0 will have the final answr
            MPI::COMM_WORLD.Reduce(&ssquares[0], &holder.ssquaretotal, 1,
MPI::DOUBLE, MPI::SUM, 0);
            //broadcast the final value back out to all nodes
            MPI::COMM_WORLD.Bcast(&holder.ssquaretotal, 1, MPI::DOUBLE, 0);

            //update the local s matrix with threads
            computeCalcWithThreads(mysFunction, attr);

            //compute our objective function using threads
            double allobjective[endMembers];
            double *objective = computeCalcWithThreads(objectiveFunction, attr);

            //reduce the sum computed for each band on each node
            //this gives the full value to node 0
            for (i=0; i<bands; i++) {
                MPI::COMM_WORLD.Reduce(&objective[i], &allobjective[i], 1,
MPI::DOUBLE,MPI::SUM,0);
                allobjective[i] *= 0.5;
            }

            //compute the sparsity for each band
            double allsparsity[endMembers];
            double *sparsity = computeCalcWithThreads(SparsityFunction, attr);
            //reduce the sparsity value calculated on each node so the
           //sum is given to node 0
            MPI::COMM_WORLD.Reduce(&sparsity[0], &allsparsity[0], 1,
MPI::DOUBLE,MPI::SUM,0);

            //display the result for the first band only
            if (rank == 0) {
                allsparsity[0] *= holder.lambda;
                cout << "Round: " << rounds << " Objective: " << allobjective[0] <<
```

```
" sparsity: " << allsparsity[0] << endl;
            }
        }
        else {
            //compute using a single thread or openmp
            //openmp pragmas are ignored when not compiled with openmp
            //this allows us to turn openmp on or off through compilaton

            //compute A*S with openmp or single thread
            #pragma omp parallel for private(i,j,k,sum2)
            for (i=0; i< bands+1; i++)
            {
                for (int j=0;j<chunk; j++){
                    sum2 = 0;
                    for (int k=0; k< bands; k++){
                        sum2 += holder.A[i*bands+k]*holder.mys[k*chunk+j];
                    }
                    holder.As[i*chunk+j]=sum2;
                }
            }

            //compute x*sT and AS from above times sT using openmp or single thread
            #pragma omp parallel for private(i,j,k,sum3, sum4)
            for (i=0; i< bands+1; i++){
                for (j=0; j<bands; j++){
                    sum3 = 0; sum4 = 0;
                    for (k=0; k<chunk; k++) {
                        sum3 +=
holder.data[i*pixels+startd+k]*holder.mys[j*chunk+k];
                        sum4 += holder.As[i*chunk+k]*holder.mys[j*chunk+k];
                    }
                    holder.xst[i*bands+j]=sum3;
                    holder.Asst[i*bands+j]=sum4;
                }
            }

            //reduce the results so the full matrix is possessed by the main node
            //reduces should always be called by main thread
            MPI::COMM_WORLD.Reduce(holder.xst, holder.xstreduce, bands*bands,
MPI::DOUBLE,MPI::SUM,0);
            MPI::COMM_WORLD.Reduce(holder.Asst, holder.Asstreduce, bands*bands,
MPI::DOUBLE,MPI::SUM,0);

            //if we are the first node, update A
            if (rank==0)
            {
                for (i=0; i<bands; i++) {
                    for (j=0; j<bands; j++) {
                        holder.A[i*bands+j] *= holder.xstreduce[i*bands+j]/
(holder.Asstreduce[i*bands+j]);
                    }
                }
                //redo the last augmented row of A
                for (i=0; i<bands; i++) {
                    holder.A[bands*bands + i] = delta;
                }
            }

            //broadcast the A matrix we updated to everyone else
```

```
            MPI::COMM_WORLD.Bcast(holder.A, (bands+1)*bands, MPI::DOUBLE, 0);

            //compute At times x using openmp or a single thread
            #pragma omp parallel for private(i,j,k)
            for (i=0; i< bands; i++) {
                for (j=0; j<chunk; j++) {
                    double sum = 0;
                    for (k=0; k< bands+1; k++) {
                        sum += holder.A[k*bands+i]*holder.data[k*pixels+startd+j];
                    }
                    holder.Atx[i*chunk+j] = sum;
                }
            }

            //compute A time s again using openmp or a single thread
            #pragma omp parallel for private(i,j,k,sum1, sum2)
            for (i=0; i< bands+1; i++) {
                for (j=0; j<chunk; j++) {
                    sum1 = 0; sum2 = 0;
                    for (k=0; k< bands; k++){
                        sum2 += holder.A[i*bands+k]*holder.mys[k*chunk+j];
                    }
                    holder.As[i*chunk+j]=sum2;
                }
            }

            //add the sqaure roots of all elements of s
            double ssquare = 0;

            #pragma omp parallel for private(i,j) shared(ssquare)
            for (i=0; i<bands; i++)
            {
                for (j=0; j<chunk; j++)
                {
                    #pragma omp atomic
                    ssquare += sqrt(holder.mys[i*chunk + j]);
                }
            }

            MPI::COMM_WORLD.Reduce(&ssquare, &holder.ssquaretotal, 1, MPI::DOUBLE,
MPI::SUM, 0);
            MPI::COMM_WORLD.Bcast(&holder.ssquaretotal, 1, MPI::DOUBLE, 0);

            // update S = S*AT*X / AT * A * S + lambda /2 * S-1/2
            #pragma omp parallel for private(i,j,k,sum5)
            for (i=0; i< bands; i++)
            {
                for (j=0; j<chunk; j++){
                    sum5 = 0;
                    for (k=0; k< bands+1; k++) {
                        sum5 += holder.A[k*bands+i]*holder.As[k*chunk+j];
                    }
                    holder.mys[i*chunk+j] *= holder.Atx[i*chunk+j]/(sum5+
(holder.lambda*holder.ssquaretotal/2));
                }
            }


            double objective[endMembers];
```

```
            double allobjective[endMembers];

            //compute our final objective function
            #pragma omp parallel for private(i,j,k,sum1)
            for (i=0; i< bands; i++) {
                objective[i] = 0;
                allobjective[i] = 0;

                for (j=0; j<chunk; j++) {
                    sum1 = 0;
                    for (k=0; k< bands; k++){
                        sum1 += holder.A[i*bands+k]*holder.mys[k*chunk+j];
                    }
                    objective[i] += pow(holder.data[i*pixels+startd+j]-sum1,2);
                }
            }

            //perform a reduce from the main node so he gets the sum of all values
            for (i=0; i<bands; i++)
            {
                MPI::COMM_WORLD.Reduce(&objective[i], &allobjective[i], 1,
MPI::DOUBLE,MPI::SUM,0);
                allobjective[i] *= 0.5;
            }

            double sparsity[endMembers];
            double allsparsity[endMembers];

            //now compute the sparsity
            #pragma omp parallel for private(i,j)
            for (i =0; i< bands; i++) {
                sparsity[i] = 0;
                for (j=0; j<chunk; j++) {
                    double sq = sqrt(holder.mys[i*chunk + j]);
                    sparsity[i] += sq;
                }
            }

            //perform a reduce to get the total sum sparsity in the main node
            for (i=0; i<bands; i++) {
                MPI::COMM_WORLD.Reduce(&sparsity[i], &allsparsity[i], 1,
MPI::DOUBLE,MPI::SUM,0);
                allsparsity[i] *= holder.lambda;
            }

            //if we are the master, display the results of the calculations
            if (rank == 0) {
                double sqrtot = 0.0;
                for (i=0;i<chunk;i++) {
                     double ss = sqrt(holder.mys[i]);
                     sqrtot = (double)(sqrtot + ss);
                }
                cout << "Round: " << rounds << " Objective: " << allobjective[0] <<
" sparsity: " << allsparsity[0] << endl;
            }
        } //end if not basic threading
    }

    //now cleanup everything
```

```
    delete[] holder.AtAs;
    delete[] holder.Asst;
    delete[] holder.xst;
    delete[] holder.As;
    delete[] holder.Atx;

    delete[] holder.Asstreduce;
    delete[] holder.xstreduce;

    delete[] holder.A;
    delete[] holder.mys;
    delete[] holder.data;

    pthread_attr_destroy(&attr);
}

//this function wraps our thread calling and joining
//this allows the main algorithm to simply pass an enum value specifying
//the function we want to run
double *computeCalcWithThreads(TargetFunction function, pthread_attr_t attr)
{
    int index;
    int index2;
    double *sum = new double[endMembers];
    for (index=0;index<holder.bands;index++)
    {
       sum[index] = 0;
    }
    int rc;
    void *status;

    for (index=0; index<totalThreads; index++) {
        threadInfos[index].index = index;
        threadInfos[index].targetFunction = function;
        rc = pthread_create(&thread[index], &attr, computeInThread,(void
*)&threadInfos[index]);
    }
    for (index=0; index<totalThreads; index++) {
        rc = pthread_join(thread[index], &status);
        for (index2=0; index2<holder.bands; index2++) {
          double val = sum[index2];
          val += threadInfos[index].sum[index2];
          sum[index2] = val;
        }
    }
    return sum;
}

//This is the function that contains our calculation logic
//it is called from the thread creation call in the computeCalcWithThreads function
//It always executes inside a thread. The main idea behind these computations is
//that it further breaks up the already chunked work done based on which thread
//index you are and how many threads are currently running
void *computeInThread(void *threadArg)
{
    //first read our structure from the passed in value to get our index value
    struct threadInfo *info;
    info = (struct threadInfo*)threadArg;
    int index = info->index;
```

```
int pixels = holder.pixels;
int startd = holder.startd;
int bands = holder.bands;
int chunk = holder.chunk;
int i, j, k;
info->sum[0] = 0;

//now check which computation we want to perform
//computations here feature two levels of chunking
//the inner loop is chunked based on our node
//the outer loop is chunked based on which thread we are
if (info->targetFunction == SSquareFunction) {
    //determine our part of the band data to tackle based on our thread index
    int bandsPerThread = (int)ceil((double)bands / (double)totalThreads);
    int startBand = index*bandsPerThread;
    int endBand = startBand+bandsPerThread;
    if (endBand > bands)
        endBand = bands;
    //compute the square roots of s using thread chunking in the outer loop
    //and node chunking in the inner loop
    for (i=startBand; i<endBand; i++) {
        for (j=0; j<chunk; j++) {
            info->sum[0] += sqrt(holder.mys[i*chunk + j]);
        }
    }
}
else if (info->targetFunction == SparsityFunction) {
    //determine our part of the band data to tackle based on our thread index
    int bandsPerThread = (int)ceil((double)bands / (double)totalThreads);
    int startBand = index*bandsPerThread;
    int endBand = startBand+bandsPerThread;
    if (endBand > bands)
        endBand = bands;
    //compute the sparsity by looping through this thread's chunk of bands
    for (i=startBand; i<endBand; i++) {
        info->sum[i] = 0;
        for (j=0; j<chunk; j++) {
            info->sum[i] += sqrt(holder.mys[i*chunk + j]);
        }
    }
}
else if (info->targetFunction == AsFunction) {
    //determine our part of the band data to tackle based on our thread index
    int bandsPerThread = (int)ceil((double)(bands+1) / (double)totalThreads);
    int startBand = index*bandsPerThread;
    int endBand = startBand+bandsPerThread;
    if (endBand > bands + 1)
        endBand = bands + 1;
    //compute A*S using thread chunking in the outer loop
    //and node chunking (middle loop)
    for (i=startBand; i<endBand; i++) {
        for (j=0; j<chunk; j++) {
            double sum = 0;
            for (k=0; k<bands; k++) {
                sum += holder.A[i*bands+k]*holder.mys[k*chunk+j];
            }
            holder.As[i*chunk+j] = sum;
        }
    }
```

```
    }
    else if (info->targetFunction == xstAsstFunction) {
        //determine our part of the band data to tackle based on our thread index
        int bandsPerThread = (int)ceil((double)(bands+1) / (double)totalThreads);
        int startBand = index*bandsPerThread;
        int endBand = startBand+bandsPerThread;
        if (endBand > bands+1)
            endBand = bands+1;
        //compute xst and Asst using thread chunking in the outer loop
        //and node chunking in the inner loop
        for (i=startBand; i<endBand; i++) {
            for (j=0; j<bands; j++) {
                double sum3 = 0;
                double sum4 = 0;
                for (k=0; k<chunk; k++) {
                    sum3 += holder.data[i*pixels+startd+k]*holder.mys[j*chunk+k];
                    sum4 += holder.As[i*chunk+k]*holder.mys[j*chunk+k];
                }
                holder.xst[i*bands+j]=sum3;
                holder.Asst[i*bands+j]=sum4;
            }
        }
    }
    else if (info->targetFunction == AtxFunction) {
        //determine our part of the band data to tackle based on our thread index
        int bandsPerThread = (int)ceil((double)(bands) / (double)totalThreads);
        int startBand = index*bandsPerThread;
        int endBand = startBand+bandsPerThread;
        if (endBand > bands)
            endBand = bands;
        //compute At * x using thread chunking in the outer loop
        //and node chunking in the middle loop
        for (i=startBand; i<endBand; i++) {
            for (j=0; j<chunk; j++) {
                double sum = 0;
                for (k=0; k<bands+1; k++) {
                    sum += holder.A[k*bands+i]*holder.data[k*pixels+startd+j];
                }
                holder.Atx[i*chunk+j] = sum;
            }
        }
    }
    else if (info->targetFunction == mysFunction) {
        //determine our part of the band data to tackle based on our thread index
        int bandsPerThread = (int)ceil((double)bands / (double)totalThreads);
        int startBand = index*bandsPerThread;
        int endBand = startBand+bandsPerThread;
        if (endBand > bands)
            endBand = bands;
        //update s using thread chunking in the outer loop
        //and node chunking in the middle loop
        for (i=startBand; i<endBand; i++) {
            for (j=0; j<chunk; j++) {
                double sum = 0;
                for (k=0; k<bands+1; k++) {
                    sum += holder.A[k*bands+i]*holder.As[k*chunk+j];
                }
                holder.mys[i*chunk+j] *= holder.Atx[i*chunk+j]/(sum +
((holder.lambda/2)*holder.ssquaretotal) );
```

```
            }
        }
    }
    else if (info->targetFunction == lambdaFunction) {
        //determine our part of the band data to tackle based on our thread index
        int bandsPerThread = (int)ceil((double)bands / (double)totalThreads);
        int startBand = index*bandsPerThread;
        int endBand = startBand+bandsPerThread;
        if (endBand > bands)
            endBand = bands;
        //compute our starting lambda value using threads
        for (i=startBand; i<endBand; i++) {
            double xl1temp = 0;
            double xl2temp = 0;
            for (j=0; j<pixels; j++) {
                double current = holder.data[i*pixels + j];
                xl1temp += current;
                xl2temp += current * current;
            }
            double xl1 = xl1temp;
            double xl2 = sqrt(xl2temp);
            double xdiv = xl1 / xl2;
            double numerator = sqrt((double)pixels) - xdiv;
            double denominator = sqrt((double)pixels) - 1;
            info->sum[0] += (numerator / denominator);
        }
    }
    else if (info->targetFunction == objectiveFunction) {
        //determine our part of the band data to tackle based on our thread index
        int bandsPerThread = (int)ceil((double)bands / (double)totalThreads);
        int startBand = index*bandsPerThread;
        int endBand = startBand+bandsPerThread;
        if (endBand > bands)
            endBand = bands;
        //compute objective using thread chunking in the outer loop
        //and node chunking in the middle loop
        for (i=startBand; i<endBand; i++) {
            info->sum[i] = 0;
            for (j=0; j<chunk; j++) {
                double sum = 0;
                for (k=0; k<bands; k++) {
                    sum += holder.A[i*bands+k]*holder.mys[k*chunk+j];
                }
                info->sum[i] += pow(holder.data[i*pixels+startd+j] - sum, 2);
            }
        }
    }
    pthread_exit(NULL);
}
```