PARALLELIZATION OF HYPERSPECTRAL IMAGING CLASSIFICATION AND DIMENSIONALITY REDUCTION ALGORITHMS

By

Wilfredo E. Lugo-Beauchamp

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER ENGINEERING

University of Puerto Rico
Mayagüez Campus
December 2004

Approved by:

Shawn Hunt, Ph.D. Member, Graduate Committee	Date
Jaime Seguel, Ph.D. Member, Graduate Committee	Date
Wilson Rivera, Ph.D. President, Graduate Committee	Date
Pedro Vásquez, Ph.D. Representative of Graduate Studies	Date
Isidoro Couvertier, Ph.D. Chairperson of the Department	Date

ABSTRACT

PARALLELIZATION OF HYPERSPECTRAL IMAGING CLASSIFICATION AND DIMENSIONALITY REDUCTION ALGORITHMS

By

Wilfredo E. Lugo-Beauchamp

Hyperspectral imaging provides the capability to identify and classify materials remotely. The applications of such technology is applied everywhere from medical devices and military targets to environmental sciences. With the ongoing advances in spectrometers (spatial resolution and bits per pixel density) the data gathered is constantly increasing. Some hyperspectral imaging algorithms could easily take days or weeks in analyzing a full single hyperspectral data set. In this thesis we performed a porting and parallelization of four hyperspectral algorithms representative of the type of analysis done in a typical data set. Two of the algorithms are in the area of data classification, one in the area of feature reduction and the other one is a combination of both areas. The parallelized algorithms were benchmarked on the Intel 32 bits Pentium M architecture and the new Intel 64 bits Itanium 2 architecture. For three of the four algorithms we demonstrated that the use of parallel approaches in combination with computational clusters speedup significantly the executions times and provide great scalability. On the other algorithm, based on linear algebra manipulations using distributed objects, we obtained execution times that took longer than the sequential implementation. A systematic performance analysis is carried out to explain the performance behavior of the algorithms.

RESUMEN

PARALELIZACION DE ALGORITMOS DE CLASIFICACION Y DE REDUCCION DE DIMENSIONALIDAD DE IMAGENES HIPER-ESPECTRALES

Por

Wilfredo E. Lugo-Beauchamp

La capacidad de analizar imágenes hiper-espectrales provee la habilidad de indentificar y clasificar materiales remotamente. Las aplicaciones de este tipo de tecnología tiene aplicaciones en un gran sinúmero de áreas que van desde aparatos médicos y objectivos militares a ciencias ambientales. Debido a los continuos avances en los sensores espectrales (resolución espacial y en la cantidad de bits en un pixel) la cantidad de data recojida está aumentando constantemente. Algoritmos hiper-espectrales pueden tomar días e incluso semanas en analizar todas las bandas de una muestra. Como parte de esta tesis portamos y paralelizamos 4 algoritmos hiper-espectrales representativos del tipo de analísis efectuado en una imagen hiper-espectral comunmente. Dos de los algoritmos son basados en classificadores, uno en el area de redución de bandas y el restante es una combinación de ambas areas. Los algoritmos paralelizados fueron probados en las arquitecturas de Intel Pentium M (32 bits) e Intel Itanium 2 (64 bits). En tres de los cuatro algoritmos quedó demostrado que la paralelización de los algoritmos proveen tiempos de ejecución mucho mas rápidos y con una gran escalabilidad. En el algoritmo restante, basado en manipulaciones de algebra lineal y objectos distribuídos, los tiempos de ejecución resultaron ser mayores que los de la implementación secuencial. Un análisis sistemático de eficiencia es llevado a cabo para explicar el comportamiento de crecimiento computacional de los algoritmos.

To Lisie.....

I stole so much time from you to finish this, thus it is more yours than mine.

TE AMO!

ACKNOWLEDGMENTS

There have been a lot of people that have helped me in so many ways throughout all my life. It will be almost impossible to acknowledge everybody, but I would feel bad if I don't mention these special people. First of all and most important, I want to thank my mother Nidia; I don't know how you did it, but you did it. Being a single mom is not easy today and sure it was not easy at the time we were growing up. You raised three children alone and under difficult economic conditions; thanks for everything and I am still learning from you. To my Dad, Enrique, I barely remember you living in the same house with Mom, but I can't think of any moment in my life that you were not there for me, thanks. Lisie, my love, thanks for everything and for your support. I know I am not an easy person to deal with but with you I feel complete. Isaac and Kiara, my little prince and princess, I love you with all my heart, thanks for changing my life. Dr. Gerson Beauchamp, thanks for triggering my college interests in the engineering area. The pre-engineering camp changed my life. Dr. José Luis Cruz, thanks for convincing me to pursue graduate studies, there are very few persons I could consider role models, you are one of them. There are also two very special friends that helped me a lot during this journey, they were always there for me with their friendship and help, Michelle and Gunther, I owe you two a lot. I also want to thanks my co-worker Carlos J. Félix, thanks for all the moments of unconditional debugging help and great ideas. You are one of the brightest persons I have met. It has been a pleasure working with you these past years. I also want to thanks Hewlett Packard Puerto Rico management. Once I came with the idea to join efforts between my graduate work and hp interests they did not hesitate to agree. Manuel Martinez and Luis López: without your support this task would have been almost impossible to achieve. Last but not least, Dr. Wilson Rivera, when I was trying to get enrolled back to finish my graduate studies, I wrote to more than 10 professors and you were the only one who answered. I only hope I have not disappointed you, I will always be in debt with you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Overview

Hyperspectral imaging allows a spatial scene to be decomposed into multiple two-dimensional images obtained at different spectral bands (Figure 1.1). These images can then be analyzed to discriminate among different features within the scene.

Processing techniques generally identify the presence of materials through measurement of spectral absorption features. Today's image processing systems incorporate a frame buffer that captures an image in memory. Then a commercial DSP microprocessor processes the image sequentially.

	AVIRIS [JPL] (typical sample)	TRWIS [TRW]	HYDICE [Hughes] (typical sample)	Future sensor array
Image resolution (pixels)	614 x 512	512 x 512	320 x 240	1000 x 1000
*Dynamic range (bits/pixel)	12	12	12	12
Spectral Bands	224	384	210	200
Image Size (MB)	105	150.9	24.1	300

Table 1.1: Data bandwidth for a selection of sensor arrays[30]. *Before atmospheric correction

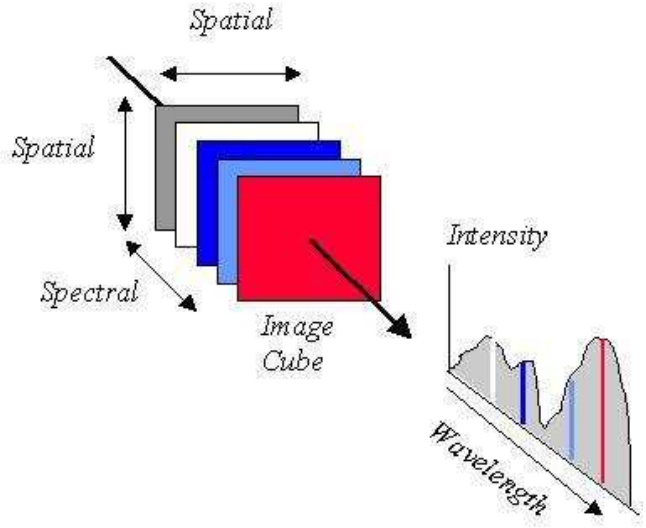

Figure 1.1: Multiple images in different spectral bands form an image cube for the same spatial image. Spatial and spectral analysis are performed on the image cube to obtain chromatic, textural, and regional information.

With the rapid advances in the resolution, frame rate, and dynamic range of spectrometers, the required bandwidth has exceed throughput limits inherent in store and process systems. Table 1.1 points out existing and future hyperspectral sensor arrays. Hyperspectral algorithms require more operations per pixel, and thus, higher processing throughput is necessary.

The main issues on hyperspectral imaging are concentrated on band selection or dimensionality reduction and classification. Classification of a hyperspectral image sequences sometimes identifies which pixels contain various spectrally distinct materials. On the other hand, band selection algorithms reduce the data volume (dimensionality), without loss of critical information, so that it can be processed efficiently. Most applications of hyperspectral imagery require processing techniques that achieve one fundamental goal: Detect and classify the constituent materials for each pixel in the scene. Different classification techniques have been proposed ranging from minimum distance (Euclidean, Fisher Linear Discriminant, Mahalanobis, etc.) and maximum likelihood [1] to correlation matched filter-based approaches such as spectral signature matching [2]. Once all pixels are classified into

one of several classes or themes, the data may be used to produce thematic maps. Depending on the nature of the application, the thematic maps may be used to produce summary statistics regarding the objects in a scene or for object or target recognition purposes.

There are two major techniques to image classification: Supervised and unsupervised. In supervised classification techniques, an analyst develops quantitative descriptions of the spectral characteristics of the various classes of interest for a particular scene. These descriptions are then used as reference spectral signatures against which every pixel in an image is compared. The pixels are classified according to the spectral signature they most closely resemble. In unsupervised classification, the algorithms do not use training data as the basis for classification. Instead, the algorithms used examine the unknown pixels in the image and aggregate them into various classes according to the clusters found in the spectral space that contains the image.

With the advances in spectrometers we could differentiate theoretically among any materials. However, the analysis of these data requires a lot of computation, complexity and significant problems to the end user [3]. Also, on supervised classification the number of independent training samples should be greater than the number of bands in order that the class matrices may be inverted [4]. These problems are resolved by the removal of redundant information and trying to keep only the information relevant to the application. These could be done by taking advantage of the high correlation between the spectral bands. Different approaches are available, among which the most popular is the principal component. The principal component reorganizes the data in a way that the principal axis is one, and the data has the maximum variance. The problem with the principal component approach is that there is no physical relation between the original spectral bands and the transformation produced by the algorithm. Other methodologies could obtain similar results as the principal component but preserve the physical spectral bands. Among these techniques are the Single Value Decomposition (SVD) [5], Residual in Percent Error [6], Canonical Analysis [7], Projection Pursuit [8], Orthogonal Subspace Projection [9] and

Branch and Bounds [10].

1.2 Problem Statement

Hyperspectral algorithms require very high levels of computational throughput. These types of algorithms running on common workstations or on multiprocessors server environments could take days to finish. State of the art research is focused on finding better results by combining different algorithms and approaches but the complexity and computational workloads are increasing.

1.3 Solution Approach

A sequential optimization, parallelization and distribution of one dimensionality reduction algorithm, two unsupervised classification and one feedback classifier algorithm is performed. The development is done using the C programming language and Message Passing Interface (MPI) technology on the parallel programs. The work is developed on a HP IA64 computational cluster with 16 nodes running Linux operating system and a HP IA32 computational cluster with 16 nodes running also Linux. For the sequential code, optimizations libraries are used which contains advanced linear algebra functions as BLAS and LAPACK. Also the Parallelized Linear Algebra PACKage (PLAPACK) [11] based on MPI is used to leverage complex matrix manipulations operations in a parallel fashion. All algorithms are first developed in a sequential model on a single processor machine to test them and to gather benchmark data. Afterward the parallelization benchmarks are compared to the sequential results to provide check for improvements, benefits or penalties.

1.4 Objectives of this Thesis

The main objectives of this thesis are the following:

- Port and optimized two hyperspectral imaging unsupervised classifiers for IA32 and Itanium 2 architecture (maximum likelihood and euclidean distance)

- Port and optimized one dimensionality reduction algorithm for IA32 and Itanium two architecture (principal component)
- Port and optimized the feedback classification algorithm, which is a combination of a dimensionality reduction and a classifier in the same algorithm.
- Parallelize the unsupervised classifiers optimizations
- Parallelize the feature reduction optimization
- Parallelize the Feedback Classification Algorithm
- Gather all benchmarks and provide a performance analysis

1.5 Contributions

Most hyperspectral imaging classifiers exhibit the characteristic that each pixel classification is independent of each other. This is a characteristic that can be exploited for parallelization. But what happens when we need to calculate means and covariances for each class that are dependent of all pixels members of the class? These pixels could be anywhere on the distributed image or at least should be replicated on all nodes. How the communication of these large data sets impacts the explicit parallelization of the classification?

Our main contribution could be summarized as follows:

- Demonstrate that exhaustive search across subsets of features combinations is possible with reasonable execution times.
- Completed an analysis of image classification on a parallel environment and provide suggestions that should be taking into account depending on the image spatial and spectral resolution and the number of classes.
- Demonstrated how hyperspectral imaging algorithm characteristics can be exploited to develop new parallel approaches.

- Derived from all our literature review is the first time that classification and dimensionality reduction algorithms based on hyperspectral imaging have been implemented in a parallel environment. Also is the first time that implementation issues are documented and analyzed.

1.6 Thesis Structure

The rest of this thesis is organized as follows: In chapter 2 related work previously done in the area is discussed. Chapter 3 describes in detail the algorithms and its basic implementation issues. In chapter 4, a detailed discussion of the implementation of the applications for each algorithm is presented. It covers the sequential phase and the parallel phase on both architectures IA32 and IA64. Chapter 5 presents the performance evaluation of the sequential and parallel algorithms. Finally we conclude with a summary of results and future work in chapter 6.

CHAPTER 2

Related Work

We present here relevant work upon which this thesis is based. The areas of relevance are : a) Hyperspectral Feature Reduction Algorithms, b) Hyperspectral Classifiers c) Parallel Feature Reduction algorithms d) Parallel Classifiers and e) Computational Clusters

2.1 Hyperspectral Feature Reduction Algorithms

Hyperspectral images provides abundant information about the target area. With digital images in many contiguous and very narrow (about 0.010μm wide) spectral bands the computational burden increases with the dimensionality. Since the adjacent spectral slices are very high correlated in most cases we can reduce the dimensionality with minimum impact in the data representation. Feature reduction algorithms try to reduce the amount of the dimensions of each data set. Basically two approaches are used, band selection and feature extraction. On the band selection approach the "best" bands are selected based on different criteria. On the feature extraction mathematical and statistical models are applied to the data to obtain the best representation of the original data without the correlation between bands and thus reducing the dimensionality. On the later approach the concept on bands is lost and the data representation can not be correlated back to the spectral bands. The following algorithms are representative of the feature reduction approaches commonly used in the remote sensing community. A lot of other methods have been published with very efficient results but these are the most populars [12-13].

2.1.1 Principal Component Analysis

Principal Component Analysis (PCA) is a common technique for multivariate data. PCA is a procedure for transforming a set of correlated variables into a new set of uncorrelated variables. This transformation is a rotation of the original axes to new orientations that are orthogonal to each other and therefore there is no correlation between variables. In this new rotation, the first variable or axis contains the maximum amount of variation, or accounts for the maximum amount of variation. The second axis contains the maximum amount of variation orthogonal to the first. The third axis contains the maximum amount of variation orthogonal to the first and second axis and so on until one has the last new axis that is the last amount of variation left. To calculate this rotation we need to obtain the covariance matrix for the data. The data or hyperspectral image is represented as a matrix where each row represents a pixel or observation and each column represents a wavelength or spectral band. If we have an nxn image with $\boldsymbol{N}$ bands the data matrix has n^2 rows and $\boldsymbol{N}$ columns. Using this data matrix we could calculate the covariance matrix for the whole data. The resulting matrix is of dimensions of NxN and provides the relation between each spectral band. From a symmetric matrix such as the covariance matrix the orthogonal basis can be obtained by finding its eigenvalues and eigenvectors. A total of $\boldsymbol{N}$ eigenvalues $(\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_{n-1}, \lambda_n)$ is obtained from the covariance matrix. Each of these eigenvalues has a correspondent eigenvector that is orthogonal between the other vectors. If the eigenvalues are ordered from maximum to minimum the vector related to the biggest value contains the direction of the largest variance of the data. Since the first $\boldsymbol{K}$ direction vectors represents a significant amount of variance of the whole set, we can reduce the dimensionality of the original observations or pixels by projecting each one of them to the first $\boldsymbol{K}$ orthogonal vectors. This reduces the dimensionality from $\boldsymbol{N}$ to $\boldsymbol{K}$ and in turn the complexity and computational workload is reduced considerably [14].

2.1.2 Independent Component Analysis

The Independent Component Analysis (ICA) instead of transforming the original hyperspectral image like the PCA, tries to determine which bands are the best suitable for classification. It basically extracts independent source signals by searching for a linear or nonlinear transformation which minimizes the statistical dependence between components. The main work of the ICA is to determine a weight matrix $\boldsymbol{W}$. Assuming we have an observed signal $\boldsymbol{X}$ which contains the spectral profile of all pixels in the image, then the source signal $\boldsymbol{S}$ resides in a lower dimension space corresponding to the present materials in the hyperspectral image, and each dependent component $\boldsymbol{s_i}$ is different for each material. Since the number of materials contained on the image is unknown the number of materials m is assumed randomly and the weight matrix $\boldsymbol{W}$ is observed and the contribution of each original band to the ICA transformation is logged. At the end it can be estimated the importance of each spectral band for all materials [15].

2.1.3 Projection Pursuit

On supervised classification environments the number of training samples should be adequately large so the estimation of statistics at full dimensionality could be accurate enough. However, on most cases there are not enough training samples available to handle all the spectral bands and the estimated features may not be as effective as they could be. This suggests the need for dimensionality reduction via a processing mechanism that takes into account high-dimensional feature space properties [8]. Projection Pursuit (PP) is able to bypass many of the problems of the limitation of small number of training samples by making the computations in a lower dimensional space, and optimizing a function called the projection index, in which the Bhattacharyya distance (Equation 4.5) is commonly used. The PP approach is basically a linear low-dimensional projection of a high dimensional data set [14].

2.1.4 Genetic Algorithms

The basic idea of genetic algorithms (GA) can be described as follows: A population is created with a group of individuals created randomly. The individuals in the population are then evaluated. The evaluation function is provided by the programmer and gives the individuals a score based on how well they perform at the given task. Two individuals are then selected based on their fitness. The higher the fitness, the higher the chance of being selected. These individuals then "reproduce" to create one or more offspring, after which the offspring are mutated randomly. This process continues until a suitable solution has been found or a certain number of generations have passed, depending on the needs of the programmer [16-17]. .

When we use GA on hyperspectral imaging the spectral data is referred as population and each band is an individual. Thus the individuals "reproduce" itself and could create different mutations. Basically each band is represented in an encoded binary vector (also known as chromosome) where the combinations of different bands are represented. This chromosome is then evaluated via fitness criteria that could be a class discriminant distance. Different probabilities are assigned for reproduction, crossover and mutation depending on the algorithm. Finally the resultant chromosome which contains the best fitness is selected. The chromosome vector is then decoded back to the proper bands. Genetic Algorithms provides a fast and reliable way to get the best features of a data set without the need to compare overall search methods [18-19].

2.1.5 Feedback Classification Algorithm

The search for optimal bands by analyzing all possible combinations is be very computing exhaustive. The number of combinations of bands increases exponentially as the dimensionality increases and, as a result, an exhaustive search quickly becomes impractical or impossible, at least on sequential approaches. The goal of the feedback classification algorithm (FCA) is to select the subset of bands that best separates the centroids of a

given number of classes. This is done by creating the whole possible combinations of $\boldsymbol{m}$ bands from the total of $\boldsymbol{N}$, where m is the desire final number of bands and $\boldsymbol{N}$ is the total number of bands. The number of possible combinations is given by 2.1. Each combination is referred to as a set. The covariance matrix and the mean for each class are calculated for each set. These values can be obtained by using the pixel class membership of a previous classification. This classification can be an initial one when the algorithm is starting or a classifier output on a previous iteration.

$$\begin{pmatrix} N \\ m \end{pmatrix} = \frac{N!}{(N-m)!m!} \tag{2.1}$$

Among all the sets the one with the largest average distance between its class centroids is selected. The set selected is then the input to the classifier and when it finishes the computation the classified pixels is used again to select another possible sets. The algorithms stop when the same set is continuously selected or the algorithm reaches its maximum iteration number [20].

2.2 Hyperspectral Classifiers

2.2.1 Euclidean Distance

In the classification area we want to obtain a thematic map that classifies each pixel into one of $\boldsymbol{C}$ classes. The variable $\boldsymbol{C}$ is a parameter that establishes beforehand the number of classes in which each vector pixel could be classified. How this parameter is obtained depends totally on the region of interest and on the prior knowledge of the area. On most classification algorithms each pixel on the final thematic map should be classified as a member of one of the $\boldsymbol{C}$ classes.

The Euclidean Distance classifier takes $\boldsymbol{C}$ initial points in the image. In this case points are the signatures of the materials we want to detect. The distance between each vector pixel on the image and the $\boldsymbol{C}$ vectors is calculated. The Euclidean equation is used

to calculate this distance (2.2). Where $\boldsymbol{X}$ is the vector pixel, $\boldsymbol{M_i}$ is the mean vector for the class $\boldsymbol{i}$ and $\boldsymbol{N}$ is the number of bands.

$$g_i(x) = (X - M_i)^T (X - M_i) \tag{2.2}$$

Each vector pixel is then assigned as a member of the closest $\boldsymbol{C}$ point. Once all the pixels in the image are assigned as members of one of the $\boldsymbol{C}$ points, each $\boldsymbol{C}$ group calculates its own new point or centroid. With these new points or means we repeat the process an each vector pixel is classified again as member of one of the $\boldsymbol{C}$ groups. The whole process continues until none of the pixels change from one group or class to another or it could be based on iterations. On most analysis good results are obtained using five or more iterations. Another approach is using the sum of square distances (SSD) to monitor the decreasing distances between the class centroids and its members, since SSD is a non increasing sequence and has been demonstrated that it converges.

Euclidean distance is widely used in the community but it has one setback. It assumes that all points are at the same distance from its mean, so the area of classification will always be circular. It basically assumes the covariance of the data as an identity matrix. If the data distribution on each class is not based on equidistant points, the classification may fail [14].

2.2.2 Fisher's Linear Discriminant

To avoid the Euclidean problem of equal distances between all points of a class and its mean, Fisher's Linear Discriminant (FLD) incorporates the data covariance into the equation. The data covariance provides the equation with a relationship between spectral bands. With this extra data in the equation, better results are obtained compared to the Euclidean distance. However a problem remains when the class covariances start changing and the the data covariance is not representative anymore for each class [14].

$$g_i(x) = -(X - M_i)^T \hat{\Sigma}_c^{-1} (X - M_i) \tag{2.3}$$

2.2.3 Mahalanobis Distance Classifier

Mahalanobis distance (Equation 2.4) is used in analyzing cases in discriminant analysis. For instance, one might wish to analyze a new, unknown set of cases (pixels) in comparison to an existing set of known cases. Mahalanobis distance is the distance between a case and the centroid for each group (of the dependent) in attribute space (n-dimensional space defined by n variables). A case (pixel) has one Mahalanobis distance for each group (class), and it is classified as belonging to the group for which its Mahalanobis distance is smallest. Thus, the smaller the Mahalanobis distance, the closer the case is to the group centroid and the more likely it is to be classed as belonging to that group. The main difference between Mahalanobis distance and Fisher's Linear discriminant is that in the Mahalanobis each group of class has its own covariance. In FLD the same data covariance is used for all classes. Thus, Mahalanobis provides in the covariance of each class a method to classify groups with different distributions in its data, and thus better accuracy [14].

$$g_i(x) = -(X - M_i)^T \hat{\Sigma}_i^{-1} (X - M_i) \tag{2.4}$$

2.2.4 Maximum Likelihood

This classifier is based on statistical information. Assuming that the vector pixel $\boldsymbol{X}$ is normally distributed with mean M and variance $\hat{\Sigma}$ where both M and $\hat{\Sigma}$ are unknown, the likelihood function becomes:

$$g_i(x) = -\frac{1}{2} \ln \hat{\Sigma}_i - \frac{1}{2} (X - M_i)^T \hat{\Sigma}_i^{-1} (X - M_i) \tag{2.5}$$

The vector pixel $\boldsymbol{X}$ belongs to the class that has the functions with the largest $g_i(x)$. When the above equation is maximized and solved we have:

$$M_i = \frac{1}{n_i} \sum_{k=1}^{n_i} X_k \tag{2.6}$$

$$\hat{\Sigma}_i = \frac{1}{n_i - 1} \sum_{k=1}^{n_i} (X_k - M_i)(X_k - M_i)^T \tag{2.7}$$

The mean (2.6) and covariance (2.7) have to be recomputed for every class. The algorithm stops when there is not a significantly change between the M_i and $\hat{\Sigma}_i$ previously calculated. Also as in the Euclidean classifier the algorithm can be stopped based on iterations or by using SSD.

As we notice from equations 2.5 and 2.4, the main difference between Mahalanobis distance and maximum likelihood is the function threshold. The threshold on the maximum likelihood equation is in charge to move classifier boundary by taking into account the data covariance. Thus on must cases maximum likelihood exhibits more accurate results [14].

2.3 Parallel Feature Reduction Algorithms

Some implementations used Principal Components in a parallel way, but they simply used the components as inputs of artificial neural networks. Then different processors could process its own principal component concurrently. But the PCA calculation is done sequentially [21].

A parallel version of the Projection Pursuit for high dimensional feature reduction was presented on [22]. On this approach each group of adjacent bands is linearly projected to obtain one feature. The projections in every group are independent of each other. The advantage of this approach is that it is fast because every group of adjacent bands is projected in parallel and independently of one another.

In summary, on all of our literature review we found limited references regarding dimensionality reduction algorithms on a parallel environment. Since most of feature ex-

traction algorithms relies on mathematical transformation and thus heavy linear algebra operations, the difficulty to develop them in a parallel approach is amazing. On most attempts the parallel implementation have not better execution times than sequential versions. However on the subset selection algorithms where the each band is preserved is an area that could be exploited since on most cases a criteria is applied to all bands or combinations or bands independently.

2.4 Parallel Classifiers

Some work has been done on parallelization of classifiers on the deta mining area. The most used classifiers are the ones based on decision tree based classification. In this type of parallel classifiers the nodes of the decision based tree are dispersed into processors and each processor performs all the node computation. They provide load balancing and data distribution mechanisms and are very scalable [23-25].

A parallel Euclidean distance approximation had been developed for basic image analysis operations such as Distance Transformation (EDT). In this method the spatial image is divided into as many sub regions as processors available. These sub regions are then processed in a parallel fashion with some processors communication for global calculations [28].

On the computer vision area, parallel classifiers are used to classify images regions into objects models from a database which is commonly known as knowledge-based. In this approach a master or host processor interprets and analyze an image or frame into symbolic regions representations like lines, color, texture, size, shape, orientation, length, etc. Then this host processor distribute the regions between slaves processors and each processor classify its own regions. After all regions have been classified as described above, a global consistency check is then performed. If there are no conflicts the classification obtained holds [26].

Text classification is another are where parallelization of classifiers have been im-

plemented. The main problem is the amount of unclassified documents into predefined categories. For text classification a document is parsed an a collection of unique words is obtained. Each document then creates an histogram of words frequency and then converts it to a matrix of frequencies for all documents. The classification is the performed by using a priori training documents already classified. In the parallel approach a master processor builds global parameters based on the a priori labeled documents and broadcast them to all processors. Each processor then estimates a class of each of its documents by using the global parameters and estimates a new local parameters given the estimated class. These local parameters are then sum up to obtain new global parameters. The iterations continues until convergence [27].

From all the models of parallel classifiers studied the text classifiers have one of the characteristics of the hyperspectral classifiers, since local nodes needs global parameters and global parameters depends on local nodes results. It is based also on iterations and fits perfectly on hyperspectral parallel image processing.

2.5 Computational Clusters

Because of the relevance of computational clusters to this thesis, a brief description regarding high performance cluster systems is provided.

High Performance clusters started back in 1994 when Donald Becker and Thomas Sterling built a cluster for NASA. This cluster was made up of 16 DX4 processors connected by 10 Mbit Ethernet, and they named it Beowulf. Since then, the Beowulf Project has been joined by other software projects trying to provide useful solutions to turning Commercial Off the Shelf (COTS) hardware into clusters capable of supercomputer speed. These clusters have been used for everything from simple data mining, file serving, database serving, or web serving, to flight simulation, computer graphics rendering, weather modeling, or ripping CDs at truly outstanding speeds [29].

A Computational cluster is a multi computer architecture which can be used for

parallel computations. Clusters are built upon commodity hardware components, e.g. any PC capable of running Linux, standard Ethernet adapters, and switches. It does not contain any custom hardware components and is trivially reproducible. In addition, a cluster also uses commodity software like the Linux operating system, Parallel Virtual Machine (PVM) and Message Passing Interface (MPI). The server node controls the whole cluster and serves files to the client nodes. It is also the cluster's console and gateway to the outside world. Large Beowulf machines might have more than one server node, and possibly other nodes dedicated to particular tasks, for example consoles or monitoring stations. In most cases client nodes in a system are dumb, the dumber the better. Nodes are configured and controlled by the server node, and do only what they are told to do. In a disk-less client configuration, client nodes don't even know their IP address or name until the server tells them what it is. One of the main differences between computational cluster and a Cluster of Workstations (COW) is the fact that Beowulf behaves more like a single machine rather than many workstations. In most cases client nodes do not have keyboards or monitors, and are accessed only via remote login or possibly serial terminal. Computational nodes can be thought of as a CPU + memory package which can be plugged in to the cluster, just like a CPU or memory module can be plugged into a motherboard.

A computational cluster or Beowulf is not a special software package, new network topology or the latest kernel hack. Beowulf is a technology of clustering Linux computers to form a parallel, virtual supercomputer. Although there are many software packages such as kernel modifications, PVM and MPI libraries, and configuration tools which make the Beowulf architecture faster, easier to configure, and much more usable, one can build a Beowulf class machine using standard Linux distribution without any additional software. If you have two networked Linux computers which share at least the ***home*** file system via NFS, and trust each other to execute remote shells (rsh or ssh), then it could be argued that you have a simple, two node Beowulf machine.

2.6 Assessment

As demonstrated in above related topics there are a lot of different methods for performing hyperspectral imaging classification and dimensionality reduction. Since the main objective of this thesis is to explore the possibility to apply parallel computation to some representative algorithms, we decided to choose two classifiers, one dimensionality reduction algorithm, and one hybrid algorithm which included both. The classifiers selected are Maximum Likelihood and Euclidean distance. While the feature reduction algorithm is Principal Component and the hybrid algorithm is Feedback Classification (FCA). FCA is a combination of a band selection algorithm using previous classifier information. Most of feature reductions algorithms are designed to avoid the overall search of combinations. Since the Feedback Classification Algorithm do just that we will be attacking the worst case scenario. On our research we found some attempts previously to use parallelized computation on hyperspectral imaging, but they where based on special purpose architectures specifically design for image processing [30-31] Up today, we have not found a previous attempt to parallelize these algorithms from a hyperspectral imaging perspective and using common computational clusters.

CHAPTER 3

Parallel Hyperspectral Imaging

In this chapter we discuss in detail the development environment, the hardware used to perform the testing as well as the different parallelization approaches followed in the implementation of the parallel algorithms.

3.1 Development Environment

The development environment is summarized on Table 3.1. All the algorithms were developed on the Linux environment. On the Itanium side RedHat Advanced Server version 2.1 and the Intel C compiler version 8.1 were used. On IA32 all the development was carried out using RedHat 7.3 and the ***gcc*** [42] compiler version 2.96-20000731. Optimization libraries where used on both architectures. The Hewlett Packard Mathematical Library (HP MLIB) [32] version 1.21 was used on the Itanium architecture and the Intel Math Kernel Library (MKL) [33] was used on IA32, respectively. All algorithms were developed from scratch by the author by leveraging from Matlab [34] programs and built-in modules.

On the parallel implementation LAM/MPI [35-36] version 6.5.4-1 was used on IA64, while the version 6.5.6-4 was used on IA32. On the Principal Component Algorithm the Parallel Linear Algebra Package (PLAPACK) [11] was used on both architectures. Since PLAPACK requires a BLAS [37-39] base, on IA32 we used the MKL libraries and the MLIB libraries on IA64. All parallel programs were run on a 16 node cluster using a 1 Gigabit network connectivity. There was shared storage among all the nodes on the cluster and the

programs were compiled using shared libraries. The libraries were distributed on all the nodes in the cluster.

On both environments we used ***gdb*** [43] for algorithms debugging, ***gprof*** [44] for profiling and the Memcheck Deluxe [45] tool for memory usage.

	IA32 Architecture	Itanium Architecture
OS	Linux RH 7.3	Linux RHAS 2.1
Compiler	gcc 2.96-20000731	Intel C 8.1
BLAS Library	Intel MKL	HP MLIB
MPI	LAM 6.5.6-4	LAM 6.5.4-1

Table 3.1: Software Development Environment for both architectures.

3.2 Hardware

The specifications of the hardware used to test our implementations is as follows : 16 HP rx4640 machines with one Itanium 2 CPU at 1.5GHz and with 6MB of cache. On the IA32 side 16 HP BL10e G2 systems each with a Pentium M processor running at 1GHz and 512MB of cache. The network interconnection is done at 1Gbit on both clusters.

	IA32 Architecture	Itanium Architecture
Model	HP BL10e G2	HP rx4640
CPU	Pentium M 1GHz	Itanium 2 1.5GHz
Cache	1MB L-2	6MB L-3
RAM	512MB	2GB
Nodes	16	16
Interconnection	HP 5308 1Gbit switch	HP 5308 1Gbit switch

Table 3.2: Hardware Development Environment for both architectures.

3.3 Parallelization Analysis

The following section describes a parallelization design analysis for the four algorithms. The design analysis serves as a guide on the development phase of the research.

The designs were based on a multi node cluster using Message Passing Interface (MPI) as the nodes interface.

3.3.1 Feedback Classification Algorithm

In this algorithm the first classification is carried out only by the master node. Since this classification will be calculated using few bands, a single node can handle the task. After some efforts we were able to parallelize the combination generation among nodes. Each node receives a combination start and a combination end, with this range the node could start evaluating its own combinations totally independent of the other nodes. Once the node finishes its local calculations, it sends the local selected combination to the master node and receives another combination range to start again the process. If the remaining combinations are attended by other nodes then the node finishes its processing.

```
id = get_node_id
if(id == 0)
{
    first_classification()
    get_combination_range()
continue:
    foreach node
    {
        send_combination_range()
    }
    //there are no combinations left but there could be nodes without finish
    for(id = 1 to id = size)
    {
        criteria = recv() //receive combination to slaveN node
    }
    if(all_combinations_attended())
        get_lowest_critera();
    else
        goto continue:
}
else
{
    recv() //gets combination range
    calculate_average_distance_for_my_range();
    send() //send results to master node
    wait_for_more()
}
```

Figure 3.1: Pseudocode for the FIM parallel implementation

The master node gets all the local combinations selected by local nodes and the discrimination criteria. With this criteria the master can select the final combination. Figure 3.1 summarizes the pseudocode for the FIM parallel implementation.

3.3.2 Principal Component Analysis

For this algorithm there is very little opportunity to be parallelized. Since it is based on the covariance matrix of the image data all data elements should be on a same node. After this matrix is obtained its eigenvalues and eigenvectors also should be computed locally on a singular node. Since PCA calculation involves a lot of linear algebra calls and there is not obvious parallelization for the algorithm, we will be using PLAPACK to handle all linear algebra calls, and data distribution.

3.3.3 Classifiers

The Euclidean distance and the maximum likelihood classifiers are good algorithms where parallelism can be exploited, since each pixel independently calculates its membership to a class. The problem arrives at calculating the new means and the covariance for each class using the pixel membership. Since a class will have member pixels distributed across nodes there should be a way means and covariances could be calculated in an efficient parallel way. A master delegates nodes approach was developed. Since at some point in the iteration we need to gather and distribute data, we try to transfer the smallest amount of data possible. We decided to transfer the local classification vector to the master node at each iteration and then propagates the final vector back to the nodes. In this way the transfer is for an integer vector of the size of the pixels resolution. Once each node has its own copy of the global vector covariances and means are calculated locally. No additional transfers are done until the next iteration. With this approach each node has all the data necessary to calculate pixel memberships without the need to transfer huge amount of data for each node (Means an Covariance). However this approach could be problematic in some images. If we look at Table 3.3 we can observe that for the first image is best to use our approach of broadcasting the classification vector since is very low spatial resolution image (145x145) and high spectral resolutions. But for the second image it is not the case since it has high spatial resolution but low spectral resolution, in that case it is better to

Scenario	Indian Pine Image	UPRM Image
Means and Covariances distributed	Means = 8,800 bytes Cov = 1,936,000 bytes	Means = 1,584 bytes Cov = 52,272 bytes
Classification vector distributed	84,100	1,228,800

Table 3.3: Possible distribution scenarios for the classifiers. For the first image it makes sense to only distribute the classification vector, but for UPRM image it would make sense to distribute the means and covariance

broadcast the covariance and mean than the classification vector. Figure 3.2 summarizes the pseudocode for Euclidean distance and maximum likelihood parallel implementations.

```
If(id == 0)
{
    divide_image_into_subregions()
    send_subregions()
}
else
    recv() //nodes_receive_subregion
while(iter != 0)
{
    if(id == 0)
        goto wait;
    else
    {
            recv() //receives subregion
            calc_subregion_membership()
    }
wait:
    MPI_Gather(); //Gather all subregions memberships
                  //into a single vector
    MPI_Bcast(); //Distribute final classification vector to
                  //all nodes
    calc_new_means() //Every node calculates the means and
                  //covariances
}
```

Figure 3.2: Pseudocode for the classifiers parallel implementation

3.4 Image Information

The algorithms developed use two images for testing them. In this section an overview of these two images is presented.

3.4.1 Aviris Pine Site

The first test data image is an AVIRIS image provided by Laboratory of Applied Remote Sensing (LARS) at Purdue University. The image size is 145 by 145 pixels with 220 bands and 12 bits of pixel information on the sensor. After atmospheric correction the pixels are stored as 16-bit words. The storage size is about 9.3 MB. Figure 3.3 shows a perspective picture of the image (band 30). The image was taken in 1992, covering the NW Indiana's Indian Pine Site 3, an agriculture area. The ground truth is also available for image classification evaluation. There are 16 land cover classes, in which some classes may be grouped into single landuse types. For example, corn, corn-min, and corn-notill belong to the corn landuse type, but due to the differences of crop canopies they are categorized into three different land-cover classes. Classes of a same group tend to possess similar spectral properties, so that it is usually difficult to differentiate them in a multispectral image. Our analysis always assume 5 classes on this image.

Figure 3.3: Indian Pine Site Image, 220 bands 145x145

Figure 3.4: University of Puerto Rico at Mayagüez Image, 33 bands 480x640

3.4.2 UPRM Test Image

The second image is a laboratory image provided by the Laboratory for Applied Remote Sensing and Image Processing (LARSIP) University of Puerto Rico at Mayagüez Campus (UPRM). The image size is 480x640 and each pixel has a resolution of 8 bit on the sensor and since it was obtained using a spectral camera no atmospheric correction bits are added. Figure 3.4 shows the image visible spectrum of the image. By visual analysis we could see at least 6 different classes. Thus, all the analysis is done using 6 classes.

CHAPTER 4

The Developed Application

In this chapter we present detailed description of the developed application. and the implementation of each of the algorithms based on both, the sequential and the parallel approaches.

4.1 Sequential Approach

In this section we describe the sequential approach implementation for all the algorithms. These algorithms were developed first than the parallel versions. The main modules are discussed for each of them. Some modules such as image load, covariance, means calculation, are used on all the algorithms. Thus if an algorithm block have been already discussed in a previous section, then a link to that section will be provided.

4.1.1 Feedback Classification Algorithm

Figure 4.1 presents the main program blocks of the implementation. We describe each of these blocks and its proper implementation in the application as follows.

4.1.1.1 Image Load

A module was developed to load hyperspectral data into a type double matrix. Each row of the matrix represents an observation or pixel of the image. Each column represents a dimension or a spectral band. Thus, if we have an image with 200 bands and 256 by 256 resolution, we will generate a matrix of 65536 rows and 200 columns.

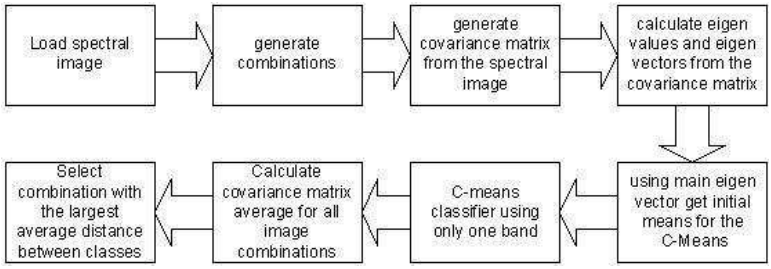

Figure 4.1: Feedback Classification block diagram for the sequential implementation.

The functions in Table 4.1 are in charge of loading the hyperspectral data from the binary image file and store it on a double address memory pointer passed as input to the function. The data is read only one time and all the subsequent modules use the same pointer. Since each hyperspectral image has its own sets of attributes it was decided that each spectral image would have its own loading function. In this way we avoid the development of a very complex loading function that would have to auto sense image attributes and specific characteristics like spectral bands, resolution, pixel depth, etc.

Module API :	int load_imagename_image(double **ObservationMatrix);
Parameters :	double **ObservationMatrix //Double pointer were the image would be loaded as a 2D matrix.
File Location :	ImageName/imageload.c

Table 4.1: Image Load module API.

4.1.1.2 Combinations Generation

For the combinations generation we use an algorithm developed by Donald Knuth and coded by Glenn C. Rhoads at the Rutgers State University of New Jersey. This algorithm basically prints the combination sets generated but do not store them. This was a major challenge since the sets obtained are very memory intensive. The first approach for the implementation was to store the combinations obtained on an integer two-dimensional array where each row represents a combination generated. However, the amount of memory needed was impractical. Table 4.2 presents the calculations made of the required memory

depending on the set size or final bands $\boldsymbol{m}$. Since the memory needed could be in the order of 10^{27} storing the combinations on local disks was also out of the question.

m (# of desire bands)	∗Number of Combinations	⋆Amount of memory needed (bytes)
3	1.75×10^6	21×10^6
5	4.10×10^9	82.05×10^9
7	4.49×10^{12}	125.83×10^{12}
∗Combinations are calculating using Indiana Spectral Image and equation 2.1 ⋆Each integer use 4 bytes		

Table 4.2: Combinations computational requirements assuming the Indian Pine Site pine test site image. As could be seen storing these combinations will be impractical.

The final approach was to generate combinations as needed. The code implementation was modified so each time the function is called it returns the next combination. In this way the combinations are not stored and the memory resources are better used. Since each combination will be sent as indexes of the spectral image to be analyzed, it is expected that the algorithm execution times would be immerse on larger execution times and thus there is no bottleneck. However, if we assume that the results will be received immediately from the nodes $\boldsymbol{t \approx 0}$ then the whole method could not be executed in less than the combination generation execution times. The problem is not the combination generation algorithm performance. Indeed the problem is that the quantity of combinations generated tends toward ∞. when $\boldsymbol{m}$ increases. This behavior continues until $\boldsymbol{m}$ is greater than $\frac{N}{2}$. At this point the combinations decrease again. The module API as shown in Table 4.3 consists of two functions: First set_combinations() sets the module with the proper parameters. Then new_combination() is in charge of get the new combination and stored in the memory address set previously. When there are no more combinations this function returns -1.

4.1.1.3 Covariance Matrix Generation

The covariance matrix is a symmetric matrix, which represents the correlation between a single spectral band and all the others bands. For example, element $\boldsymbol{(m, n)}$

Module API :	int set_combination(int N, int r, int *comb);

Parameters :	int N
	//Integer which represent the spectral channels.
	int r
	//Integer which represent the set size for each combination.
	int *comb
	//Integer pointer where each combination will be stored.

Module API :	int get_combination();
Parameters :	NONE
File Location :	Combinations/combination.c

Table 4.3: Combinations module API.

of the covariance matrix represents the correlation between spectral band $\boldsymbol{m}$, and spectral band $\boldsymbol{n}$. By definition a correlation between $\boldsymbol{A}$ and $\boldsymbol{B}$ is the same as the correlation between $\boldsymbol{B}$ and $\boldsymbol{A}$, which is why the covariance matrix is a symmetric matrix. The symmetric matrix for a set of observations is calculated using equations 2.6 and 2.7. $\boldsymbol{X_k}$ is the $\boldsymbol{k^{th}}$ observation obtained from the spectral image.

The module API, as shown in Table 4.4, consists of the covariance() function. This function basically calculates the covariance matrix from the spectral image and stored it on a memory address.

Module API :	void covariance(double **Data, int m, int n, double **Cov);

Parameters :	double **Data
	//Double pointer were the spectral image resides.
	int m
	//Integer Number of rows or pixels on the image.
	int n
	//Integer number of the image spectral channels.
	double ** Cov
	//Double pointer where the Covariance Data will be stored.

File Location :	Covariance/covariance.c

Table 4.4: Covariance module API.

4.1.1.4 Calculate eigenvalues and eigenvectors

By calculating the eigenvalues ($\boldsymbol{\lambda_i}$) and eigenvectors ($\boldsymbol{V_i}$) (Equation 4.1) of the covariance matrix $\hat{\boldsymbol{\Sigma}}_{\boldsymbol{i}}$ we could obtain the direction of the variances in the spectral data (Equation 4.1). Moreover if we order the eigenvectors in the order of the descending eigenvalues (largest first), we can create and ordered orthogonal basis with the first eigenvector having the direction of largest variance. In this way, we can find directions in which the data set has most significant amounts of energy. Instead of developin our own Eigen vector calculation functions we used the Basic Linear Algebra Subprogram (BLAS) Level 3 function ***dgeev()***. The ***dgeev()*** computes for an $\boldsymbol{N * N}$ real non symmetric matrix A, the eigenvalues and, optionally, the left and/or right eigenvectors. This function is contained on the HP MLIB and Intel MKL optimizations libraries. More info could be obtained at [32-33][37-39].

$$[\Sigma_i - \lambda_i I] * V_i = 0 \tag{4.1}$$

Because of the ***dgeev()*** needs some extra parameters that are not important to the rest of the program, this function was wrapped under the eigen() function. Table 4.5 shows and explains the parameters for this function.

Module API :	int eigen(double **Data, int m, int n, double *Eval, double **EVec);

Parameters :	double **Data
	//Double pointer were the spectral image resides.
	int m
	//Integer Number of rows or pixels on the image.
	int n
	//Integer number of the image spectral channels.
	double *Eval
	Double pointer array where Eigen Values will be stored
	double **Evec
	//Double pointer where the Eigen Vector matrix will be stored.

File Location :	NLIB/eigen.c

Table 4.5: Eigen Vectors and Values module API.

4.1.1.5 Initial Means generation

As explained in the previous section, using the eigenvalues and eigenvectors we have the directions of variance of the data. For unsupervised classifiers is a common procedure to calculate the initial means by using the largest EigenVector ($\boldsymbol{V_{Max}}$). First we need to calculate the spectral data mean $\boldsymbol{M}$, and then using equations 4.2 and 4.3 the initial means are calculated. The β value on equation 4.4 used for this implementation is 1.5.

$$M_1 = M - i * k * \sqrt{\lambda_{Max}} * V_{Max} \tag{4.2}$$

$$M_2 = M + i * k * \sqrt{\lambda_{Max}} * V_{Max} \tag{4.3}$$

$$k = \begin{cases} \frac{2\beta}{N-1} & \text{N is odd} \\ \frac{2\beta}{N} & \text{N is even} \end{cases} \tag{4.4}$$

The means module API, as shown in table 4.6, consists of a two functions that returns the initial means for the spectral image data. On the Classifier step the classification is not necessarily done using all spectral bands on the image. We could use a selected number of bands to perform the classification. Due this requirement two functions to get the initial means are needed. The first one getmeans_from_all_data() is used when calculating the initial means from the whole spectral image. On the other hand getmeans_from_subset_data() is used when only a subset of spectral image available is used. The means on a subset data are calculated using only the spectral bands that where selected. This module internally calls other modules API as covariance() and eigen().

4.1.1.6 Classifier

Since the classifiers used for the FIM algorithm are the same used on the classifier, they will be discussed in detailed in section 4.1.3.2.

Module API :	void getmeans_from_all_data(double **Data, int obs, int channels, int no_classes, double **means);

Parameters :	double **Data
	//Double pointer were the spectral image resides.
	int obs
	//Integer Number of rows or pixels on the image.
	int channels
	//Integer number of the image spectral channels.
	int no_classes
	//Integer Number of classes for the classes.
	double ** Means
	//Double pointer were the Means will be stored.

Module API :	void getmeans_from_subset_data(double **Data, int obs, int channels, int no_classes, int no_subset_bands, int *bands, double **means);

Parameters :	double **Data
	//Double pointer were the spectral image resides.
	int obs
	//Integer Number of rows or pixels on the image.
	int channels
	//Integer number of the image spectral channels.
	int no_classes
	//Integer Number of classes for the classes.
	int no_subset_bands
	//Integer number of single bands to be used
	int *bands
	//Integer array of size no_subset_of_bands that contains the bands
	double ** Means
	//Double pointer were the Means will be stored.

File Location :	Means/means.c

Table 4.6: Means module API.

4.1.1.7 Discrimination by largest mean average distance

Once we have the classification vector we could start the discrimination between combinations. Each spectral combination is obtained and with the classification vector we could calculate the means ($\boldsymbol{M_i}$) and covariances ($\hat{\boldsymbol{\Sigma}}_{\boldsymbol{i}}$) for each class. These parameters will be different between spectral combinations since each combination will represent a different set of spectral bands. With the means and covariances we then calculate the distances between classes. Using equation 2.1 we could obtain the classes combinations of distances for each spectral combination. For example, if the number of classes is 5 and the distances needs to be calculated in pairs (2) we have 10 possible means combinations. All these distances are calculated and then an average is obtained. This average will be the average for the spectral set. The spectral set with the largest average will be selected. The distance between classes is then calculated using the Battacharyya equation 4.5. For simplicity purposes our current implementation uses Euclidean distance.

$$B = \frac{1}{8}[M_1 - M_2]^T \left(\frac{\hat{\Sigma}_1 + \hat{\Sigma}_2}{2} \right)^{-1} [M_1 - M_2] + \frac{1}{2} \ln \left(\frac{\frac{|\hat{\Sigma}_1 + \hat{\Sigma}_1}{2}}{\sqrt{|\hat{\Sigma}_1||\hat{\Sigma}_2|}} \right) \tag{4.5}$$

4.1.2 Principal Component

Figure 4.2 shows the block diagram for the sequential principal component analysis (PCA). In these block diagrams all major PCA modules are presented. As could be seen some modules are the same ones used on the FIM algorithm. In our implementation we try to reuse code as much as possible, so the same modules are used. Here are a brief description of the PCA modules.

4.1.2.1 Covariance Calculation

The image data covariance calculations is on of the few modules common between the four algorithms. It was explained in detailed in section 4.1.1.3.

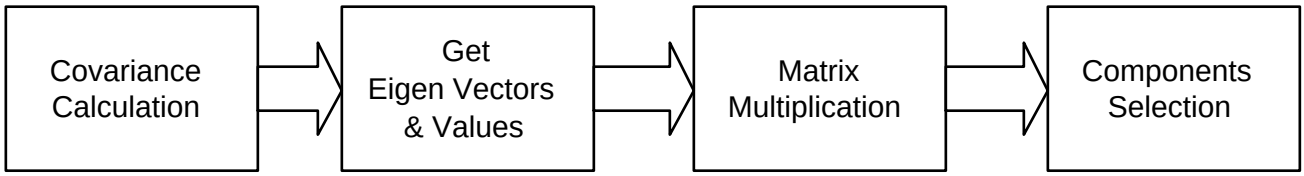

Figure 4.2: Principal Component block diagram.

4.1.2.2 Eigen Values and Vectors Calculation

In these steps we get the orthogonal vectors matrix. The matrix size will be of size $\boldsymbol{N * N}$ where $\boldsymbol{N}$ is the number of spectral bands. This module was explained on section 4.1.1.4

4.1.2.3 Matrix Multiplication

The matrix multiplication on the PCA algorithms accounts for more than 80% of its execution if we used a common $\boldsymbol{O(n^3)}$ matrix multiplication implementation. Thus we decided to use the optimizations libraries. The level 3 of the Basic Linear Algebra Subprograms (BLAS) provides an optimized function for matrix multiplication. The function $\boldsymbol{dgemm()}$ contains on HP MLIB and Intel MKL libraries provide an efficient method for matrix multiplication. When we replace our original method with this procedure the matrix multiplication accounts for only 10% of the algorithm execution. More information of this and other BLAS functions could be obtained at [32-33][37-39]. [37]

4.1.3 Classifiers

The Figure 4.3 presents the block diagram for the Classifiers. Each of block of the classifier is defined as a module on the implementation. Some modules are the same for the above sections so the a link will be provided in such cases to the corresponding sections. The modules described on this section are valid for both classifiers, Maximum Likelihood and Euclidean Distance.

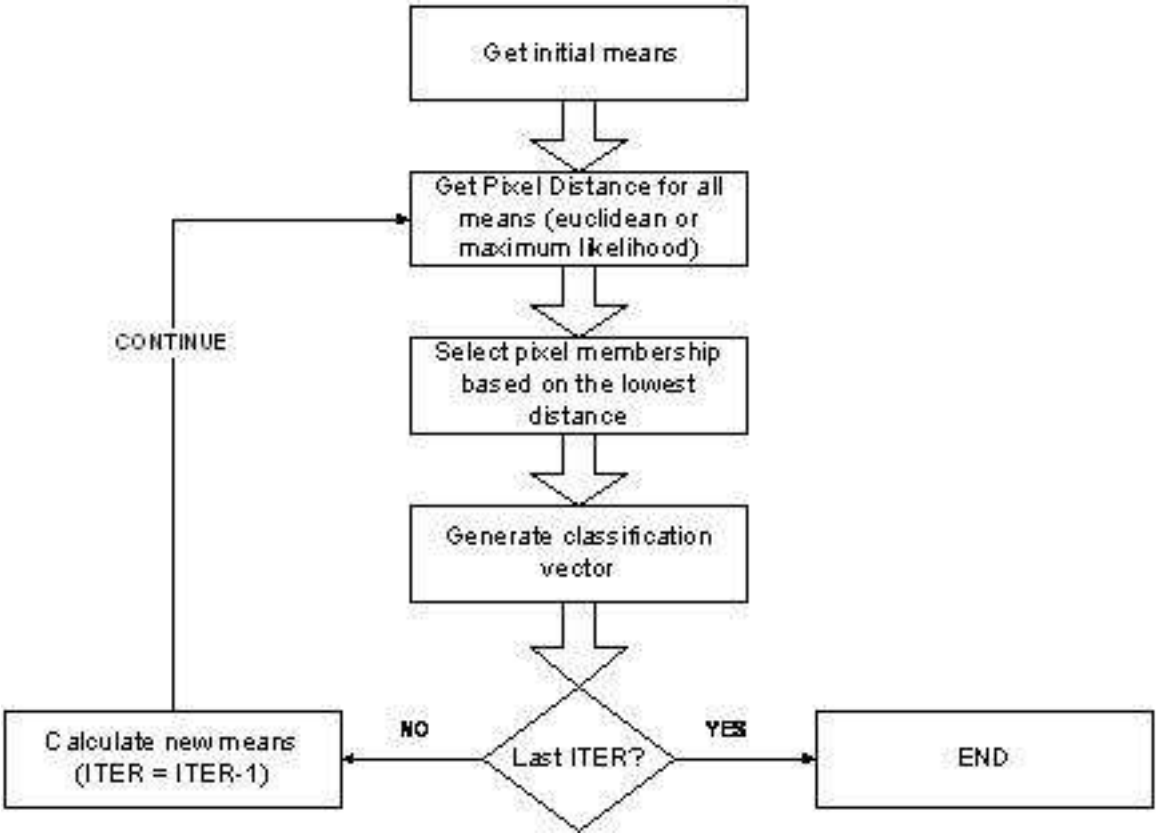

Figure 4.3: Classifier block diagram for the sequential implementation.

4.1.3.1 Get Initial Means

The first step on the classification algorithms is to get the initial means from the whole data. Since we are working with unsupervised parametric classifiers, the initial means are obtained from the covariance matrix. In a supervised classifier the initial means were obtaining by processing the training samples and generating a covariance with such samples. The initial means for unsupervised classifier is the same described in sections 4.1.1.3, 4.1.1.4 and 4.1.1.5.

4.1.3.2 Discriminant Classifier

The idea when we start developing this hyperspectral imaging suite was to begin with a few classifiers and then expand it to more. So the classifiers API are the same for both discriminants. Basically the functions have the same input parameters and are totally independent from each other. Also they provide the same output, an integer array of size $\boldsymbol{n} * \boldsymbol{m}$, where $\boldsymbol{n}$ is the number of rows (observations) in the image and $\boldsymbol{m}$ is the number of columns (bands). Each $\boldsymbol{i_{th}}$ element of the array is the classification result for the $\boldsymbol{i_{th}}$ pixel on the image. Table 4.7 presents the euclidean and maximum likelihood modules API.

4.2 Parallel Approaches

In this section we describe the parallel implementation for the algorithms. Most of the parallel work was based on the sequential algorithms. With small modifications to the sequential implementations and exploiting the algorithm's parallel characteristics we were able to implement parallel versions of them.

4.2.1 Feedback Classification Algorithm

The parallelization of the Feedback Classification Algorithm was basically the same as proposed on Figure 3.1. We spent a lot of time trying to parallelized the combinations generation. Using the sequential code explained on section 4.1.1.2 we were able to send each node a combination range. At the first of the algorithm after the FIM classification, a

Module API :	void euclidean(double **Data, in no_bands, int no_obs, int *bands, int no_subset_bands, double **Means int no_classes, int *class_vector, int iter);
Module API :	void ml(double **Data, in no_bands, int no_obs, int *bands, int no_subset_bands, double **Means int no_classes, int *class_vector, int iter);

Parameters :	double **Data
	//Double pointer were the spectral image resides.
	int no_bands
	//Integer Number of cols or bands on the image.
	int no_obs
	//Integer Number of rows or pixels on the image.
	int *bands
	//Integer Array which contains the single bands (optional).
	int no_subset_bands
	//Integer Number of single bands to be used (optional).
	double **Means
	//Double pointer were the initial class means resides.
	int no_classes
	//Integer Number containing the number of classes
	int *class_vector
	//Integer Array where the classification results will be stored
	int iter
	//Integer Number of the number of iterations for the classifier

File Location :	Classifier/euclidean.c
File Location :	Classifier/ml.c

Table 4.7: Euclidean and Maximum Likelihood modules API.

master node calculates sends $\boldsymbol{m}$ to each node until all nodes are working in a combination range. Each node then starts calculating the average distance for its range of combinations. After a node finishes, it send the selected combination with the average distance to the master node. The master node then receives the data and evaluate it from a previous largest average obtained from a previous node. If the new average is greater then the combination is selected as the new largest average and all subsequent nodes results are going to be evaluated based on the new average distance. If the distance is lower than the previously obtained, then the combination is ignored. Once a node finishes it will wait until the master node send a work flag or a finish flag. If a node receives a work flag a new combination range is provided and the node will start again the evaluations. After all nodes have received a finish flag the algorithms stop and the master nodes has the largest average distance and its respective combination.

Originally we divided the number of spectral bands between the number of nodes in the cluster and assigned this range of combinations to the node. This was an error since the calculation of the combinations from 110 to 100 is exponentially more computing intensive than the calculation from 10-0. The result was that the first nodes finished very rapidly while the latest nodes will remain working for hours. We then start testing the algorithm to provide a way where all the nodes could remain busy most of the time. We get to the conclusion that the shortest the range the better. In this way nodes may finish its calculations very rapidly and then get another results. The only limitation was that the range can not be lower than the number of expected bands $\boldsymbol{m}$ because anything lower than that will result in a program error. The problem is that we wanted to calculated the best three bands but the node is evaluating the best two for a range. After all the work we concluded that the combination range it will be always set to the same value as the desire subset images, $\boldsymbol{m}$.

4.2.2 Principal Component

The main problem with the principal component as established in section 4.1.2 is that the algorithm does not provide and explicit parallel behavior. Since all of the algorithm modules are basically linear algebra manipulation we decided to use current Parallel Linear Algebra libraries. Among the evaluated ones were PBLAS, PSBLAS[40], PLAPACK[11] and ScaLAPACK[41]. We decided to used PLAPACK since it provides the simplify library implementation. Basically it provides matrix distribution across the cluster mesh and most of the BLAS functions are ported to PLAPACK. It is an upper layer framework that uses optimized BLAS libraries and MPI to archive the results. To be able to distribute an object it has to be defined as a specific type (PLA_Object). For this object to work it needs a distribution template. A Template is basically a way to attach a PLA_Object to specific nodes on the mesh. Each object may have different templates and each template requires a ***nb*** length. The ***nb*** basically contains the number of elements each node on the mesh will contain. If a 2x2 mesh is created (4 nodes) and we have a distributed matrix of size 10x10 we could select a ***nb*** of 5 and each node on the mesh contains a subregion of the matrix. If we choose a ***nb*** of 10 or greater, only the first node will contain the distributed matrix. Calculating the optimum ***nb*** is a very difficult task and its depend of the distributed matrix size, the size of the mesh and the number of nodes in the cluster.

After having the distributed matrix with the image we can them simply start calling PLAPACK built-in functions to perform the PCA algorithm. We created a new PLAPACK Covariance function that basically compute the covariance for a distributed matrix. The function PLA_Spectral_decomp() is used to obtain the Eigen vectors and values (that are also distributed across the mesh). Then the PLA_Gemm() was used to perform the matrix multiplication.

4.2.3 Classifiers

The Maximum Likelihood and Euclidean Distance parallelization are very similar. The approach is to exploit the independent pixel classification characteristics. Basically on both algorithms the image is divided into subregions and each subregion is computed in a parallel fashion between the nodes. The number of subregions is dependent on the size of the cluster. Since pixel classification are independent on each other the membership calculation can be done in a totally parallel way. The problem is that when going to another iteration the means for each class are needed on both classifiers and the covariance is needed on the maximum likelihood. To calculate the means and covariances we need to access all the pixels members of each class. Since this information is needed to the next iteration, we have two approaches. Either the means and covariances are calculated on a single node locally and then distributed by broadcasting to all the participating nodes on the cluster or each node can send to a master the subregion classified. With the subregion classification the master node could recreate the whole image classification and send them back to the nodes. As Table 3.3 establishes the optimum approach depends on the image spatial resolutions and spectral resolution. Since each classification vector contains all pixels on the image in the UPRM image, it will be better to propagate the Means and covariances than the classification vector. However since the main hyperspectral problem is more the amount of spectral bands than image resolution we decided to propagate the classification vector instead of the means and covariances.

4.3 Directory Structure

The structure of the directory is described as follows:

```
SSI/
        bin/              => Place where the binaries are created
        Classifiers/
          euclidean.c     => Euclidean Distance Classifier
```

```
  ml.c            => Maximum Likelihood classifier
  tools.c         => Contains some important functions
                     used on the classifiers
CMeans/
  main.c          => Main function for the classifiers algorithm
Combinations
  combinations.c  => Combinations generation functions
Covariance/
  covariance.c    => Covariance function
FIM/
  main.c          => Main function for the FIM algorithm
Data/
  IA32/           => Directory to store data for IA32
                     algorithms (debugging only)
  IA64/           => Directory to store data for IA64
                     algorithms (debugging only)
include/
  global.h        => Contains classes structure and the includes for the
                     different optimizations libraries
Images/
  Indiana         => Indian Pine Site Image
  Test.bip        => UPRM Test image
IndianaImage/
  imageload.c     => Contains the code to load the Indian AVIRIS Image
Means/
  means.c         => Contains all the functions that are
                     used for means calculation
```

```
NLIB/
  nlib.h          => Contains all the linear algebra functions
PCA/
  main.c          => Main function for the PCA algorithm
TestImage/
  imageload.c     => Contains the code to load the UPRM testing image
Makefile.ia32     => IA32 Makefile
Makefile.ia64     => IA64 Makefile
```

CHAPTER 5

Experimental Results

In this chapter the results for parallelized versions of the hyperspectral imaging algorithms are provided. First the methodology for gathering benchmarks and images is described. Second the algorithms are validated and finally the results gathered for all the algorithms are presented and discussed.

5.1 Methodology

All the algorithms were tested on the specific architectures. For the sequential algorithms the clock() function was used. Clock calculates the best available approximation of the cumulative amount of time used by your program since it started. To convert the result into seconds, divide by the macro 'CLOCKS_PER_SEC'. In this way the execution times has a better accuracy since it is not dependent on the machine workload. Each of the sequential algorithms was run once to obtain the execution time.

On the parallel algorithms the MPI function called MPI_Wtime(). This is intended to be a high-resolution, elapsed (or wall) clock. It returns the current time in seconds. The time resolution is dependent of the MPI_Wtick() function and is dependent on the architecture. The problem with getting benchmarks using this function is that it does not takes into account the machine load at the moment of the execution. To try to obtain more accurate results each parallel algorithm was executed two times. If the results obtained were similar then no more executions were done. If the results have discrepancies (more

than 5 seconds) a third run was performed and the execution time was the average between the lowest two executions.

5.2 Validating Algorithms Accuracy

In this section we compare our application results with the Matlab results. In this way we assure that the algorithms implemented are accurate.

5.2.1 Euclidean Accuracy

Figure 5.1 provides the thematic classification of the Indiana Pine site image. Every single pixel on both implementations (Matlab and C) were classified equally. For this result 5 classes and 50 iterations were used. Also on Figure 5.2 we present the euclidean result of our application for the UPRM test image. We also obtained a 0 pixel difference between the Matlab implementation.

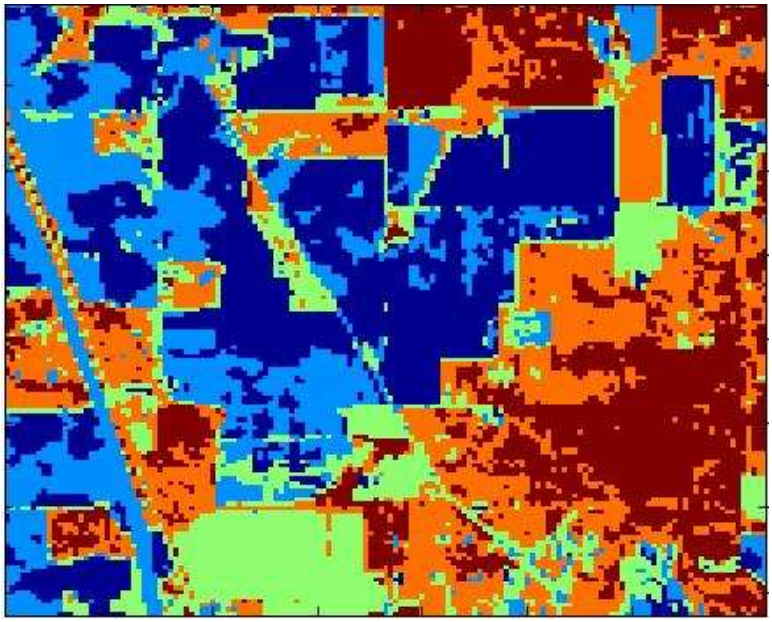

Figure 5.1: Euclidean Indiana image result. Pixels Discrepancies with Matlab: 0

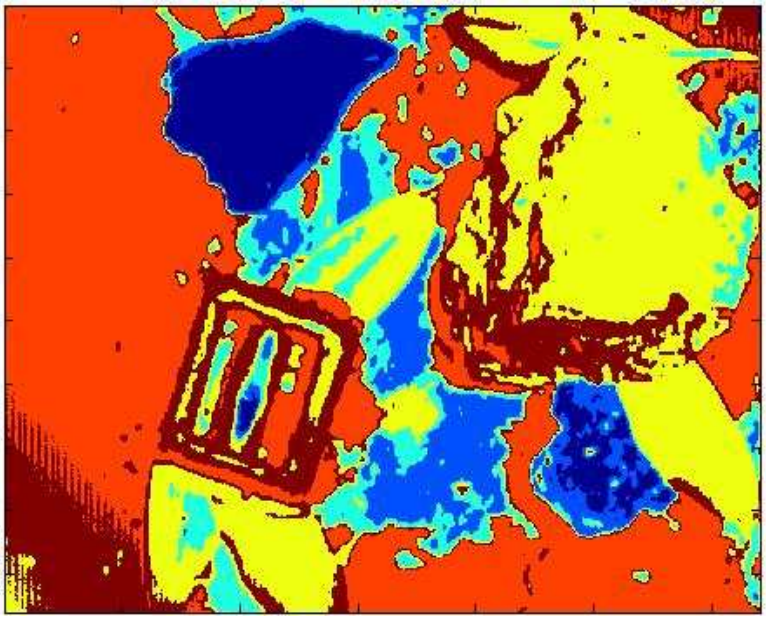

Figure 5.2: Euclidean UPRM image result. Pixels Discrepancies with Matlab: 0

	Indian Pine Site Image	UPRM Image
FIM Selected Bands (Matlab)	28,29,42	19,18,17
FIM Selected Bands (Our Application)	28,29,42	19,18,17

Table 5.1: FCA Algorithms results for both images.

5.2.2 Maximum Likelihood Accuracy

Figures 5.3 and 5.4 shows the maximum likelihood classification result for both images. There were no discrepancies between our application and the Matlab results.

5.2.3 Feedback Classification Algorithm Accuracy

Table 5.1 shows the bands selected on our application and on the Matlab code. As it could be seen, both applications got the same results. The algorithm inputs was the indian pine site image with all its bands (220) and the desire subset was 3. The classifier used was euclidean distance. The same inputs were given to the UPRM image.

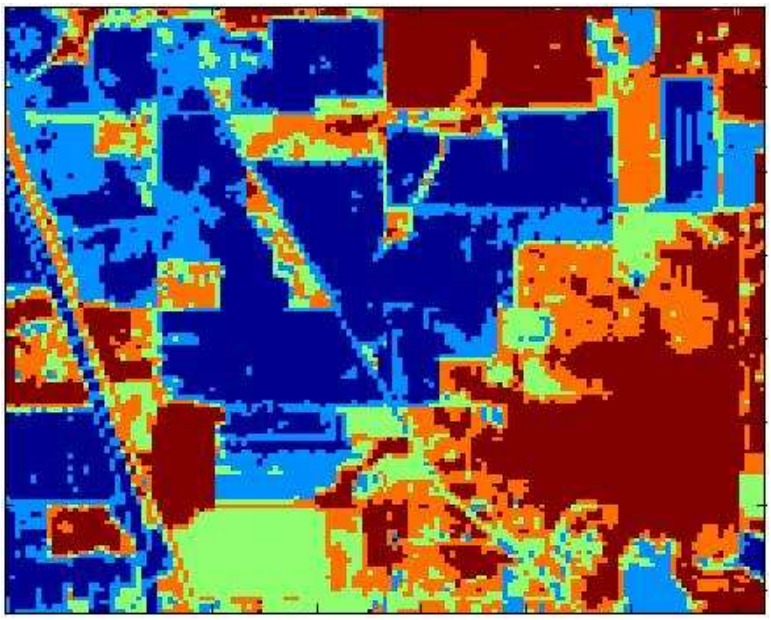

Figure 5.3: ML Indiana image result. Pixels Discrepancies with Matlab: 0

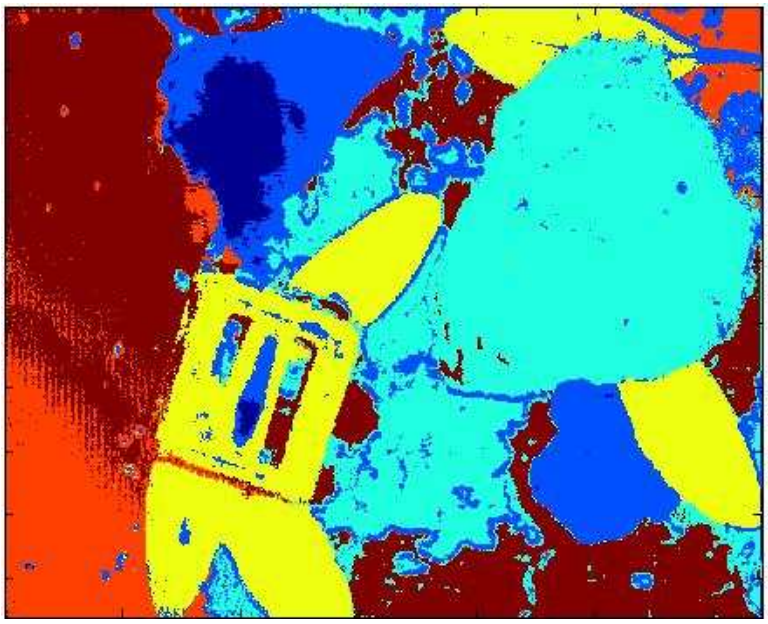

Figure 5.4: ML UPRM image result. Pixels Discrepancies with Matlab: 0

5.2.4 Principal Component Accuracy

Figures 5.5 and 5.6 shows the thematics image results using the first 10 principal components obtained from our application. The classification was done using the Maximum Likelihood classifier. Pixel discrepancies between our final classification and Matlab classification using Matlab's generated Principal Components are minimal.

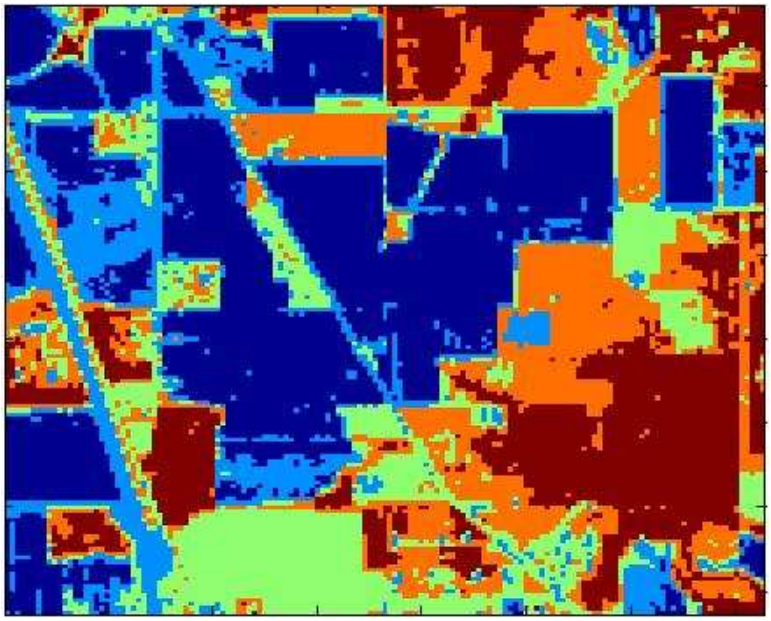

Figure 5.5: Maximum Likelihood Indiana image result using first 10 principal components. Pixels Discrepancies with Matlab: 4

5.3 Feedback Classification Algorithm

Figures 5.7 and 5.8 shows the execution times for the feedback classification algorithm on both images. Each figure also show the execution time of the sequential algorithm on each architecture. On both images we can see that the parallel implementation is highly scalable and that the parallel algorithm behaves as an ideal parallel program. However on the UPRM test we could see a U shape on the graph. This happens because the UPRM image has 33 spectral bands and the number of subset bands is 3. So when we start dis-

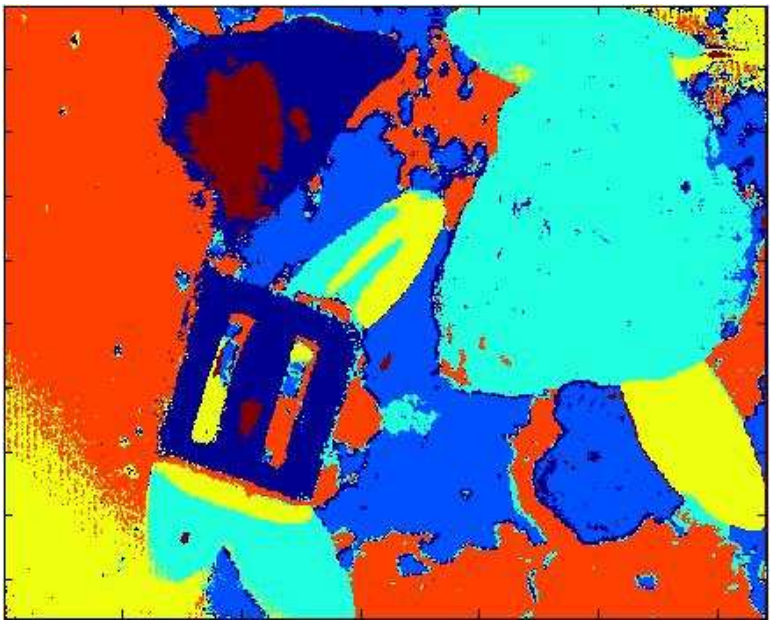

Figure 5.6: Maximum Likelihood UPRM image result using first 10 principal components. Pixels Discrepancies with Matlab: 8

tributing the combinations range, after node 11 there is no more combinations available. Thus, at that point new processors are added and they don't have any tasks to perform, but they contribute on the communications penalty. We came with Equation 5.1 which calculates the number of processors ($\boldsymbol{P}$) needed by an image to achieve only one pass of combinations distributions. $\boldsymbol{N}$ is the number of the image's spectral bands, and $\boldsymbol{m}$ is the number of subsets bands. It could be argued that when the amount of this processors are present it could represent the lowest execution time. However, when we add new processors new factors should be taken into account such as communication latency, memory, cache, etc. More nodes and images are needed in order to best characterize this behavior and suggest a general model.

$$P = ceil(N/m) \tag{5.1}$$

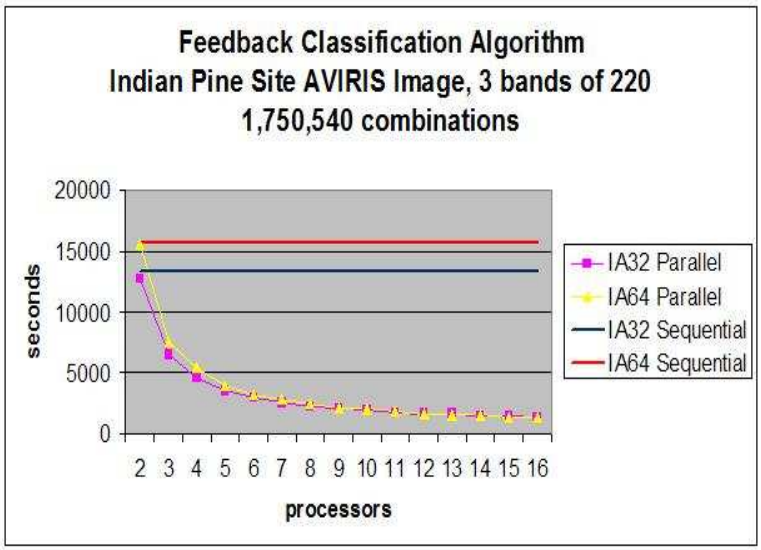

Figure 5.7: Feedback Classification Algorithm execution results for the Indiana image

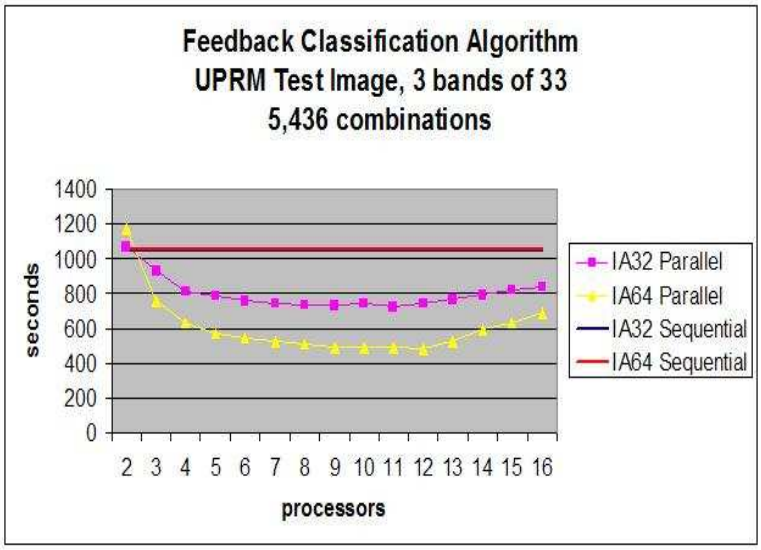

Figure 5.8: Feedback Classification Algorithm execution results for the UPRM test image

5.4 Principal Component Analysis

Principal Component parallelization provides the worst results of all the algorithms. The PLAPACK implementation results in very large execution times. The main problem we found with PLAPACK library is that there is no way to load a matrix globally. The only method the framework provide is one axpy_local_to_global, where a single node loads the 21025x220 matrix and then distribute it across nodes. This approach could work with small matrices but with the amount of data we are dealing is very time consuming and prohibitive. The same problem appears when we want to move a distributed matrix to a single node. To attack these issues we developed some methods to loads the image in a more efficient way. Basically all the nodes with a corresponding subregion of the matrix could load it in parallel. This approach speed up the process but is totally dependent on the image loaded, so a new methods has to be developed whenever a spectral image is added to the toolkit. Another method developed by the author unloads a global distributed matrix in a parallel way. Basically each node which has subregions of the matrix dumps its content to a file. Since each node knows its part of the image, the filename contains the part of the image. Then with common unix scripting, the image is gathered from all files. All the optimizations tried by the author were not sufficient, from Figure 5.9 we could observe that the parallel execution times are way over the sequential implementation.

5.5 Classifiers

5.5.1 Euclidean Distance Results

Execution results for the euclidean distance classifier provided in Figure 5.10 are very promising. On both architectures we see a highly scalable curve. Based on these, suggest that the algorithm could be used with images with more spectral bands and higher resolutions. Here we found another interesting behavior. On IA32 the parallel execution times are better than the sequential approach. This could be an expected result, however

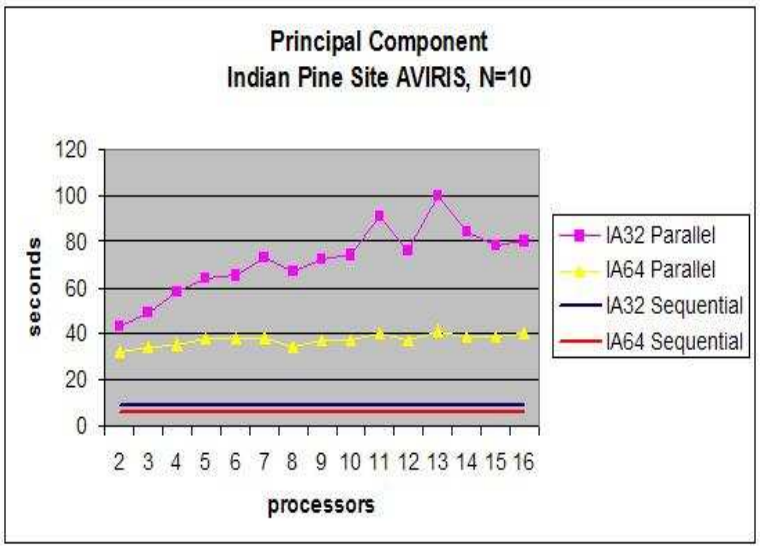

Figure 5.9: Principal Component execution results for the Indiana image.

what it is a surprise is that the sequential execution times on IA64 are so lower compared to IA32 that the parallel approach on IA32 does not compare to IA64. So for this image, it could be faster to run the algorithm sequentially on IA64 than the parallelized version on IA32.

On the UPRM test image IA64 sequential implementation has better execution times than the IA32 parallel version. Here we found that the IA32 parallel approach is not so great as the indian pine site image. This is due the image resolution of the UPRM test image is so big that the amount of data transfered is tremendous. As explained with Table 3.3 for this image it should be better to distribute the means than the classification vector. Equations 5.2 and 5.3 provides a guide to know when it will be better to broadcast the classification vector or the means. These equations calculates the size in bytes for the classification vector and for the means, if the classification vector is greater than the means, then the means should be broadcast, otherwise the vector. $\boldsymbol{C}$ is the number of classes and $\boldsymbol{N}$ is the number of the image spectral bands.

$$ClassVec = Height * Width * sizeof(int) \tag{5.2}$$

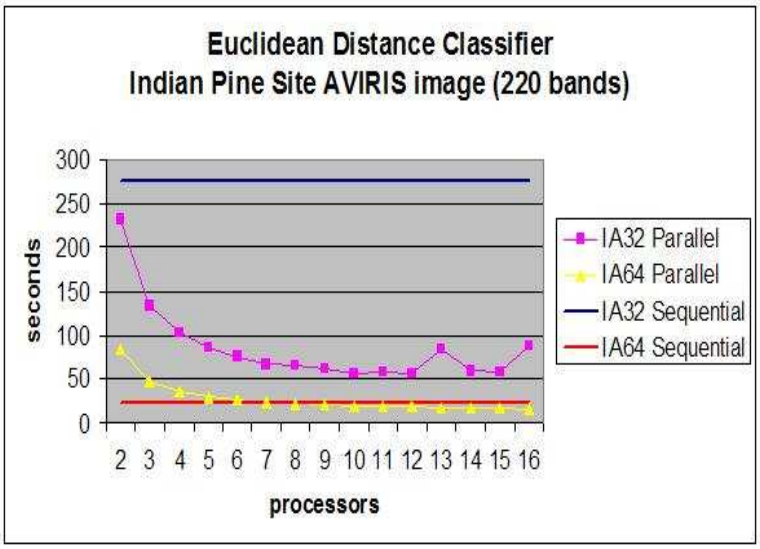

Figure 5.10: Euclidean Classifier execution times for the Indiana image.

$$Means = C * N * sizeof(double) \tag{5.3}$$

5.5.2 Maximum Likelihood Results

Results on the Maximum Likelihood are very similar to those of the Euclidean distance classifier. On Figure 5.12 we see huge execution time benefits on IA32, but at the same time the IA32 parallel version which seems highly scalable are in the same range than the IA64 sequential approach.

As explained on the euclidean distance above, in Figure 5.13 the parallel implementation is not as good as in the indian pine site image. This is due the high spatial resolution of the UPRM image make it much worse the communication process when calculating means and covariances. Here, equation 5.3 has to be modify to add the covariance size. Equation 5.4 provides the new criteria in order to know when it will be better to broadcast the classification vector or the covariances and means. Equation 5.2 is still valid for the Maximum Likelihood classifier. As in the Euclidean distance $\boldsymbol{C}$ is the number of classes and $\boldsymbol{N}$ is the number of the image spectral bands.

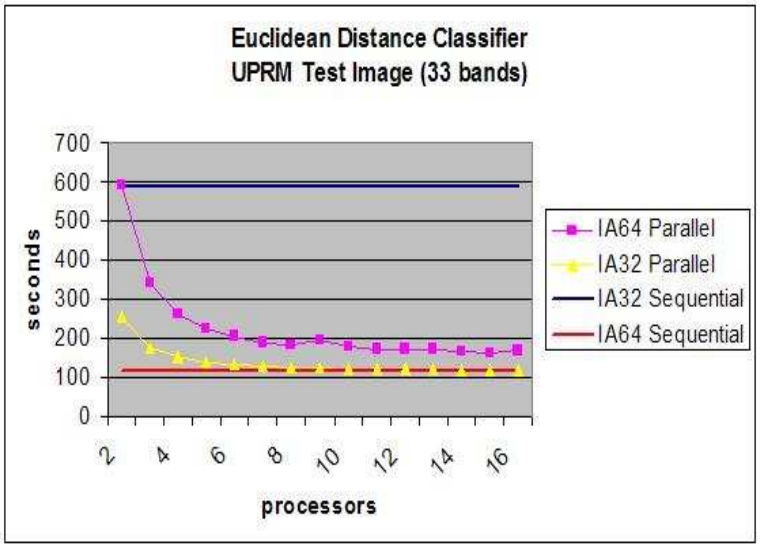

Figure 5.11: Euclidean Classifier execution times for the UPRM image.

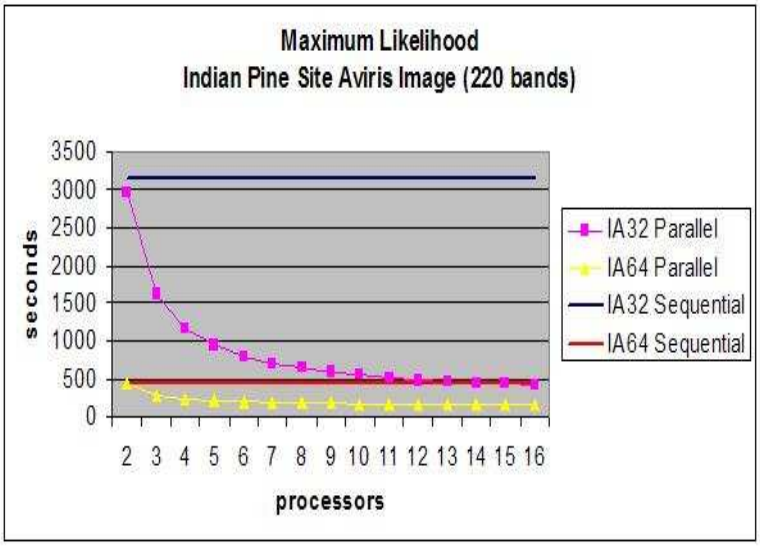

Figure 5.12: ML Classifier execution times for the Indiana image.

$$Means = N^2 * C * sizeof(double) + C * N * sizeof(double) \tag{5.4}$$

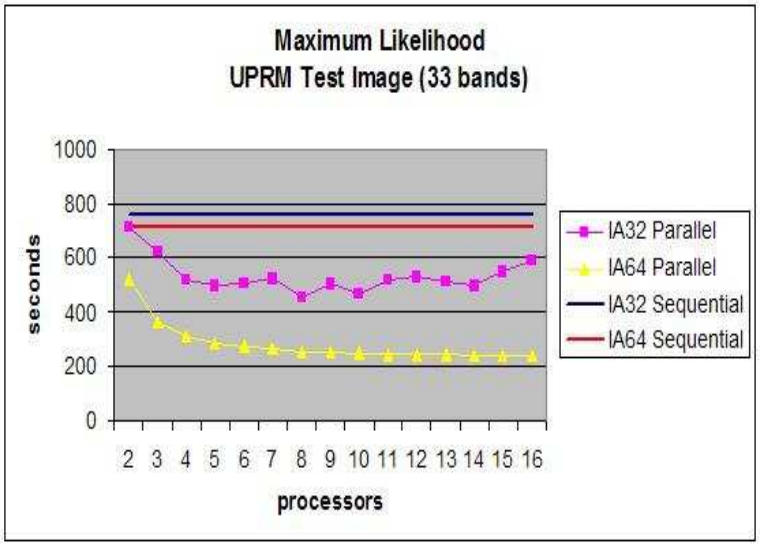

Figure 5.13: ML Classifier execution times for the UPRM image.

CHAPTER 6

Conclusion and Future Work

In this thesis we presented the parallelization of four hyperspectral imaging algorithms which are representative of the algorithms used by the remote sensing community. According to previous research on the topic, this is the first attempt to parallelized such algorithms by using a computational clusters. We also presented the parallelized algorithms behavior on two different computer architectures. Each algorithm was optimized for its specific architecture by using optimization libraries to perform the linear algebra tasks.

The Euclidean distance classifier, maximum likelihood and Feedback Classification Algorithm method were parallelized by changing the sequential algorithm behavior and exploiting its parallelism. The Principal Component Algorithm was parallelized using parallel linear algebra packages since the whole algorithm is based on mathematical transformations and there is no explicit parallelism on the operations.

6.1 Research Conclusion

1. It has been demonstrate that exhaustive search across subsets of features combinations is possible with reasonable execution times.

2. We have demonstrated that definitely hyperspectral imaging algorithms can benefits from the advantages of high performance computational clusters.

3. Parallel computation can provides the capacity to handle today and future hyperspectral imaging algorithms. Moreover three of four algorithms were scalable enough

to handle bigger spectral images in the same computational time.

4. Parallel Linear Algebra Packages such as PLAPACK, provides tremendous abstraction from the parallelization layer. However these packages lacks the optimization necessarily to attend current and future hyperspectral needs.

6.2 Future Work

As demonstrated some algorithms sequential implementations on Itanium 2 provides better performance than the parallel ones. It will be very interesting if we could use this sequential algorithms to perform the same computation on different data. The results should be fast enough but the results are very promising. Imaging that we could integrate the sequential algorithms on a cluster scheduler environment like PBS. In this approach we could classify different images using the same classifier at the same time. But most interesting is the capability to classify different bands using different classifiers and then perform some kind of data fusion to get the final results. All classifiers will work on different bands and the result will be obtained as fast as a common classification. This approach will provide an excellent environment to investigate ensembles of classifiers.

On the Principal Component Algorithm we want to port it to ScalaPack. Scalapack is highly optimized library available for IA32 and IA64 and maybe we think it will provided better results than PLAPACK.

BIBLIOGRAPHY

[1] P. Swain and S. Davis *Remote Sensing: The Quantitative Approach*. McGraw-Hill, 1993.

[2] A. S. Mazer, M. Martin, M. Lee, and J.E. Solomon *Image processing software for imaging spectrometry data analysis*. Remote Sensing of the Environment, Vol 24, No. 1, pp. 201-210, 1988.

[3] G. M. Petrie and P. G. Heasler *Optimal Selection Strategies for Hyperspectral Data Sets*. International Geoscience and Remote Sensing Symposium, 1998.

[4] T. A. Warner and M. C. Shank. *Spatial Autocorrelation Analysis of Hyperspectral Imagery for Feature Reduction*. Remote Sensing of the Environment, Vol. 60, No. 1, pp. 58-70, 1998.

[5] M. Vélez-Reyes, L. O. Jiménez, F. Pagán, and G. Fernández. *Subset Selection for the Analysis of Hyperspectral Data*. International Symposium on Spectral Sensing Research, 1998.

[6] J. C. Price. *An Approach for Analysis of Reflectance Spectra*. Remote Sensing of the Environment, Vol. 64, No. 1, pp. 316-330, 1998.

[7] T. Tu, C. Chen, J. Wu, and C. Chang. *A fast two-stage Classification Method for High-Dimensional Remote Sensing Data*. IEEE Transactions on Geoscience and Remote Sensing Vol 36, No. 1, pp. 182-191, 1998.

[8] L. O. Jiménez and D. .A Landgrebe. *Hyperspectral Data Analysis and Supervised Feauture Reduction Via Projection Pursuit*. IEEE Transactions on Geoscience and Remote Sensing Vol 37, No. 6, pp. 2653-2667, 1999.

[9] J. C. Harsanyi and C. Chang. *Hyperspectral Image Classification and Dimensionality Reduction: An Orthogonal Subspace Projection Approach*. IEEE Transactions on Geoscience and Remote Sensing, Vol 32, No. 4, pp. 779-785, 1994.

[10] A. de Bruin, G.A.P. Kinderwater, and H.W.J.M. Tirienekens *Towards an Abstract Parallel Branch and Bound Machine*. Report EUR-CS-95-05, Erasmus University, Department of Computer Science, The Netherlands 1995.

[11] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn *PLAPACK : High Performance Through High-Level Abstracton* Proceedings of International Conference on Parallel Processing pp. 414-422, 1998

[12] A. Ifarraguerri and M. W. Praire *Visual Method for Spectral Band Selection* IEEE TRansactions on Geoscience and Remote Sensing, Vol 1, No. 2, pp. 101-106, 2004

[13] N. H. Younan, R. L. King, and H. H. Bennett *Hyperspectral Data Analysis Using Wavelet-Based Classifiers* IEEE Proceedings of Geoscience and Remote Sensing Symposium Vol. 1, pp. 390-392, 2000

[14] L. O. Jímenez *INEL 6007 : Sensores Remotos* UPRM INEL 6007 Course notes, LARSIP 2000

[15] H. Du, H. Qi, X. Wang, R. Ramanath, and W. E. Snyder *Band Selection Using Independent Component Analysis for Hyperspectral Image Procesing* Proceedings of the 32nd Applied Imagery Pattern Recognition Workshop, pp. 93, October 2003

[16] M. Skinner *Genetics Algorithm Overview* http://geneticalgorithms.ai-depot.com/Tutorial/Overview.html

[17] M. Alfonseca *Genetic Algorithms* Proceedings of the international conference on APL pp. 1-6, 1991

[18] J. Ma, X. Zheng, Q. Tong, and L. Zheng *An application of genetical algorithms on band selection for hyperspectral image classification* Procedings of the Second International Conference on Machine Learning and Cybernetics pp 2810-2813, November 2003

[19] M. L. Raymer, W. F. Punch, E. D. Goodman, L. A. Kuhn, and A. K. Jain *Dimensionality Reduction Using Genetic Algorithms* IEEE Transactions on Evolutionary Computation Vol. 4, pp. 164-171, 2000

[20] S. Hunt and D. Rivera *Pattern Recognition in Hyperspectral Imagery using Feedback* Proceedings of the SPIE's 9th international Symposium on Remote Sensing pp. 359-366, 2002

[21] J. R. Sveinsson, J. A. Benediktsson and S. B. Stefansson *Classification of Event-Related Potential Waveforms with Parallel Principal Component Neural Networks* IEEE 17th Annual Conference of Engineering in Medicine and Biology Society Vol. 1, pp. 799-800 1995

[22] L. O. Jímenez and D. A. Landgrebe *Projection Pursuit for High Dimensional Feature Reduction: Parallel and Sequential Approaches* International Geoscience and Remote Sensing Symposium pp. 148-150, 1995.

[23] J. Shafer, R. Agrawal, and M. Mehta *SPRINT : A Scalable Parallel Classifier for Data Mining* Proceedings of the 22nd Very Large Data Base Conference 1996

[24] M. V. Joshi, G. Karypis, and V. Kumar *ScalParC : A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets* Proceedings of the First Merged International Parallel Processing and Parallel and Distributed Symposium pp. 573-579, 1998

[25] L. Breiman, J.H. Friedman, R. A. Olshen, and C. J. Stone *Classification and Regression Trees* Wasdworth, Belmont 1984

[26] D. I. Moldovan and C. I. Wu *Parallel Processing of a Knowledge-Based Vision System* Proceedings of 1986 ACM Fall joint computer conference pp. 269-276, 1986

[27] C. Kruengkrai and C. Jaruskulchai *A Parallel Learning Algorithm for Text Classification* Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining pp. 201-206, 2002

[28] H. Embrechts and D. Roose *A Parallel Euclidean Distance Transformation Algorithm* Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing pp. 216-222, 1993

[29] The Linux Cluster Information Center *Computational Clusters* http://lcic.org/

[30] S. M. Chai, A. Gentile, W. E. Lugo-Beauchamp, J. L. Cruz-Rivera, and D. S. Wills *Hyperspectral Image Processing Applications on the SIMD Pixel Processor for the Digital Battlefield* Proceedings IEEE Workshop on Computer Vision Beyond the Visible Spectrum : Methods and Applications pp. 130-138, 1999

[31] J. Wessels, M. Bucheit, and A. Expesset *The Develoment of High Performance, High Volume Distributed Hyperpectral Processor and Display System* IEEE International Symposium of Geoscience and Remote Sensing Vol 4, pp 2519-2521, 2002

[32] Hewlett Packard Company *HP's Mathematical Software Library (MLIB)* http://www.hp.com/go/mlib

[33] Intel Corporation *Intel Math Kernel Library Reference Manual* ftp://download.intel.com/software/products/mkl/docs/mklman.pdf

[34] The MathWorks *MATLAB and Simulink for Technical Computing* http://www.mathworks.com

[35] G. Burns, R. Daoud, and J. Vaigl *LAM : An open cluster environment for MPI* Proceedings of Supercomputing Symposium pp. 379-386, 1994

[36] W. Groop and E. Lusk *The MPI communication library : its design and a portable implementation* Proceedings of the Scalable and Parallel Libraries Conferences pp. 160-165, 1993

[37] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh *Basic Linear Algebra Subprograms for FORTRAN Usage* ACM Transactions on Mathematical Software Vol. 5, No. 3, pp. 308-323, 1979

[38] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson *An extended set of FORTRAN basic linear algebra subprograms* ACM Transactions on Mathematical Software Vol. 14, No. 1, pp. 1-17, 1988

[39] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff *A set of level 3 basic linear algebra subprograms* ACM Transactions on Mathematical Software Vol. 16, No. 1, pp. 1-17, 1990

[40] S. Filippone and M. Colajanni *PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices* ACM Transactions on Mathematical Software Vol. 26, No. 4, pp. 527-550, 2000

[41] J. Choi, J. Demmel, I. Dhillon, J. J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. W. Walker, and R. C. Whaley. *ScaLAPACK: A portable linar algebra library for distributed memory computers - design issues and performance* Proceedings of the seconds international workshop on Applied Parallel Computing pp. 95-106, 1995

[42] GCC: The GNU C Compiler *GCC: The GNU C Compiler Home Page* http://www.gnu.org/software/gcc/

[43] GDB: The GNU Project Debugger *GDB: The GNU Project Debugger Home Page* http://www.gnu.org/software/gdb/

[44] S. Graham, P. Kessler, and M. McKusick *gprof: A call Graph Execution Profiler* Proceedings of the Symposium on Compiler Construction Vol. 17, No. 6, pp. 120-126, 1982

[45] The Memcheck Deluxe Memory Tracking Program *The MemCheckDeluxe Home Page* http://prj.softpixel.com/mcd/

APPENDICES

APPENDIX A

IA32 Setup Environment

In this section we described in detailed how to setup the IA32 development environment. Since optimizations libraries were used, the reader should handle the installation of these libraries as documented below. If this procedure is follow the developer could simply start building the application suite without any modification to the makefile.

First of all you noticed on section 4.3 that we have one makefile for each architecture. Since we are using different optimizations libraries and different compilers for each architecture, each makefile contain different linking libraries and compiler options. Once the code is downloaded from the repository a symbolic link should be made to the proper Makefile architecture (ln -s Makefile.arch Makefile). Below are the instructions for the library installations.

A.1 Installing Intel MKL Library

The Intel Math Kernel Library is obtained from the Intel web site at http://www.intel.com/software/products/mkl/. This library is not free and it needs a license before it could be installed. However Intel provides a non-commercial license for those who qualify. In order to qualify for a non commercial license the following criteria should be satisfy: I should be used for personal non-commercial purposes and there is no support associated with the software.

Once the license is in place, the Math Kernel Library can be installed. Is very

important that the library is installed on all nodes in the computational cluster, so each node should have a license. Another important step is to install the libraries on the default directory (/opt/intel).

A.2 Installing LAM

In order to use the Message Passing Library (MPI) the LAM package should be installed. On some systems the MPICH package is already installed, since there could be problems by using both packages our recommendation is to uninstall mpich and install LAM. I think the MPICH package could work, but I haven't tested it. So to be sure I recommend installing LAM.

A.3 Installing PLAPACK

The Parallel Linear Algebra PACKage (PLAPACK) could be obtained from http://www.cs.utexas.edu/users/plapack/. The package should be decompressed on the $HOME of the account that will be running the programs. Once the file is decompressed and the $HOME/PLAPACKR30/ has been created we need to replace the Make.include file. We need to copy the file Make.includes/Make.include.linux to Make.include on the PLAPACKR30 directory. Once the file is in place we need to change the BLASLIB and SEQ_LAPACK variables to the following:

```
BLASLIB = /opt/intel/mkl70/lib/32/libmkl_p3.so\
        /opt/intel/mkl70/lib/32/libguide.a
SEQ_LAPACK = /opt/intel/mkl70/lib/32/libmkl_lapack.a
```

After that the Make.include files points to the proper libraries we can then compile the PLAPACK library by using the ***make*** command. Since the library will be located on the users HOME directory and we are assuming a shared storage across the nodes, there is no need to install the library on all the nodes.

A.4 Updating Library Path

In order the Math Kernel Libraries could be used at runtime, these libraries should be added to the global library search path. This is done by adding the /opt/intel/mkl70/lib/32 path to the /etc/ld.so.conf file, then the ***ldconfig*** command. This has to to be performed on all the nodes.

APPENDIX B

IA64 Setup Environment

In this section we described in detailed how to setup the IA64 development environment. Since optimizations libraries were used, the reader should handle the installation of these libraries as documented below. If this procedure is follow the developer could simply start building the application suite without any modification to the makefile.

First of all you noticed on section 4.3 that we have one makefile for each architecture. Since we are using different optimizations libraries and different compilers for each architecture, each makefile contain different linking libraries and compiler options. Once the code is downloaded from the repository a symbolic link should be made to the proper Makefile architecture (ln -s Makefile.arch Makefile). Below are the instructions for the library installations.

B.1 Installing HP-MLIB Library

The HP Mathematical Library is obtained from the HP web site at http://www.hp.com/go/mlib/. The library could be installed without a license but there is a performance penalty if it used without a license. All of the execution times presented on this thesis were gathered using the non-license version of the libraries.

The HP MLIB compressed file contains different RPMS for the different libraries. The only ones needed are the LAPACK and VECLIB ones. However it does not affect to install them all.

B.2 Installing Intel Fortran Compiler

The HP MLIB were compiled using the Intel Fortran Compiler. Thus it needs some of the compilers libraries in order to resolve all symbols.

The Intel Fortran Compiler is obtained from the Intel web site at http://www.intel.com/software/products/compilers/flin/. This compiler is not free and it needs a license before it could be installed. However Intel provides a non-commercial license for those who qualify. In order to qualify for a non commercial license the following criteria should be satisfy: I should be used for personal non-commercial purposes and there is no support associated with the software.

Once the license is in place, the Fortran Compiler can be installed. Is very important that the library is installed on all nodes in the computational cluster, so each node should have a license. Another important step is to install the compiler on the default directory (/opt/intel).

B.3 Installing LAM

In order to use the Message Passing Library (MPI) the LAM package should be installed. On some systems the MPICH package is already installed, since there could be problems by using both packages our recommendation is to uninstall mpich and install LAM. I think the MPICH package could work, but I haven't tested it. So to be sure I recommend installing LAM.

B.4 Installing PLAPACK

The Parallel Linear Algebra PACKage (PLAPACK) could be obtained from http://www.cs.utexas.edu/users/plapack/. The package should be decompressed on the $HOME of the account that will be running the programs. Once the file is decompressed and the $HOME/PLAPACKR30/ has been created we need to replace the Make.include file. We need to copy the file Make.includes/Make.include.linux to Make.include on the

PLAPACKR30 directory. Once the file is in place we need to change the BLASLIB and SEQ_LAPACK variables to the following:

```
BLASLIB = /opt/mlib/lib/linux/libveclib.a
SEQ\_LAPACK = /opt/mlib/lib/linux/liblapack.a
```

After that the Make.include files points to the proper libraries we can then compile the PLAPACK library by using the ***make*** command. Since the library will be located on the users HOME directory and we are assuming a shared storage across the nodes, there is no need to install the library on all the nodes.

B.5 Updating Library Path

In order the Math Kernel Libraries could be used at runtime, these libraries should be added to the global library search path. This is done by adding following paths:
/opt/intel/lib
/opt/mlib/lib/linux
/opt/intel/compiler70/ia64/lib
After the entries have been added the ***ldconfig*** command should be executed. This has to to be performed on all the nodes.