High-Performance Computing in Remotely Sensed Hyperspectral Imaging: The Pixel Purity Index Algorithm as a Case Study

Antonio Plaza, David Valencia and Javier Plaza

Computer Architecture and Technology Section
Computer Science Department, University of Extremadura
Avda. de la Universidad s/n, E-10071 Caceres, SPAIN
Phone: +34 (927) 257195; Fax: +34 (927) 257203
E-mail: aplaza@unex.es

Abstract

The incorporation of last-generation sensors to airborne and satellite platforms is currently producing a nearly continual stream of high-dimensional data, and this explosion in the amount of collected information has rapidly created new processing challenges. For instance, hyperspectral imaging is a new technique in remote sensing that generates hundreds of spectral bands at different wavelength channels for the same area on the surface of the Earth. The price paid for such a wealth of spectral information available from latest-generation sensors is the enormous amounts of data that they generate. In recent years, several efforts have been directed towards the incorporation of high-performance computing (HPC) models in remote sensing missions. This paper explores three HPC-based paradigms for efficient information extraction from remote sensing data using the Pixel Purity Index (PPI) algorithm (available from the popular Kodak's Research Systems ENVI software) as a case study for algorithm optimization. The three considered approaches are: 1) Commodity cluster-based parallel computing; 2) Distributed computing using heterogeneous networks of workstations; and 3) FPGA-based hardware implementations. Combined, these parts deliver an excellent snapshot of the state-of-the-art in those areas, and offer a thoughtful perspective on the potential and emerging challenges of adapting HPC models to remote sensing problems.

1. Introduction

Hyperspectral imaging (also known as imaging spectroscopy) is a new technique that has gained tremendous popularity in many research areas, most notably, in remotely sensed satellite imaging and aerial reconnaissance [1]. Recent advances in sensor technology have led to the development of so-called hyperspectral instruments, which are capable of collecting hundreds of images corresponding to different wavelength channels for the same area on the surface of the Earth (see Fig. 1). In particular, NASA is continuously gathering imagery data with hyperspectral Earth-observing sensors such as Jet Propulsion Laboratory's Airborne Visible-Infrared Imaging Spectrometer (AVIRIS) [2], or the Hyperion imager aboard Earth Observing-1 (EO-1) spacecraft. The incorporation of hyperspectral sensors on airborne/satellite platforms is currently producing a nearly continual stream of high-dimensional data (it is estimated that NASA collects and sends to Earth more than 850 Gb of hyperspectral data every day).

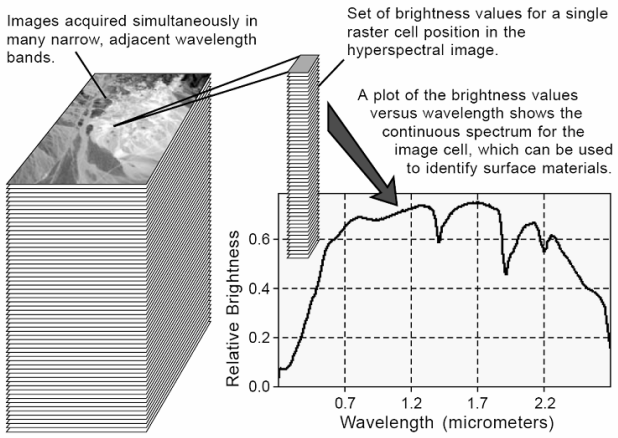

Figure 1. The concept of hyperspectral imaging

The development of computationally efficient techniques for transforming the massive amount of hyperspectral data collected on a daily basis into scientific understanding is critical for space-based Earth science and planetary exploration. In particular, the wealth of spatial and spectral information provided by last-generation hyperspectral instruments has opened ground-breaking perspectives in many applications, including environmental modeling and assessment, target detection for military and defense/security purposes, urban planning and management studies, risk/hazard prevention and

1-4244-0054-6/06/$20.00

response including wild land fire tracking, biological threat detection, monitoring of oil spills and other types of chemical contamination, etc. Most of the above-cited applications require analysis algorithms able to provide a response in real- or near real-time.

In recent years, several efforts have been directed towards the incorporation of high-performance computing (HPC) models in remote sensing applications [3-5]. Despite the growing interest in the development of HPC techniques for hyperspectral imaging, only a few research efforts devoted to the design of parallel implementations currently exist in the open literature [6,7]. It should be noted that several existing parallel techniques are subject to non-disclosure restrictions, mainly due to their use in military and defense applications. However, with the recent explosion in the amount of hyperspectral imagery, parallel processing is expected to become a requirement in virtually every remote sensing application. As a result, this paper takes a necessary first step toward the comparison of different HPC techniques and strategies for parallel hyperspectral image analysis.

As a case study, this paper focuses on the pixel purity index (PPI) algorithm, one of the most widely used standard algorithms in the hyperspectral imaging community. The algorithm was originally developed by Boardman et al. [8], and was soon incorporated into Kodak's Research Systems ENVI [9], a world-class commercial software package for remote sensing. Due to the algorithm's propriety and limited published results, its detailed implementation has never been made available in the public domain. This paper presents our experience with the PPI algorithm and investigates several strategies for its implementation in parallel. The description of several techniques and strategies for parallelization of the PPI algorithm provides an excellent snapshot of the state-of-the-art of the application of HPC models in remote sensing applications, and an in-depth study of a well-known commercial algorithm that will appeal to both practitioners and developers alike, thus providing a thoughtful perspective on the potential of applying HPC paradigms in remote sensing missions.

The paper is structured as follows. Section 2 reviews the PPI and discusses several issues encountered in its implementation. Section 3 develops several high-performance implementations, including: 1) a commodity cluster-based parallel implementation; 2) a distributed implementation on heterogeneous networks of workstations; and 3) a hardware-based implementation using FPGAs. Section 4 provides an experimental comparison of the proposed parallel implementations. Here, we use a massively parallel Beowulf cluster at NASA's Goddard Space Flight Center, a heterogeneous network of workstations at University of Maryland, and a Xilinx Virtex-II FPGA device. Finally, section 5 concludes with some remarks and future research lines.

2. Pixel Purity Index Algorithm

The underlying assumption under the PPI algorithm is that the spectral signature associated to each pixel vector (see Fig. 1) measures the response of multiple underlying materials at each site. For instance, it is very likely that the pixel vector shown in Fig. 1 would actually contain a mixture of different substances (e.g., different minerals, different types of soils, etc.). This situation, often referred to as the "mixture problem" in hyperspectral analysis terminology [10], is one of the most crucial and distinguishing properties of spectroscopic analysis. To deal with this problem, spectral unmixing techniques have been proposed as a procedure in which the measured spectrum of a mixed pixel is decomposed into a collection of spectrally pure constituent spectra, called endmembers in the literature [11], and a set of correspondent fractions, or abundances, that indicate the proportion of each endmember present in the mixed pixel.

The PPI algorithm falls into the category of endmember extraction algorithm [8]. It was designed to search for a set of vertices of a convex hull in a given data set that are supposed to represent the purest signatures present in the data. The algorithm proceeds by generating a large number of random, N-dimensional (N-D) unit vectors called "skewers" through the dataset. Every data point is projected onto each skewer, and the data points that correspond to extrema in the direction of a skewer are identified and placed on a list. As more skewers are generated, the list grows, and the number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the final endmembers, where the number of endmembers to be extracted by the algorithm, p, is set in advance by the user. Two additional parameters are input to the algorithm: k, the number of random skewers to be generated during the process, and t, a cut-off threshold value that is used to select as final endmembers only those pixels that have been selected as extreme pixels at least t times throughout the PPI process.

Although the algorithmic description of the PPI algorithm has never been made fully disclosed in the open literature, we provide below an outline of the algorithm which is based on limited published results and our own interpretation. All steps in the algorithm below have been verified via experiments using the PPI available in Research Systems ENVI version 4.0 and our own implementation, where both versions produced exactly the same results. The inputs to the algorithm are an N-D data cube, $\mathbf{F}$, the number of endmembers to be extracted, p, the number of skewers to be generated throughout the process, k, and a cut-off threshold value t. The output of the algorithm is a set of final endmembers, $\{\mathbf{e}_i\}_{i=1}^{p}$. Below, we provide a step-by-step description of the classic PPI algorithm:

1. *Skewer generation*. Produce a set of k randomly generated unit vectors called "skewers", $\{\mathbf{skewer}_j\}_{j=1}^{k}$.
2. *Extreme projections*. For each $\mathbf{skewer}_j$, all the data sample vectors in $\mathbf{F}$ are projected onto $\mathbf{skewer}_j$ via dot products of $\mathbf{f} \cdot \mathbf{skewer}_j$ to find sample vectors at its extreme (maximum and minimum) projections, to form an extrema set for the skewer $\mathbf{skewer}_j$ denoted by $S_{extrema}(\mathbf{skewer}_j)$. Despite the fact that a different $\mathbf{skewer}_l$ generates a different extrema set $S_{extrema}(\mathbf{skewer}_l)$, it is very likely that some sample vectors may appear in more than one extrema set. Define an indicator function of a set S, $I_S(\mathbf{f})$ by: $I_S(\mathbf{f}) = \begin{cases} 1; \text{if } \mathbf{f} \in S \\ 0; \text{if } \mathbf{f} \notin S \end{cases}$ and $N_{PPI}(\mathbf{f}) = \sum_j I_{S_{extrema}(\mathbf{skewer}_j)}(\mathbf{f})$, where $N_{PPI}(\mathbf{f})$ denotes the PPI score associated to the sample pixel vector $\mathbf{f}$ of the input image $\mathbf{F}$.
3. *Endmember selection*. Find the pixel vectors with scores of $N_{PPI}(\mathbf{f})$ above t, and form a unique set of endmembers $\{\mathbf{e}_i\}_{i=1}^{p}$ by calculating the spectral angle distance (SAD) [1] for all possible pixel vector pairs, where the SAD is given by $\mathrm{SAD}[\mathbf{f}, \mathbf{f}'] = \cos^{-1}\left[\mathbf{f}, \mathbf{f}' / \|\mathbf{f}\| \cdot \|\mathbf{f}'\|\right]$.

In order to set parameter values for the PPI, the authors recommend using as many random skewers as possible in order to obtain optimal results. As a result, the PPI can only guarantee to produce optimal results asymptotically and its computational complexity is very high. According to our experiments with the hyperspectral data set in Fig. 1, which comprises 614x512 pixels and 224 spectral bands, the algorithm required $k = 10^4$ skewers to find all spectral endmembers present in the scene, and the whole process took more than 50 minutes to project every data sample vector of the input data into the 10^4 skewers in a PC with AMD Athlon 2.6 GHz processor and 512 MB of RAM [12]. This response time is unacceptable in most remote sensing applications.

3. High-Performance Implementations

This section develops three HPC-based parallel implementations for the PPI algorithm. The first one is a massively parallel implementation designed for data mining and information extraction from large data archives, such as the AVIRIS data repository at NASA's Jet Propulsion Laboratory which contains of several Terabytes of hyperspectral data. This implementation assumes homogeneity in the parallel computing platform. Quite opposite, the second implementation was specifically designed for distributed environments characterized by their heterogeneity, both in the processing elements and communication links. Finally, we also develop an FPGA-based implementation which is intended for onboard, real-time data processing.

3.1. Cluster-Based Parallel Implementation

In this subsection, we describe a master-slave parallel implementation of the PPI algorithm. To reduce code redundancy and enhance reusability, our goal was to reuse much of the code for the sequential algorithm in the parallel implementation. For that purpose, we adopted a spatial-domain decomposition approach that subdivides the image cube into multiple blocks made up of entire pixel vectors, and assigns one or more blocks to each processing element. It should be noted that the PPI algorithm described in Section 2 is mainly based on projecting pixel vectors which are always treated as a whole spectral signature. Therefore, a spectral-domain partitioning scheme (which subdivides the whole multi-band data into blocks made up of contiguous spectral bands or sub-volumes, and assigns one or more sub-volumes to each processing element) does not seem appropriate in our application. This is because the latter approach breaks the spectral identity of the data because each pixel vector is split amongst several processing element. A further reason that justifies the above decision is that, in spectral-domain partitioning, the calculations made for each hyperspectral pixel need to originate from several processing elements, and thus require intensive inter-processor communication [13]. Therefore, in our proposed implementation, a master-worker spatial domain-based decomposition paradigm is adopted, where the master processor sends partial data to the workers and coordinates their actions. Then, the master gathers the partial results provided by the workers and produces a final result. A description of the homogeneous cluster-based implementation of PPI is given below:

1. *Data partitioning*. Produce a set of L spatial-domain homogeneous partitions of $\mathbf{F}$ and scatter all partitions by indicating all partial data structure elements which are to be accessed and sent to each of the workers.
2. *Skewer generation*. Generate a set of k randomly generated unit vectors called "skewers", $\{\mathbf{skewer}_j\}_{j=1}^{k}$, and broadcast the entire set of skewers to the workers.
3. *Extreme projections*. For each $\mathbf{skewer}_j$, project all the data sample vectors at each local partition l onto $\mathbf{skewer}_j$ to find sample vectors at its extreme projections, and form an extrema set for $\mathbf{skewer}_j$ denoted by $S^{(l)}(\mathbf{skewer}_j)$. Calculate $N_{PPI}^{(l)}(\mathbf{f}^{(l)}) = \sum_j I_{S^{(l)}(\mathbf{skewer}_j)}(\mathbf{f}^{(l)})$ for each $\mathbf{f}^{(l)}$ at the

local partition using equation (1). Select those pixels with $N_{PPI}^{(l)}(\mathbf{f}^{(l)}) > t$ and send them to the master node.

4. *Endmember selection*. The master gathers all the endmember sets provided by the workers and forms a unique set $\{\mathbf{e}_i\}_{i=1}^{p}$ by calculating the SAD for all possible pixel vector pairs in parallel.

It should be noted that the proposed parallel algorithm has been implemented in the C++ programming language, using calls to message passing interface (MPI). We emphasize that, in order to implement step one of the parallel algorithm, we resorted to MPI *derived datatypes* to directly scatter hyperspectral data structures, which may be stored non-contiguously in memory, in a single communication step. As a result, we avoid creating all partial data structures on the root node (thus making a better use of memory resources and compute power).

3.2. Implementation on Heterogeneous Networks

Before introducing our implementation, we first formulate a general optimization problem in the context of fully heterogeneous systems (composed of different-speed processors that communicate through links at different capacities). Such a computing platform can be modeled as a complete graph G, where each node models a computing resource p_i weighted by its relative cycle-time w_i. Each edge models a communication link weighted by its relative capacity, where c_{ij} denotes the maximum capacity of the slowest link in the path of physical communication links from p_i to p_j. We also assume that the system has symmetric costs: $c_{ij} = c_{ji}$, and denote by W the workload to be performed by the PPI algorithm. Processor p_i will accomplish a share of $\alpha_i \cdot W$ of the workload, where $\alpha_i \geq 0$ for $1 \leq i \leq L$ and $\sum_{i=1}^{L} \alpha_i = 1$. With the above assumptions in mind, an abstract view of our problem can be simply stated in the form of a master-worker architecture, much like the homogeneous implementation described in the previous subsection. However, in order for such parallel algorithm to be also effective in fully heterogeneous systems, the master program must be modified to produce a set of L spatial-domain heterogeneous partitions of $\mathbf{F}$ in step 1. Below, we provide a description of such step in the implementation for heterogeneous platforms:

1. Generate necessary system information, i.e., number of available processors L, each processor's $\{p_i\}_{i=1}^{L}$ identification number, and their cycle-times $\{w_i\}_{i=1}^{L}$.
2. Set $\alpha_i = \left\lfloor \frac{(L/w_i)}{\sum_{i=1}^{L}(1/w_i)} \right\rfloor$ for all $i \in \{1, \cdots, L\}$.
3. For $m = \sum_{i=1}^{L} \alpha_i$ to W, find $k \in \{1, \cdots, L\}$ such that $w_k \cdot (\alpha_k + 1) = \min\{w_i \cdot (\alpha_i + 1)\}_{i=1}^{L}$ and set $\alpha_k = \alpha_k + 1$.
4. 4. Use $\{\alpha_i\}_{i=1}^{L}$ to obtain a set of L spatial-domain heterogeneous partitions of $\mathbf{F}$, so that the spectral channels corresponding to the same pixel vector are never stored in different partitions.

A homogeneous version of the algorithm above can be simply obtained by replacing step 3 with $\alpha_i = L/w_i$ for all $i \in \{1,..., L\}$, where w_i is a constant cycle-time for all processors in the homogeneous system. This version was used to produce the set of L spatial-domain homogeneous partitions of $\mathbf{F}$ used by the standard parallel implementation of the PPI described in the previous subsection.

3.3. FPGA Implementation

Our hardware-based strategy is aimed at enhancing replicability and reusability of slices in FPGA devices through the utilization of systolic array design [14]. One of the main advantages of systolic array-based implementations is that they are able to provide a systematic procedure for system design that allows for the derivation of a well defined processing element-based structure and an interconnection pattern which can then be easily ported to real hardware configurations. Using this procedure, we can also calculate the data dependences prior to the design, and in a very straightforward manner. We intend to maximize computational power of the hardware and minimize the cost of communications. These goals are particularly relevant in our specific application, where hundreds of data values will be handled for each intermediate result, a fact that may introduce problems related with limited resource availability and inefficiencies in hardware replication and reusability. After several empirical experiments using real data sets, we have opted for the configuration illustrated in Fig. 2.

In our proposed design, local results remain static at each processing element, while pixel vectors are input to the systolic array from top to bottom and skewer vectors are fed to the systolic array from left to right. The processing nodes labeled as "dot" in Fig. 2 perform the individual products for the skewer projections, while the nodes labeled as "max" and "min" respectively compute the maxima and minima projections after the dot products have been completed (asterisks represent delays). In Fig. 2, $s_j^{(i)}$ denotes the reflectance value of the *j*-th band of the *i*-th skewer, with $i \in \{1,..., k\}$ and $j \in \{1,..., b\}$, where *b* is the number of bands. Similarly, $f_j^{(m)}$ denotes the reflectance value of the *j*-th band of the *m*-th pixel, with $m \in \{1,..., p\}$, where *p* is the number of pixels.

The synthesis was performed using Handel-C, a design and prototyping language that allows using a pseudo-C programming style [15]. The implementation was compiled and transformed into an EDIF specification automatically by using the DK3.1 software package [16]. We also used other tools such as Xilinx ISE 6.1i to carry out automatic place and route (PAR), and to adapt the final steps of the hardware implementation to the Virtex-II XC2V6000-6 FPGA used in experiments.

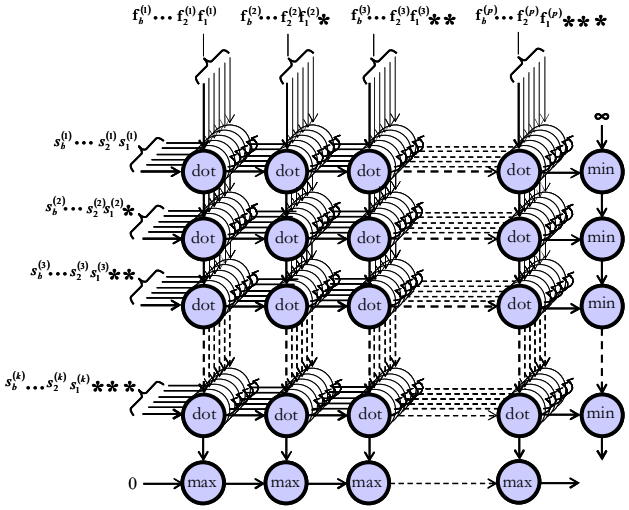

Figure 2. Systolic array design for the proposed FPGA implementation of the PPI algorithm.

4. Experimental Results

This section provides an assessment of the effectiveness of the parallel algorithms described in section 3. Before describing our study on performance analysis, we first describe the HPC computing architectures used in this work. Then, a detailed survey on algorithm performance in a real application is provided, along with a discussion on the advantages and disadvantages of each particular approach.

4.1. Parallel Computing Architectures

The first HPC system considered in experiments is Thunderhead, a 512-processor homogeneous Beowulf cluster located at NASA's Goddard Space Flight Center (GSFC) in Maryland [17]. It is composed of 256 dual 2.4 GHz Intel Xeon nodes, each with 1 GB of memory and 80 GB of main memory. The total peak performance of the system is 2457.6 GFlops. Along with the 512-processor computer core, Thunderhead has several nodes attached to the core with 2 Ghz optical fibre Myrinet. The operating system used at the time of experiments was Linux RedHat 8.0, and MPICH was the message-passing library used.

Despite the computational power offered by massively parallel systems such as Thunderhead, a current design trend at GSFC and other NASA centers is to exploit low-cost heterogeneous networks of workstations made up of commodity components, able to operate in distributed environments. To explore the performance of the proposed parallel algorithms in heterogeneous networks, we have considered two networks of workstations in this work. The first one is a small-scale, distributed heterogeneous network of 16 different SGI, Solaris and Linux workstations, and four communication segments at University of Maryland. Table 1 shows the cycle-times of the heterogeneous processors, where processors $\{p_i\}_{i=1}^{4}$ are attached to communication segment s_1, processors $\{p_i\}_{i=5}^{8}$ communicate through s_2, processors $\{p_i\}_{i=9}^{10}$ are interconnected via s_3, and processors $\{p_i\}_{i=11}^{16}$ share the communication segment labelled as s_4. For illustrative purposes, Table 2 also shows the capacity of all point-to-point communications, expressed as the time in milliseconds to transfer a one-megabit message between each processor pair (p_i, p_j) in the heterogeneous system. The communication network of the heterogeneous platform consists of four relatively fast homogeneous communication segments interconnected by three slower communication links with capacities $c^{(1,2)} = 29.05$, $c^{(2,3)} = 48.31$, $c^{(3,4)} = 58.14$ in milliseconds, respectively.

To evaluate the performance of parallel algorithms in the heterogeneous network above, we resort to a recently proposed framework [18] based on the utilization of a homogeneous network equivalent to the heterogeneous one under the following principles: 1) both environments have the same number of processors; 2) the speed of each processor in the homogeneous environment is equal to the average speed of processors in the heterogeneous environment; and 3) the aggregate communication characteristics of the homogeneous environment are the same as those of the heterogeneous environment. In this work, we have considered a homogeneous network (equivalent to the heterogeneous one) composed of 16 identical Linux workstations with processor cycle-time of $w = 0.0131$ seconds per megaflop, interconnected via a homogeneous network with capacity $c = 77.90$ milliseconds.

Finally, our proposed hardware-based systolic array design was implemented on a Virtex-II XC2V6000-6 FPGA of the Celoxica's ADMXRC2 board. It contains 33,792 slices, 144 Select RAM Blocks and 144 multipliers (of 18-bit x 18-bit). Concerning the timing performances, we decided to pack the input/output registers of our implementation into the input/output blocks (IOB) in order to try and reach the achievable performance.

p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}
0.0072	0.0102	0.0206	0.0072	0.0102	0.0058	0.0072	0.0102	0.0072	0.0451	0.0131	0.0131	0.0131	0.0131	0.0131	0.0131

Table 1. Processor cycle-times (in seconds per megaflop) for the heterogeneous cluster.

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}	p_{16}
p_1		19.26	19.26	19.26	48.31	48.31	48.31	48.31	96.62	96.62	154.76	154.76	154.76	154.76	154.76	154.76
p_2	19.26		19.26	19.26	48.31	48.31	48.31	48.31	96.62	96.62	154.76	154.76	154.76	154.76	154.76	154.76
p_3	19.26	19.26		19.26	48.31	48.31	48.31	48.31	96.62	96.62	154.76	154.76	154.76	154.76	154.76	154.76
p_4	19.26	19.26	19.26		48.31	48.31	48.31	48.31	96.62	96.62	154.76	154.76	154.76	154.76	154.76	154.76
p_5	48.31	48.31	48.31	48.31		17.65	17.65	17.65	48.31	48.31	106.45	106.45	106.45	106.45	106.45	106.45
p_6	48.31	48.31	48.31	48.31	17.65		17.65	17.65	48.31	48.31	106.45	106.45	106.45	106.45	106.45	106.45
p_7	48.31	48.31	48.31	48.31	17.65	17.65		17.65	48.31	48.31	106.45	106.45	106.45	106.45	106.45	106.45
p_8	48.31	48.31	48.31	48.31	17.65	17.65	17.65		48.31	48.31	106.45	106.45	106.45	106.45	106.45	106.45
p_9	96.62	96.62	96.62	96.62	48.31	48.31	48.31	48.31		16.38	58.14	58.14	58.14	58.14	58.14	58.14
p_{10}	96.62	96.62	96.62	96.62	48.31	48.31	48.31	48.31	16.38		58.14	58.14	58.14	58.14	58.14	58.14
p_{11}	154.76	154.76	154.76	154.76	106.45	106.45	106.45	106.45	58.14	58.14		14.05	14.05	14.05	14.05	14.05
p_{12}	154.76	154.76	154.76	154.76	106.45	106.45	106.45	106.45	58.14	58.14	14.05		14.05	14.05	14.05	14.05
p_{13}	154.76	154.76	154.76	154.76	106.45	106.45	106.45	106.45	58.14	58.14	14.05	14.05		14.05	14.05	14.05
p_{14}	154.76	154.76	154.76	154.76	106.45	106.45	106.45	106.45	58.14	58.14	14.05	14.05	14.05		14.05	14.05
p_{15}	154.76	154.76	154.76	154.76	106.45	106.45	106.45	106.45	58.14	58.14	14.05	14.05	14.05	14.05		14.05
p_{16}	154.76	154.76	154.76	154.76	106.45	106.45	106.45	106.45	58.14	58.14	14.05	14.05	14.05	14.05	14.05	

Table 2. Capacity of links (measured in the time in miliseconds to transfer a one-megabit message) for the heterogeneous network.

4.2. Performance Evaluation

The parallel implementations in section 3 were applied to a real hyperspectral scene collected by an AVIRIS flight over the Cuprite mining district in Nevada, and consists of 614x512 pixels and 224 bands. The site is well understood mineralogically, and has several exposed minerals of interest. Since the data are available online from http://aviris.jpl.nasa.gov/html/aviris.freedata.html, people interested in the proposed parallel algorithms can reproduce our results. An experiment-based cross-examination of endmember extraction accuracy was first conducted to assess the spectral similarity scores obtained after comparing five ground-truth library spectra collected on the field with the corresponding endmembers extracted by the three parallel implementations of the PPI algorithm. This experiment revealed that all the considered parallel implementations produced exactly the same results as the original PPI implemented in Kodak's Research Systems ENVI 4.0, with spectral similarity scores to ground-truth which were very satisfactory in all cases. It is worth noting that the PPI produced the same final set of experiments when the number of randomly generated skewers was set to $k = 10^4$ or above (values of $k = 10^3$, 10^5 and 10^6 were also tested). Based on the above simple experiments, we empirically set the cutoff threshold parameter t to the mean of N_{PPI} scores obtained after $k = 1000$ iterations. These values are in agreement with those used before [11].

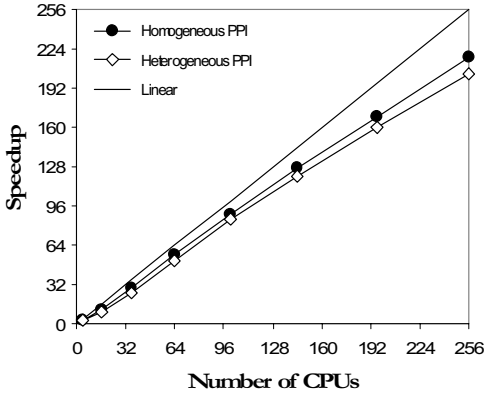

Figure 3. Scalability of the homogeneous and heterogeneous PPI implementation on Thunderhead.

To investigate the parallel properties of the parallel algorithms described in sections 3.1 and 3.2, we first tested the performance of the heterogeneous PPI algorithm in section 3.2 and that of its homogeneous version (see section 3.1) on NASA's GSFC Thunderhead Beowulf cluster. Fig. 3 plots the speedups achieved by multi-processor runs of the homogeneous and heterogeneous parallel version of the PPI algorithm over the corresponding single-processor runs of each considered algorithm on Thunderhead. As Fig. 3 shows, the scalability of the heterogeneous algorithm was esentially

the same as that evidenced by its homogeneous version. For illustrative purposes, Table 3 also reports the measured execution times by the tested algorithms on Thunderhead, using different numbers of processors. Results reveal that the heterogeneous implementation of PPI can effectively adapt to a massively parallel homogeneous environment and to produce a response in a few seconds using a moderate number of processors.

Number of CPUs	Homogeneous PPI	Heterogeneous PPI
1	2745	2745
4	1012	1072
16	228	273
36	94	106
64	49	53
100	30	32
144	21	22
196	16	17
256	12	13

Table 3. Execution times (seconds) for the homogeneous and heterogeneous PPI implementations on Thunderhead.

In order to see how the homogeneous and heterogeneous implementations perform on a fully heterogeneous network of workstations, we also tested their performance by timing the parallel programs using the heterogeneous network of workstations and its equivalent distributed homogeneous network. Table 4 shows the execution time of the parallel algorithms on the homogeneous and heterogeneous networks. For the sake of comparison, the table also shows the speedup of the heterogeneous algorithm over its respective homogeneous version on the same heterogeneous platform, where the speedup was simply calculated as the execution time of the homogeneous algorithm divided by the execution time of the heterogeneous algorithm [18]. Similarly, Table 4 shows the speedup of the homogeneous algorithm over its respective heterogeneous version on the same homogeneous platform, with the speedup calculated as the execution time of the heterogeneous algorithm divided by the execution time of the homogeneous one.

	Heterogeneous		Homogeneous	
Algorithm	Time	Speedup	Time	Speedup
Heterogeneous PPI	295	6.66	288	-
Homogeneous PPI	1967	-	271	1.07

Table 4. Execution times (seconds) and speedups for the parallel implementations in the heterogeneous and homogeneous network.

As expected, the heterogeneous algorithm was able to adapt much better to the heterogeneous environment than the homogeneous algorithm. Table 4 also reveals that the performance of the heterogeneous algorithm was almost the same as that evidenced by the homogeneous one when they were run in the same, homogeneous environment. This confirms an introspection which was already observed in experiments using the Beowulf cluster: that the heterogeneous PPI was able to adapt efficiently to both homogeneous and heterogeneous environments.

In order to measure load balance, Table 5 shows the imbalance scores achieved by the considered algorithms on both the heterogeneous and homogeneous network. The imbalance is defined as $D = R_{max} / R_{min}$, where R_{max} and R_{min} are the maxima and minima processor run times, respectively. Therefore, perfect balance is achieved when $D = 1$. In the table, we display the imbalance considering all processors, D_{All}, and also considering all processors but the root, D_{Minus}. In all cases, load balance was better when the root processor was not included, which is mainly due to sequential computations at the root node (e.g., the final endmember selection step). However, it is clear from Table 5 that the homogeneous algorithm executed on the heterogeneous network provided the highest values of D_{All} and D_{Minus} (and hence the highest imbalance), while the heterogeneous algorithm always resulted in values of D_{All} (and, in particular, of D_{Minus}) which were close to 1, regardless of the platform where it was run.

	Heterogeneous		Homogeneous	
Algorithm	D_{All}	D_{Minus}	D_{All}	D_{Minus}
Heterogeneous PPI	1.12	1.03	1.15	1.05
Homogeneous PPI	1.86	1.27	1.13	1.02

Table 5. Load-balancing rates in the heterogeneous and homogeneous network.

Although the idea of mounting clusters and networks of processing elements aboard airborne and satellite hyperspectral imaging facilities has been explored in the past, the number of processing elements in such experiments has been very limited thus far, due to payload requirements in most remote sensing missions. For instance, a low-cost, portable Myrinet cluster of 16 processors (with similar specifications as those of our homogeneous network of workstations above) was recently developed at NASA's GSFC with cost of 3000$. Unfortunately, it could still not facilitate real-time performance. The cost of a Xilinx Virtex-II XC2V6000-6 FPGA used for experiments in this work is currently only slightly higher than that of the portable Myrinet cluster mentioned above.

In order to calibrate the usefulness of our hardware-based implementation, Table 6 shows a summary of resource utilization by our systolic array-based implementation of the PPI algorithm on the considered Xilinx FPGA, which was able to provide a response in eight seconds. Since the FPGA used in experiments has a total of 33,792 slices available, the results addressed in Table 6 indicate that there is still room in the FPGA for

implementation of additional algorithms. Further experiments are still required, however, in order to optimize our FPGA-based design to be able to process full AVIRIS data sets in near real-time.

Number of gates	Number of slices	Percentage of slices used	Maximum operation frequency (MHz)
526944	12418	36%	18.032

Table 6. Summary of resource utilization for the FPGA implementation of the PPI algorithm.

5. Conclusions and Future Work

This paper has examined different HPC-based strategies for remotely sensed hyperspectral imaging. Through the detailed analysis of the PPI, a well-known hyperspectral analysis method available in commercial software, we have explored different techniques to increase computational performance of the algorithm (which can take up to several hours of computation to complete its calculations in last-generation desktop computers). Two of the considered strategies, i.e., commodity cluster-based computing and distributed computing in heterogeneous networks of workstations, seem particularly appropriate for information extraction from very large hyperspectral data archives. The scalability, code reusability and load balance achieved by the proposed implementations in such low-cost systems offers an unprecedented opportunity to explore methodologies in other fields (e.g., data mining) that previously looked to be too computationally intensive for practical applications. To address the (near) real-time computational needs introduced by many remote sensing applications, we have also developed a systolic array-based FPGA implementation of the PPI. Experimental results demonstrate that our hardware version of the PPI makes an appropriate use of computing resources in the FPGA, and further provides a response in near real-time which is believed to be acceptable in most remote sensing applications. Further, the reconfigurability of FPGA systems opens many innovative perspectives from an application point of view, including the appealing possibility of being able to adaptively select one out of a pool of available hyperspectral data processing algorithms (which could be applied on the fly aboard the airborne/satellite platform, or even from a control station on Earth). Although the experimental results presented in this paper are very encouraging, further work is still needed to arrive to optimal parallel design and implementations for the PPI and other algorithms.

References

[1] C.-I Chang, *Hyperspectral imaging: Techniques for spectral detection and classification*, Kluwer: NY, 2003.

[2] R. O. Green et al., "Imaging spectroscopy and the airborne visible/infrared imaging spectrometer AVIRIS," *Remote Sens. Environment*, vol. 65, pp. 227–248, 1998.

[3] L. Chen, I. Fujishiro, K. Nakajima, "Optimizing parallel performance of unstructured volume rendering for the Earth Simulator," *Parallel Computing*, vol. 29, pp. 355–371, 2003.

[4] G. Aloisio and M. Cafaro, "A dynamic earth observation system," *Parallel Computing*, vol. 29, pp. 1357–1362, 2003.

[5] P. Wang, K. Y. Liu, T. Cwik and R.O. Green, "MODTRAN on supercomputers and parallel computers," *Parallel Computing*, vol. 28, pp. 53–64, 2002.

[6] T. Achalakul and S. Taylor, "A distributed spectral-screening PCT algorithm," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 373-384, 2003.

[7] A. Plaza, D. Valencia, J. Plaza and P. Martínez, "Commodity cluster-based parallel processing of hyperspectral imagery," *Journal of Parallel and Distributed Computing*, to appear in 2006.

[8] J.W. Boardman, F.A. Kruse and R.O. Green, "Mapping target signatures via partial unmixing of AVIRIS data," in: *JPL Earth Science Workshop*, Pasadena, CA, 1995.

[9] Research Systems, Inc., *ENVI User's Guide*. Boulder, CO: Research Systems, Inc., 2001.

[10] N. Keshava and J.F. Mustard, "Spectral unmixing," *IEEE Signal Processing Magazine*, vol. 19, pp. 44–57, 2002.

[11] A. Plaza, P. Martínez, R. Pérez and J. Plaza, "A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data," *IEEE Trans. Geosci. Remote Sensing*, vol. 42, pp. 650–663, 2004.

[12] A. Plaza, P. Martinez, R.M. Perez and J. Plaza, "Spatial/spectral endmember extraction by multidimensional morphology," *IEEE Trans. Geosci. Remote Sensing*,, vol. 40, pp. 2025–2041, 2002.

[13] D. Valencia, A. Plaza, P. Martínez and J. Plaza, "On the use of cluster computing architectures for implementation of hyperspectral analysis algorithms," in: *Proceedings of 10th IEEE Symp. Computers and Communications*, Cartagena, Spain, pp. 995–1000, 2005.

[14] M. Valero-García, J. J. Navarro, J. Llabería, M. Valero and T. Lang, "A method for implementation of one-dimensional systolic algorithms with data contraflow using pipelined functional units," *Journal of VLSI Signal Processing*, vol. 4, pp. 7-25, 1992.

[15] Celoxica Ltd., *Handel-C Reference Manual*, 2003.

[16] Celoxica Ltd., *DK Design Suite User Manual*, 2003.

[17] J. Dorband, J. Palencia and U. Ranawake, "Commodity computing clusters at Goddard Space Flight Center," *Journal of Space Communication*, vol. 1, 2003.

[18] A. Lastovetsky and R. Reddy, "On performance analysis of heterogeneous parallel algorithms," *Parallel Computing*, vol. 30, pp. 1195–1216, 2004.