21

Toward a High-Performance Distributed CBIR System for Hyperspectral Remote Sensing Data: A Case Study in Jungle Computing

Timo van Kessel
VU University, Amsterdam, The Netherlands

Niels Drost and Jason Maassen
Netherlands eScience Center, Amsterdam, The Netherlands

Henri E. Bal
VU University, Amsterdam, The Netherlands

Frank J. Seinstra
Netherlands eScience Center, Amsterdam, The Netherlands

Antonio J. Plaza
University of Extremadura, Caceres, Spain

High-Performance Computing on Complex Environments, First Edition.
Edited by Emmanuel Jeannot and Julius Žilinskas.

Despite the immense performance of modern computing systems, many scientific problems are of such complexity and scale that a wide variety of computing hardware must be employed concurrently. Moreover, as the required hardware often is not available as part of a single system, scientists often are forced to exploit the computing power of a very diverse distributed system. As many different types of hardware are available nowadays, this results in a complex hierarchical collection of heterogeneous computing hardware, which we refer to as a *jungle computing system*. Using such a collection of hardware all at once (let alone efficiently) is a difficult problem.

An example application domain that brings forth very diverse hardware requirements is that of content-based image retrieval (CBIR) for hyperspectral remote sensing data. Whereas algorithms to process remote sensing data are often already computationally complex, instruments for earth observation generate immense amounts of (distributed) data. As a result, to process this data within reasonable time, there is a need to exploit as much available computing power as possible.

In this chapter, we present a prototype for a CBIR system for remotely sensed image data. We focus on hyperspectral data, a type of remote sensing data characterized by very high resolution in the spectral domain. We show that our prototype system can easily be adapted to match the configuration of a jungle computing system, and is able to process and search the contents of several repositories efficiently.

21.1 INTRODUCTION

Many scientific problems are of such complexity and scale that solutions are obtained only using a wide variety of computing hardware—all at once. To effectively exploit the available processing power, a thorough understanding of the complexity of such systems is essential.

Despite the fact that there is an obvious need for programming solutions that allow scientists to obtain high performance and distributed computing both efficiently and transparently, real solutions are still lacking [1, 2]. Worse even, the high-performance and distributed computing landscape is currently undergoing a revolutionary change. Traditional clusters, grids, and cloud systems are more and more equipped with state-of-the-art many-core technologies (e.g., graphics processing units or GPUs [3, 4]). Although these devices often provide orders-of-magnitude speed improvements, they make computing platforms more heterogeneous and hierarchical—and vastly more complex to program and use.

Further complexities arise in everyday practice. Given the ever-increasing need for computing power, and because of additional issues including data distribution, software heterogeneity, and so on, scientists are commonly forced to apply multiple clusters, grids, clouds, and other systems concurrently—even for single applications. In this chapter, we refer to such a simultaneous combination of heterogeneous, hierarchical, and distributed computing resources as a *jungle computing system* [5].

Constellation [6] is a lightweight software platform, specifically designed to implement applications able to run on jungle computing systems. Constellation aims to efficiently run applications on complex combinations of distributed,

heterogeneous, and hierarchical computing hardware. In addition, Constellation makes retargeting applications to completely different computing environments straightforward.

Constellation assumes that applications consist of multiple distinct activities with certain dependencies between them. These activities can be implemented independently using the tools, and targeted at the architecture, that suit them best. Multiple implementations of an activity may be created to support different hardware architectures (e.g., in C, C++/MPI, CUDA) or problem instances. Existing external codes can also be used. This approach to application development vastly reduces the programming complexity. Instead of having to create a single application capable of running in a complex, distributed, and heterogeneous environment, it is sufficient to create (or reuse) several independent activities targeted at smaller and simpler homogeneous environments. Traditional high-performance computing (HPC) tools and libraries such as MPI [7] or CUDA[1] can be used to create each of the separate activities.

Remote sensing of the Earth is a research field in which problems are of such a complexity that the use of jungle computing systems becomes indispensable.[2] For instance, the NASA Jet Propulsion Laboratory's Airborne Visible Infrared Imaging Spectrometer (AVIRIS) [10] is able to record the visible and near-infrared spectrum of the reflected light of an area several kilometers long (depending on the duration of the flight) using hundreds of spectral bands. The resulting "image cube" is a stack of images (Fig. 21.1), in which each pixel (vector) has an associated spectral signature or "fingerprint" that uniquely characterizes the underlying objects. The resulting data often comprise several gigabytes per flight.

The amount of data generated by such instruments tends to grow as new instruments are deployed. This is caused not only by an increasing number of instruments that are operational but also by the fact that modern instruments are able to produce much more complex images by increasing both the image resolution and the number of spectral bands that are captured. In addition, more advanced instruments are being developed that will produce even more data. At present, the data generated by hyperspectral instruments is stored at several locations. To make things worse, some of these data might be replicated over multiple repositories. In short, analyzing all data generated by modern and future instruments is a hard problem.

To analyze all these data, there is a need for a system that enables researchers to search in all these repositories in a sensible way. For this purpose, content-based image retrieval (CBIR) intends to retrieve, from real data stored in distributed databases, information that is relevant to a query [11]. This is particularly important in large data repositories, such as those available in remotely sensed hyperspectral imaging [12]. In [8], we described how spectral information can be extracted out of hyperspectral images to create a CBIR system suitable for the remote sensing community. In this system, spectral endmembers are extracted from an image that is used as a *query image*, and are compared with the spectral endmembers of the

[1] http://www.nvidia.com/cuda.html.

[2] Many other of such domains exists, such as computational astrophysics [9], climate modeling, etc.

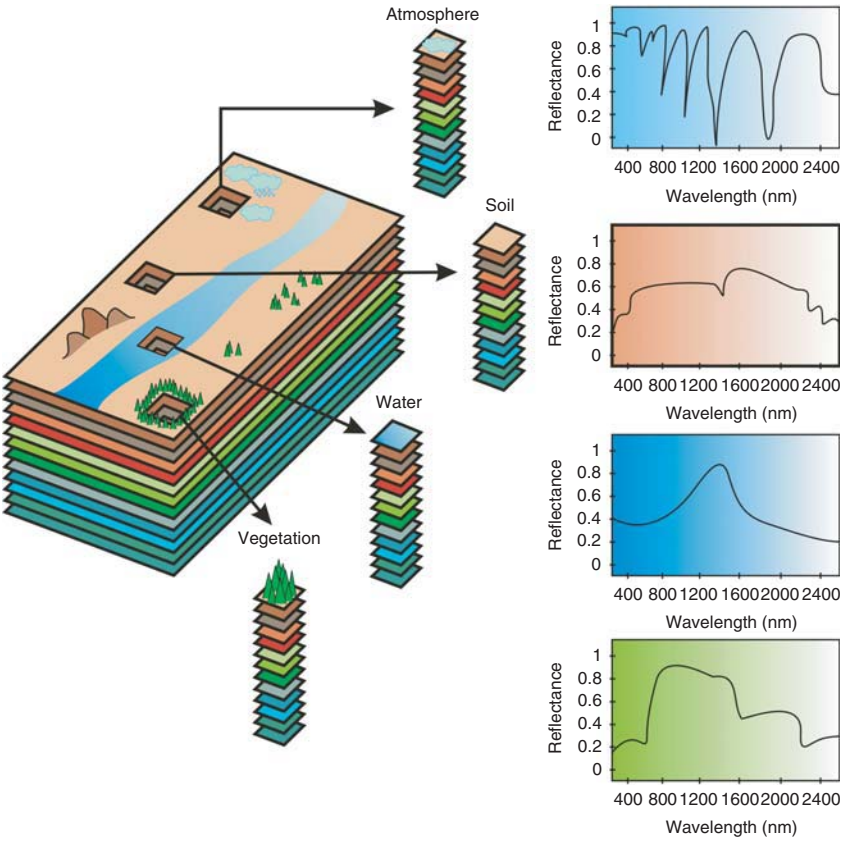

FIGURE 21.1 *Concept of hyperspectral imaging [8]. Copyright © 2009 John Wiley & Sons, Ltd. Reprinted by permission of John Wiley & Sons, Inc.*

hyperspectral images in a database to determine a similarity score. The most similar images are then returned as the result of the query.

In this chapter, we will discuss how the Constellation programming system can be used to design a flexible system for CBIR of hyperspectral remote sensing data for jungle computing systems. We argue that the system is easily adaptable to the specific circumstances of the jungle environment. Finally, we show that a prototype implementation of our system offers good performance in a real-world jungle environment.

This chapter is organized as follows: In Section 21.2, we introduce the remote sensing problem and discuss the hyperspectral imaging techniques that are required for hyperspectral image retrieval. Section 21.3 introduces the jungle computing paradigm. Section 21.4 introduces Ibis and Constellation, the software platform that enables the creation of applications for jungle computing systems. Section 21.5 describes our design of a prototype CBIR system for hyperspectral remote sensing

data using the Constellation platform. An evaluation of the prototype system is given in Section 21.6, followed by our conclusions in Section 21.7.

21.2 CBIR FOR HYPERSPECTRAL IMAGING DATA

One of the main problems involved in hyperspectral data exploitation is spectral unmixing [13], as many of the pixels collected by imaging spectrometers such as AVIRIS are highly mixed in nature due to spatial resolution and other phenomena. For instance, it is very likely that the pixel labeled as "vegetation" in Fig. 21.1 is actually composed of several types of vegetation canopies interacting at subpixel levels. Similarly, the "soil" pixel may comprise different types of geological features. As a result, spectral unmixing is a very important task for hyperspectral data exploitation, since the spectral signatures collected in natural environments are invariably a mixture of the pure signatures of the various materials found within the spatial extent of the ground instantaneous field view of the imaging instrument. Among several techniques designed to deal with the inherent complexity of hyperspectral images in a supervised manner [13, 14], linear spectral unmixing follows an unsupervised approach which aims at inferring pure spectral signatures, called *endmembers*, and their material fractions, called *abundances*, at each pixel of the scene.

21.2.1 Spectral Unmixing

Let us assume that a remotely sensed hyperspectral image with n bands is denoted by $\mathbf{I}$, in which a pixel of the scene is represented by a vector $\mathbf{x} = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^N$, where $\mathbb{R}$ denotes the set of real numbers in which the pixel's spectral response x_k at sensor channels $k = 1, \ldots, n$ is included. Under the linear mixture model assumption [15, 16], each pixel vector in the original scene can be modeled using the following expression:

$$\mathbf{x} \approx \mathbf{E}\mathbf{a} + \mathbf{n} = \sum_{i=1}^{p} \mathbf{e}_i a_i + \mathbf{n}, \tag{21.1}$$

where $\mathbf{E} = \{\mathbf{e}_i\}_{i=1}^{p}$ is a matrix containing p pure spectral signatures (endmembers), $\mathbf{a} = [a_1, a_2, \ldots, a_p]$ is a p-dimensional vector containing the abundance fractions for each of the p endmembers in $\mathbf{x}$, and $\mathbf{n}$ is a noise term. Solving the linear mixture model involves (i) identifying a collection of $\{\mathbf{e}_i\}_{i=1}^{p}$ endmembers in the image, and (ii) estimating their abundance in each pixel $\mathbf{x}$ of the scene. These processing steps are described in the following sections.

21.2.1.1 Endmember Extraction First, a set of $\mathbf{E} = \{\mathbf{e}_i\}_{i=1}^{p}$ endmember signatures are extracted from the input dataset. For this purpose, we consider the N-FINDR algorithm [17], which is one of the most widely used and successfully applied methods for automatically determining endmembers in hyperspectral image data without using *a priori* information. This algorithm looks for the set of pixels with the largest possible volume by *inflating* a simplex inside the data. The procedure begins with a

random initial selection of pixels. Every pixel in the image must be evaluated in order to refine the estimate of endmembers, looking for the set of pixels that maximizes the volume of the simplex defined by selected endmembers. The corresponding volume is calculated for every pixel in each endmember position by replacing that endmember and finding the resulting volume. If the replacement results in a an increase of volume, the pixel replaces the endmember. This procedure is repeated in iterative manner until there are no more endmember replacements. The method can be summarized by a step-by-step algorithmic description which is given as follows:

1. *Feature Reduction*: Apply a dimensionality reduction transformation such as the principal component analysis (PCA) [18] to reduce the dimensionality of the data from n to $p-1$, where p is an input parameter to the algorithm (number of endmembers to be extracted).
2. *Initialization*: Let $\{\mathbf{e}_1^{(0)}, \mathbf{e}_2^{(0)}, \ldots, \mathbf{e}_p^{(0)}\}$ be a set of endmembers randomly extracted from the input data.
3. *Volume Calculation*: At iteration $k \geq 0$, calculate the volume defined by the current set of endmembers as follows:

$$V(\mathbf{e}_1^{(k)}, \mathbf{e}_2^{(k)}, \ldots, \mathbf{e}_p^{(k)}) = \frac{\left|\det \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \mathbf{e}_1^{(k)} & \mathbf{e}_2^{(k)} & \cdots & \mathbf{e}_p^{(k)} \end{bmatrix}\right|}{(p-1)!}. \tag{21.2}$$

4. *Replacement*: For each pixel vector $\mathbf{x}$ in the input hyperspectral data, recalculate the volume by testing the pixel in all p endmember positions, that is, first calculate $V(\mathbf{x}, \mathbf{e}_2^{(k)}, \ldots, \mathbf{e}_p^{(k)})$, then calculate $V(\mathbf{e}_1^{(k)}, \mathbf{x}, \ldots, \mathbf{e}_p^{(k)})$, and so on, until $V(\mathbf{e}_1^{(k)}, \mathbf{e}_2^{(k)}, \ldots, \mathbf{x})$. If none of the p recalculated volumes is greater than $V(\mathbf{e}_1^{(k)}, \mathbf{e}_2^{(k)}, \ldots, \mathbf{e}_p^{(k)})$, then no endmember is replaced. Otherwise, the combination with maximum volume is retained. Let us assume that the endmember absent in the combination resulting in the maximum volume is denoted by $\mathbf{e}_i^{(k+1)}$. In this case, a new set of endmembers is produced by letting $\mathbf{e}_i^{(k+1)} = \mathbf{x}$ and $\mathbf{e}_l^{(k+1)} = \mathbf{e}_l^{(k)}$ for $l \neq i$. The replacement step is repeated for all the pixel vectors in the input data until all the pixels have been exhausted.

21.2.1.2 Abundance Estimation Once the set of endmembers $\mathbf{E} = \{\mathbf{e}_i\}_{i=1}^p$ has been extracted, an unconstrained solution to (21.1) is simply given by the following expression [12]:

$$\mathbf{a} \approx (\mathbf{E}^T\mathbf{E})^{-1}\mathbf{E}^T\mathbf{x}. \tag{21.3}$$

However, two physical constraints are generally imposed in order to estimate the p-dimensional vector of abundance fractions $\mathbf{a} = [a_1, a_2, \ldots, a_p]$ at a given pixel $\mathbf{x}$, these are the abundance nonnegativity constraint (ANC), that is, $a_i \geq 0$ for all $1 \leq i \leq p$, and the abundance sum-to-one constraint (ASC), that is, $\sum_{i=1}^p a_i = 1$ [16]. As indicated in [16], a fully constrained (i.e., ASC-constrained and ANC-constrained) estimate can be obtained in least-squares sense simultaneously.

Such fully constrained linear spectral unmixing estimate is generally referred to in the literature by the acronym FCLSU [16].

21.2.2 Proposed CBIR System

The proposed CBIR system for retrieval of hyperspectral imagery is based on the spectral unmixing methodology described in the earlier section. In this section, we describe the stages involved in a standard search procedure using the proposed CBIR system from an user's point of view as follows:

1. *Input Query*: The user first selects a portion or a full hyperspectral scene to be used as an input image $\mathbf{I}$. Then, the system computes a feature vector associated to that portion given by the spectral signatures of endmembers $\mathbf{E} = \{\mathbf{e}_i\}_{i=1}^{p}$, derived using the N-FINDR algorithm, and their correspondent FCLSU-derived abundances summed across all the pixels in the considered portion. The number of endmembers to be extracted from the sample portion is automatically calculated using the hyperspectral signal identification by minimum error (HySime) method [19].
2. *Signature Comparison and Sorting*: The feature vector obtained in the previous stage, which comprises the p endmembers and their macroscopic abundances in the selected image portion $\mathbf{I}$, is compared with the precomputed feature vectors of all the hyperspectral images in the database, using the following spectral signature matching algorithm (SSMA). Let $\{\mathbf{e}_i\}_{i=1}^{p}$ be a set of p endmembers extracted from the considered image $\mathbf{I}$, and let $\{\mathbf{r}_j\}_{j=1}^{q}$ be a set of q endmembers extracted from a reference hyperspectral image $\mathbf{R}$ already available in the database (we assume that the database contains a large number of hyperspectral images). The idea now is to analyze if $\mathbf{R}$ should be retrieved as a hyperspectral image that is "sufficiently similar" with regard to $\mathbf{I}$. It should be noted that the endmembers and their corresponding abundances are stored for each image dataset catalogued in the system, hence we use this information in order to decide about the similarity of the compared images. To accomplish this task, we use a spectral angle distance (SAD) based [15] similarity criterion which is implemented using the following steps:
   (a) *Initial Labeling*: Label all endmembers in the test set $\{\mathbf{e}_i\}_{i=1}^{p}$ as *unmatched*.
   (b) *Matching*: For each unmatched endmember in the test set $\{\mathbf{e}_i\}_{i=1}^{p}$, calculate the spectral angle between the test endmember and all endmembers in the reference set $\{\mathbf{r}_j\}_{j=1}^{q}$. If the pair $(\mathbf{e}_k, \mathbf{r}_l)$, with $1 \leq k \leq p$ and $1 \leq l \leq q$, results in the minimum obtained value of $\mathrm{SAD}(\mathbf{e}_k, \mathbf{r}_l)$, and the value is below threshold angle v_a, then label the endmembers $\mathbf{e}_k$ and $\mathbf{r}_l$ as "matched."
   (c) *Relative Difference Calculation*: For each matched endmember $\mathbf{e}_k$ resulting from the previous step, the identifiers of the M images which contain such endmember are extracted and ranked in descending order of spectral similarity.
3. *Display Results*: A mosaic made up of the first M images selected is assembled and then presented to the user as the search result.

21.3 JUNGLE COMPUTING

When the notion of grid computing was introduced over a decade ago; its foremost visionary aim (or "promise") was to provide *efficient and transparent (i.e., easy-to-use) wall-socket computing over a distributed set of resources* [20]. Since then, many other distributed computing paradigms have been introduced, including peer-to-peer computing [21], volunteer computing,[3] and—more recently—cloud computing [22]. These paradigms all share many of the goals of grid computing, eventually aiming to provide end users with access to distributed resources (ultimately even at a worldwide scale) with as little effort as possible.

These new distributed computing paradigms have led to a diverse collection of resources available to research scientists, including stand-alone machines, cluster systems, grids, clouds, desktop grids, and so on. Extreme cases in terms of computational power further include mobile devices at the low end of the spectrum and supercomputers at the top end.

If we take a step back, and look at such systems from a high-level perspective, then all of these systems share important common characteristics. Essentially, *all* of these systems consist of a number of basic computing nodes, each having local memories, and each capable of communicating over a local or wide-area connection. The most prominent differences are in the semantic and administrative organization, with many systems providing their own middleware, programming interfaces, access policies, and protection mechanisms [23].

Apart from the increasing diversity in the distributed computing landscape, the "basic computing nodes" mentioned above currently are undergoing revolutionary change as well. General-purpose CPUs at present have multiple computing cores per chip, with an expected increase in the years to come [24]. Moreover, special-purpose chips (e.g., GPUs) are now combined or even integrated with CPUs to increase performance by orders of magnitude [25].

With clusters, grids, and clouds thus being equipped with multicore processors and many-core "accelerators," systems available to scientists are becoming increasingly hard to program and use. Despite the fact that the programming and efficient use of many-core devices is known to be hard, this is not the only—or most severe—problem. With the increasing heterogeneity of the underlying hardware, the efficient mapping of computational problems onto the "bare metal" has become vastly more complex. Now more than ever, programmers must be aware of the potential for parallelism *at all levels of granularity*.

But the problem is even more severe. Given the ever-increasing desire for speed and scalability in many scientific research domains, the use of a single high-performance computing platform is often not sufficient. In part, the need to access multiple platforms concurrently from within a single application often is due to the impossibility of reserving a sufficient number of computing nodes at once in a single multiuser system. Moreover, additional issues such as the distributed nature of the input data, heterogeneity of the software pipeline being applied, and

[3] e.g., SETI@home, see also http://setiathome.ssl.berkeley.edu.

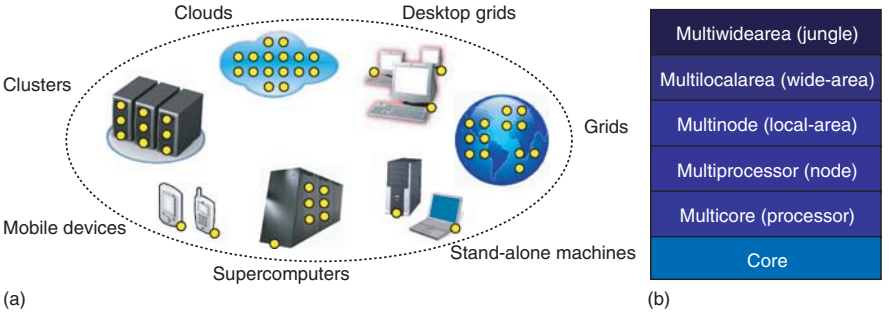

FIGURE 21.2 *Jungle computing system. (a) A "worst-case" jungle computing system as perceived by scientific end users, simultaneously comprising any number of clusters, grids, clouds, and other computing platforms. (b) Hierarchical view of a jungle computing system. [5]. © Springer-Verlag London Limited 2011. Reprinted with kind permission from Springer Science+Business Media B.V.*

ad hoc availability of the required computing resources further indicate a need for computing across multiple, and potentially very diverse, platforms. For all of these reasons, many research scientists (now and in the near future) are being forced to apply multiple clusters, grids, clouds, and other systems *concurrently*—even for executing single applications. We refer to such a simultaneous combination of heterogeneous, hierarchical, and distributed computing resources as a *jungle computing system* (Fig. 21.2).

Although the notion of jungle computing systems exposes *all* computing problems that scientists today can be (and often are) confronted with, we do not expect most (or even any) research scientists to have to deal with the "worst case" scenario depicted in Fig. 21.2. We do claim, however, that—in principle—*any* possible subset of this figure represents a realistic scenario. Hence, if we can define the fundamental methodologies required to solve the problems encountered in the worst case scenario, we ensure that our solution applies to all possible scenarios.

21.3.1 Jungle Computing: Requirements

Although jungle computing systems and grids are not identical (i.e., the latter being constituent components of the former), the generic aims of jungle computing are already defined by the "founding fathers of the grid." Foster *et al.* [20] indicate that one of the main aims of grid computing is to deliver *transparent* and potentially *efficient* computing, even at a world-wide scale. This aim extends to jungle computing as well.

It is well known that adhering to the general requirements of transparency and efficiency is a hard problem. Although rewarding approaches exist for specific application types (i.e., work-flow-driven problems [2, 26] and parameter sweeps [27]), solutions for more general applications types (e.g., involving irregular communication patterns) do not exist today. This is unfortunate, as advances in optical networking allow for a much larger class of distributed (jungle computing) applications to run efficiently [28].

We ascribe this rather limited use of grids and other distributed systems—or the lack of efficient and transparent programming models—to the intrinsic complexities of distributed (jungle) computing systems. Programmers often are required to use low-level programming interfaces that change frequently. Also, they must deal with system and software heterogeneity, connectivity problems, and resource failures. Furthermore, managing a running application is hard, because the execution environment may change dynamically as resources come and go. These problems limit the acceptance of the many distributed computing technologies available today.

The above-mentioned general requirements of transparency and efficiency are unequal quantities. The requirement of transparency decides whether an end user is capable of using a jungle computing system at all, while the requirement of efficiency decides whether the use is sufficiently satisfactory. In the following, we will therefore focus mainly on the transparency requirements. We will simply assume that—once the requirement of transparency is fulfilled—efficiency is a derived property that can be obtained amongst others by introducing "intelligent" optimization techniques, application domain-specific knowledge, and so on [5].

21.4 IBIS AND CONSTELLATION

The Ibis project [23] (see also www.cs.vu.nl/ibis/) aims to simplify the programming and deployment process of applications for jungle computing systems. The Ibis philosophy is that such applications should be developed on a local workstation and simply be launched from there. This write-and-go philosophy requires minimalistic assumptions about the execution environment, and sends most of the environment's software (e.g., libraries) along with the application. To this end, Ibis exploits Java virtual machine technology, and uses middleware-independent application programming interfaces that are automatically mapped onto the available middleware.

Ibis provides different programming abstractions, ranging from low-level message passing to high-level divide-and-conquer parallelism (Fig. 21.3). All programming abstractions are implemented on the same Java library, called the *Ibis portability layer* (IPL). Ibis also includes a deployment system, based on the JavaGAT (which provides file I/O, job submission, job monitoring, etc., in a middleware-independent manner) and on the Zorilla peer-to-peer system. The Ibis system is designed to run in a jungle computing environment that is dynamic and heterogeneous and suffers from connectivity problems. For example, Ibis solves connectivity problems automatically using the "SmartSockets" socket library.

A recent addition to the Ibis project is *Constellation*: a lightweight platform that is specifically designed for distributed, heterogeneous, and hierarchical computing environments [6]. Similar to most of the software developed in the Ibis project, Constellation itself is implemented in Java, which provides both portability and acceptable performance. Currently, we assume that a Constellation application (i.e., the glue code connecting the activities) is also a Java application. Although a Java class is used to represent an activity, the code performing the actual processing may be implemented using scripts, C, CUDA, MPI, and so on. In this section, we will give an overview of the programming model offered by Constellation.

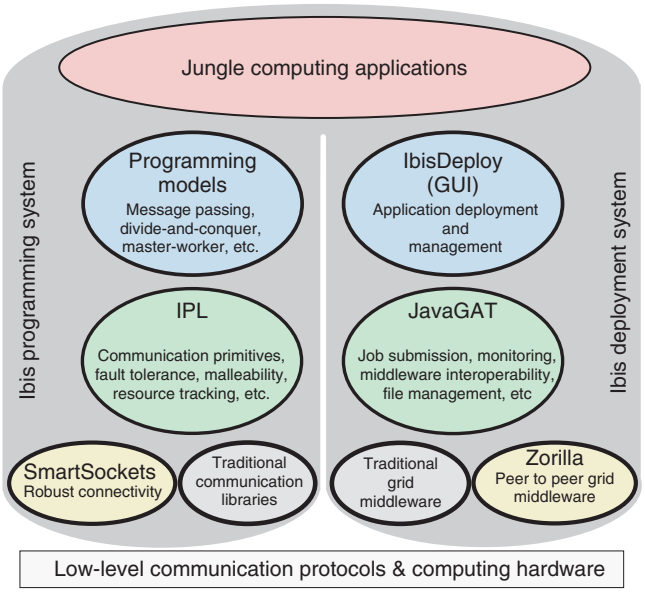

FIGURE 21.3 *Overview of the Ibis software stack and its components [5]. © Springer-Verlag London Limited 2011. Reprinted with kind permission from Springer Science+Business Media B.V.*

In Constellation, a program consists of a set of loosely coupled *activities* that communicate using *events*. The complexity of a program may vary from a simple bag of tasks to a complex workflow comprised of multiple interdependent activities. Figure 21.4 (top) shows a simple example using four activities. There, an initial activity (white) is started that submits three additional activities (gray and black).

Each activity represents a distinct action that needs to be performed by the application, for example, process a piece of data or run a simulation. As such, an activity usually represents a combination of code, parameters, and data. Each activity may consist of a script, sequential C, CUDA, a parallel application using OpenMP or MPI, and so on.

Constellation uses *executors* to represent hardware capable of running activities. An executor may represent a single core of a machine, a single machine with multiple cores, a specialized piece of hardware (e.g., a GPU), an entire cluster, and so on. The application is free to determine how the executors should represent the hardware.

Constellation assumes that a *glide-in* [29–31] approach is used to start the executors on the available hardware, for example, using IbisDeploy [23] or Zorilla [32]. In this chapter, we will not describe how the necessary resources are obtained. Instead, we assume that a heterogeneous set of resources capable of running the necessary executors is available.

In Fig. 21.4 (bottom), three executors are shown, one representing a multicore machine, one representing a machine containing a GPU, and one representing a small cluster of eight machines. Obviously, when such a heterogeneous set of executors is

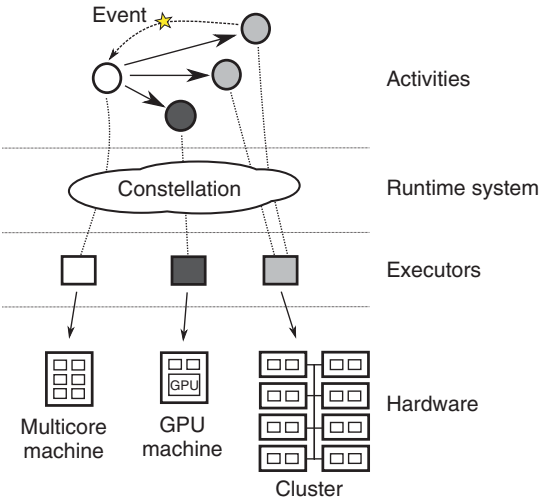

FIGURE 21.4 *Example of a Constellation application [6]. © 2011 Association for Computing Machinery, Inc. Reprinted by permission. http://doi.acm.org/10.1145/1996010.1996013.*

used, not every activity will be able to run on every executor. An activity consisting of GPU code will not run on an MPI cluster, nor will an MPI application run on a GPU. Therefore, it is essential for running the application that the activities end up on a suitable executor. For this purpose, Constellation uses the concept of *context*.

A context is an application defined label (or set of labels) that can be attached to both activities and executors. A label can describe data dependencies ("`dataset X`"), hardware capabilities ("`GPU`"), or problem size ("`large`"). When an activity and executor use the same label, they *match*; that is, the executor is assumed to offer the right context for running the activity. Constellation plays the role of matchmaker to ensure that each activity is forwarded to a suitable executor. In Fig. 21.4, this is illustrated by the different shades used for the activities and executors.

Although the concept of context matching is similar to the resource requirement matching used in traditional resource managers [29, 33], there is an important difference. Resource managers often use a predefined list of attributes describing both the hardware and software (e.g., libraries) that are available on a machine. A feature can only be used for matching if it was included in the list to begin with. In addition, complex combinations of attributes often are needed to describe a task's requirements, thereby making the matching procedure complicated and expensive.

In contrast, Constellation uses simple *application-defined* labels to describe what context an activity requires and an executor has to offer. Using application-specific knowledge, the distinctions between the different activities and executors are often easy to make. For example, for many applications, simple labels such as "`GPU`" or "`dataset X`" are sufficient to classify the executors. This makes context matching simple and fast, allowing fine-grained applications to be scheduled efficiently.

Events can be used for communication with activities. When an activity is created, it is assigned a globally unique ID. When sending an event to an activity, this ID can be used as a destination address. Events are mainly used for signaling between activities, for example, to indicate that certain data is available or that certain processing steps have finished. However, if necessary, they can also be used to transfer data between activities.

The complete set of activities does not have to be known in advance. During the application's lifetime, new activities may be submitted to the Constellation run time system (RTS), either by some external application or spawned dynamically by activities that are already running. Newly created activities are activated on a suitable executor to perform processing. When finished, the activity may decide to either terminate or suspend and wait for events. Whenever an activity receives an event, it is reactivated to allow it to process the event and perform additional processing, if necessary. When finished, the activity must again decide to terminate or suspend.

It is the task of the Constellation RTS to ensure that all activities in an application are run on suitable executors. In addition, load balancing is performed to maximize utilization of the available executors. In Constellation, a single *context-aware work stealing* algorithm is responsible for both. Whenever an executor becomes idle, it selects another executor and sends it a request for work that includes its context. The executor receiving the request can then perform context matching to find a suitable activity in its queues. If one is found, it is returned. Otherwise, the idle executor is notified that no work is available. This process is repeated until work is found.

21.5 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we discuss how we have used Constellation to create a prototype system for CBIR of hyperspectral remote sensing data for jungle computing systems. In our CBIR system, two main processes can be identified: (i) extraction of the endmembers from the actual images, and (ii) the execution of queries on the endmember data.

These two processes are tied together by the *endmember store*. The endmember store consists of the actual store, a database containing endmember information, and one or more specialized *endmember store executors* that can access the store. In an endmember store, the endmember information of each image is placed next to the names of the repositories in which the related image is stored. In addition, any extra metadata information that can be useful for querying (e.g., the date and location on which an image is taken) can be stored in the endmember store as well.

The endmember store manages the endmember extraction process for all repositories assigned to it. In turn, the end user can perform queries on the data stored in all endmember stores known to the end user. In our system, several endmember stores can be present, each of them managing their own set of repositories.

Figure 21.5 shows the activities that are involved in both the endmember extraction and query execution process, and Fig. 21.6 shows the executors of the CBIR system that must execute those activities. The labels assigned to both the activities

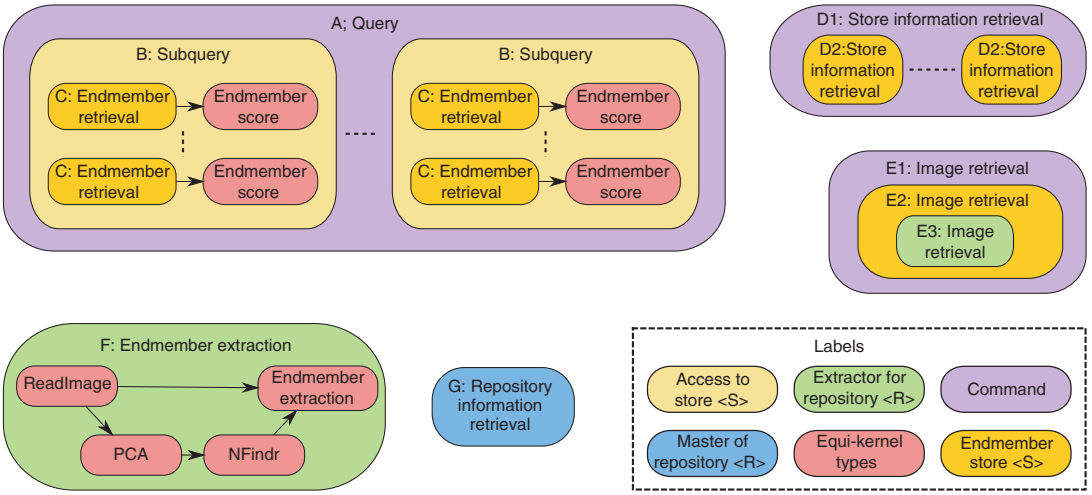

FIGURE 21.5 *Activities of the CBIR system and their labels. Activities can consist of one or more subactivities. The arrows show the data dependencies between activities.*

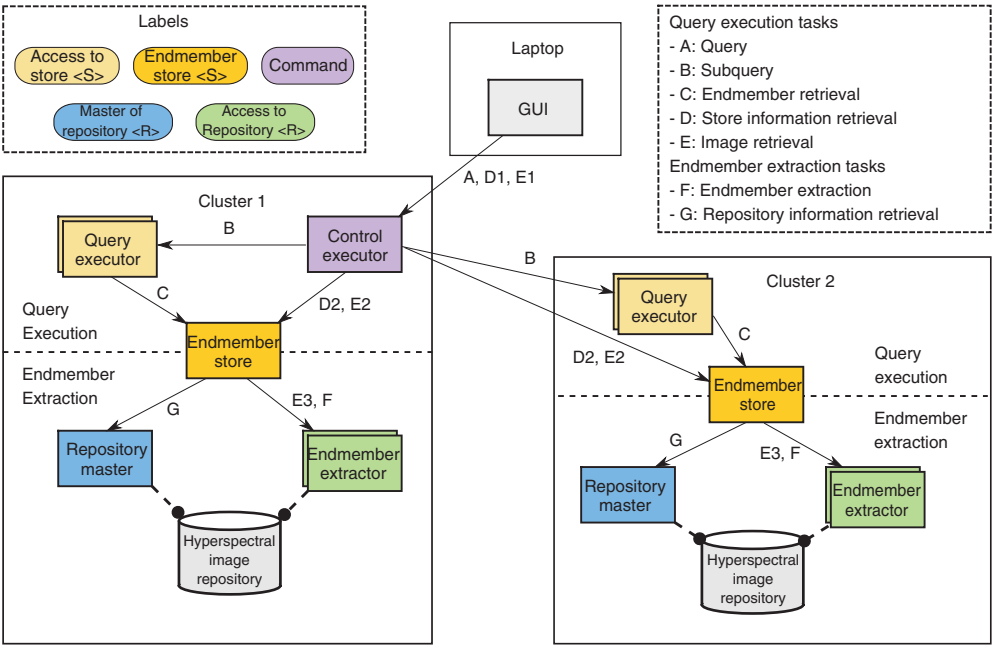

FIGURE 21.6 *Executors for the CBIR system, with the endmember store on the border between the endmember extraction and the query execution parts. The arrows show how the activities traverse the system to execute the query execution and endmember extraction tasks.*

and executors are used by the matchmaking mechanism of Constellation to assign each activity to a suitable executor, as explained in more detail in Section 21.5.4.

21.5.1 Endmember Extraction

In the process of endmember extraction, endmembers are calculated for each image in the repositories and placed in an endmember store. Next to the endmember store and the repositories themselves, this process also involves *endmember extractors*. The endmember extractors are executors capable of retrieving image data from a repository and execute the endmember extraction algorithm. Finally, a *repository master* executor is assigned to each repository. The task of a repository master is to monitor the contents of the repository it is assigned to.

To start the endmember extraction process, the endmember store first creates a *repository information retrieval activity* for each repository. In turn, this activity will start the repository monitoring process on the appropriate repository master executor. The repository master executor retrieves the information about the contents of the repository and sends it back to the endmember store using an event. When images are added to or removed from the repository later on, the repository master executor will notify the endmember store by sending additional events.

When the endmember store has received the information about the contents of the repositories, it creates endmember extraction activities for all images. The endmember extractors retrieve (using the Constellation job stealing mechanism) the activities that are suitable for their configuration and execute them, and finally they deliver the extracted endmembers to the endmember store.

21.5.2 Query Execution

As soon as the first endmembers are extracted and added to the endmember stores, queries can be executed on the contents of the endmember stores. All tasks involved in querying the stores are started by the user interface, but coordinated by the *control executor*, which acts as a proxy for the user interface. The user interface can start three operations: (i) retrieving the contents of all stores, (ii) executing a query, and (iii) retrieving an actual image from a repository.

21.5.2.1 Store Contents Retrieval The user interface can obtain a list of the contents of the stores by sending a *store information retrieval activity* to the control node. On receiving of such an activity, the control executor creates separate store information retrieval activities for each store. These activities will create a list of contents of a single store and send it back to the control executor by an event. In turn, the control node aggregates the results and sends them to the user interface.

21.5.2.2 Query Execution The user interface can execute a query by creating a *query activity*, which will be scheduled to the control node by the Constellation runtime system. To make the query easily distributable over several *query executors*, the control executor splits the query into *subquery activities*, each querying only a

small subset of the contents of the endmember stores. A subquery only contains query tasks for images that are processed by the same set of endmember stores.

Each subquery activity is assigned to an appropriate query executor by the matchmaking mechanism of Constellation. Next, the query executor creates comparison tasks for all images in the subquery. These comparison tasks consist of two steps: (i) retrieving the endmembers of the image out of the store, and (ii) calculating a similarity score between the image endmembers and the endmembers of the query image. When all comparisons are made, the images of the subquery are ordered by their score and sent back to the control node using an event.

The control node aggregates the results of all subqueries and orders the results by their scores. To conclude the query, an event containing a list of the best matching images is sent to the user interface.

21.5.2.3 Image Retrieval To retrieve an actual image, the user interface creates an *image retrieval activity* that is be processed by the control node. In turn, the control node looks up which stores contain the endmembers of the image, as these stores know how to locate a repository that contains the image. Then the control executor creates a new image retrieval activity which will be assigned to an appropriate store executor. This store executor creates another image retrieval request which is taken to a suitable endmember extraction executor. Finally, the endmember extraction executor retrieves the image from the repository and sends it back to the user interface using an event.

21.5.3 Equi-Kernels

If we regard the activities in Fig. 21.5 in more detail, we identify two classes of activities. Most of the activities described above, namely the *control activities*, deal with moving other activities or data within Constellation to a suitable location (i.e., executor). A second class of activities, the *operational activities*, perform the actual operations involved in either the query process or endmember extraction process. These activities are equipped with labels expressing *equi-kernel types* required for the operations.

Operational activities execute operations by invoking the corresponding kernel on the executor it runs on. Depending on the hardware platform it represents, an executor will use the best suitable *equi-kernel* available for each of its kernel operations.

Equi-kernels are different implementations of the same kernel functionality. For example, the endmember extraction activity, which is a control activity, contains amongst others the PCA and N-FINDR (operational) activities. These activities invoke the PCA and N-FINDR kernels, respectively. For both these kernels, a CPU and GPU equi-kernel are available, implemented in C++ and CUDA, respectively. Therefore, an endmember extraction executor representing a CPU resource will load the equi-kernels containing the CPU code, whereas executors that represent a GPU will load the GPU equi-kernels instead.

In our system, we use equi-kernels that differ in the hardware platform that is used to implement the kernel operation. However, equi-kernels can also differ from each

other in other ways. For example, they could use different heuristics to implement the same kernel functionality.

21.5.4 Matchmaking

The design of our CBIR system involves several types of executors, each with their own capabilities and roles in the overall system. Similarly, all subtasks involved in the CBIR process are represented by activities. To direct all activities to a suitable executor, we use the *labels* of the Constellation matchmaking system.

Each executor contains a label that describes its role. Additionally, a label describing the hardware represented by the executors is assigned as well. For example, an endmember extraction executor having access to repository "A" and representing a CPU core as well as a GPU will contain the following set of labels as its context: **[ExtractorForRepository< A>, GPU, CPU]**.

Similarly, each activity specifies on which type of executor it can run using the appropriate labels. For example, an endmember extraction activity for an image present in both repository "A" and "B" will have the label **[ExtractorForRepository<A>, ExtractorForRepository<B>]**, which expresses that it can be executed by any endmember extraction executor that has access to either repository "A" or "B."

In the same way, activities that represent a kernel operation carry a label that specifies the kernel type supported by that operation. The Constellation matchmaking mechanism allows executors to express their preference for a type of activity by ordering their labels accordingly. For example, an executor capable of running both CPU and GPU codes will contain the labels **[GPU, CPU]**, expressing that it prefers to execute activities that can use GPU equi-kernels. However, if such an activity is not present, it also accepts activities for which only CPU equi-kernels are available.

21.6 EVALUATION

The design for the CBIR system as described in the previous section fulfills most of the requirements for a jungle computing application. The Ibis platform already fully provides middleware independence, robust connectivity, and system-level fault tolerance, while it offers mechanisms to implement support for malleability and application-level fault tolerance. In addition, Constellation allows us to create a very flexible system that can easily adapt to all kinds of jungle environments.

First, it is very easy to exploit new resources by deploying extra executors on those new resources. Depending on the needs of the user and properties of the resources, the user can select the most appropriate executor.

Second, the matchmaking mechanism of Constellation and its labeling approach makes it very simple to configure the behavior of the CBIR system. For example, we could alter the label of the image retrieval activity in such a way that the image is retrieved by the repository master instead of the endmember extractor by changing a label of the image retrieval activity E3 (Fig. 21.5) to **[MasterOfRepository< A>]**.

Of course, the system designer has to take care not to alter the labels in such a way that activities will be assigned to executors that are unable to process the activity.

Third, the equi-kernel approach allows integration with external software while maintaining resource independence. For each operation, a special code for each platform can be included. By including a "default" equi-kernel for each operation as well, the CBIR system can transparently exploit special hardware and codes, without failing to operate when such hardware is not available.

21.6.1 Performance Evaluation

Now that we have established that our design for the CBIR system leads to a flexible solution suitable for jungle computing systems, we will validate the performance of our system. In these experiments, we use the DAS-4, which is a heterogeneous computing system consisting of six computing clusters spread over six universities and research institutes in The Netherlands. Table 21.1 lists the most important properties of the computing nodes for each cluster site.node configurations of all cluster site. Finally, all clusters are equipped with an 40 Gb/s Infiniband local network, and the DAS-4 clusters are connected to each other via a 10 Gb/s optical network.

In our experiments, we used a subset of the AVIRIS dataset of the 2010 Gulf Oil Spill Response.[4] This dataset consists of several *flight lines* of hyperspectral data, each of which can easily contain several tens of gigabytes of uncompressed data. As these flight lines are too large to process as a whole, they are split into image tiles of up to 512×512 pixels by our repository I/O kernels.

Our CBIR framework for remote sensing data executes two tasks, namely endmember extraction and query execution, largely independent of each other. We will first evaluate these tasks separately. For these experiments, we used the DAS-4 cluster located at VU University in Amsterdam. We also evaluated the CBIR system in a jungle environment. For this experiment, in addition to the DAS-4 cluster at the VU, we also used the DAS-4 clusters located at Leiden University, University of Amsterdam, TU Delft, and ASTRON.

21.6.1.1 Endmember Extraction In our first tests, we evaluated the performance of the endmember extraction process of the CBIR system. The main activity

TABLE 21.1 Node Configurations of the DAS-4 Clusters

Cluster	Nodes	CPU	Memory	Accelerators
VU Amsterdam	74	2 quad-core Intel E5620	24 GB	16x GTX480
Leiden University	16	2 quad-core Intel E5620	48 GB	
Univ. of Amsterdam	16	2 quad-core Intel E5620	24 GB	
TU Delft	32	2 quad-core Intel E5620	24 GB	8x GTX480
ASTRON	24	2 quad-core Intel E5620	24 GB	8x GTX580

[4] http://aviris.jpl.nasa.gov/html/gulfoilspill.html

of the endmember extraction phase is the endmember extraction activity. As already shown in Fig. 21.5, the endmember extraction activity consists of four kernel activities, each performing a single kernel operation:

1. `LoadImage`: Load an image from a repository into main memory;
2. `PCA`: The PCA dimensionality reduction Algorithm;
3. `N-FINDR`: The N-FINDR endmember extraction algorithm: Computes the endmembers in the reduced image space;
4. `EndmemberExtraction`: Extracts the endmembers (that are found by the N-FINDR algorithm) from original image.

In addition to loading image data from the repository, the `LoadImage` kernel converts the image data to the correct format for use in our algorithm as well. For both PCA and N-FINDR kernels, we have a CPU and GPU implementation. The `EndmemberExtraction` kernel is a simple operation that extracts the endmembers found by the N-FINDR kernel from the original image data, and is not significant in the system performance.

We benchmarked the endmember extraction process for two scenarios:

- *Static Repositories:* A repository contains a constant number of images. No new images are added at runtime;
- *Streaming Data:* New image data is delivered to the system at runtime. The CBIR system processes the data immediately and adds the endmembers to the system as soon as possible.

Although the functionality is similar for both scenarios, the evaluation criteria differ. In case of a static repository, we are mainly interested in the time needed to process the entire repository. In case of streaming data, we are interested in how many resources are needed in order to "keep up" with the data stream. Next to these two scenarios, a third scenario could be formulated in which new data is added to an already existing repository. As this scenario essentially is a combination of two scenarios we described above, we will not discuss it any further.

For the static repository scenario, we used the NFS file system of the DAS-4 cluster as our repository. In this repository, we stored a 77 GB dataset consisting of 13 flight lines of oil spill data, resulting in 784 image tiles of up to 512×512 pixels. We measured the time our system used to extract the endmembers for all image tiles and store the endmember information in the endmember store. To emulate a data stream for the streaming data scenario, we used randomly generated data as input image data, eliminating the influence of the I/O speed of the DAS-4 NFS file system.

Table 21.2a shows the time required by the endmember extraction process to process all 784 image tiles of the static repository, using only CPU kernels. As the extraction executors operate completely independently of each other, we initially expected the performance of to scale close to linear with respect to the number of executors in the system. However, as more executors are used on a single node, we

TABLE 21.2 Static Repository: Time (in seconds) Needed to Extract the Endmembers of All 784 Image Tiles: (a) Only the CPU is Used; (b) One GPU Per Node Static Repository, and One GPU Per Node

		Executors per Node			
		1	2	4	8
Nodes	1	7 951	4 680	4 048	3 581
	2	3 555	2 546	2 190	796
	4	1 853	1 261	1 082	595
	8	971	683	643	683

(a)

		Executors per Node			
		1	2	4	8
Nodes	1	6 103	5 148	3 998	1 434
	2	2 998	2 283	1 957	816
	4	1 393	1 147	635	484
	8	777	503	491	517

(b)

can clearly see that the performance increase is not as large as expected. The main source of this decrease in performance is the `LoadImage` kernel: the execution times of the `LoadImage` kernel rises with the addition of extra executors.

When we increase the number of endmember extraction nodes in the system, we initially do get close-to-linear performance increase as expected. However, the benefits of increasing the number of executors for the endmember extraction process decrease quickly, and using more than 16 executors in total hardly improves overall performance any further. This indicates that we hit a bottleneck in our system. Most likely, the NFS file system used by our repository implementation is not able to keep up with the I/O requests made by the endmember extraction executors anymore. The best results were obtained using four nodes with eight executors each, leading to a speedup factor of 13.

Looking at the execution times in more detail, we can see that the `LoadImage` kernel behaves unpredictably and takes a relatively long time. For higher numbers of executors, we notice that loading the images from the repositories is the bottleneck of the entire endmember extraction pipeline. For one part, this can be attributed to the bandwidth of the repository to the computing nodes. However, we noticed that the I/O performance is affected by another factor as well: in the repositories, the image data is ordered using the "band interleaved per pixel" format, whereas the endmember extraction algorithm expects the data to be in the "band sequential" order. Therefore, the images are converted to the band sequential layout by the input kernel immediately. This conversion step, which is a matrix transposition, appeared to be significant as well. With up to eight endmember extractors in total, an entire I/O operation (including conversion) takes about 6 s for a 512×512 tile. However, when we increase the number of executors to 32, the I/O time increases to 13 s on average.

When we include one GPU executor per node (Table 21.2b), the time required to process all images is reduced significantly. In case of a single executor per node, we measure a reduction in execution time of 15–25%. This advantage quickly reduces when more (CPU) executors are added to each node. Although the GPU executor is significantly faster in computing the endmembers, the overall benefit of using the GPU is limited because of the large impact of the I/O on overall system performance.

For the PCA kernel, the GPU implementation is about 3 times faster than the CPU implementation, whereas for N-FINDR, the performance increases by up to a factor

TABLE 21.3 Streaming Data. Extraction Speed (Tiles/s). (a) With CPU Only. (b) With One GPU Per Node

		Executors per Node			
		1	2	4	8
Nodes	1	0.0829	0.163	0.303	0.478
	2	0.157	0.307	0.584	0.906
	4	0.358	0.639	1.14	1.82
	8	0.725	1.25	2.17	3.26

(a)

		Executors per Node			
		1	2	4	8
Nodes	1	0.797	0.787	0.884	0.834
	2	1.62	1.54	1.64	1.54
	4	2.89	3.01	3.15	3.13
	8	5.90	6.30	6.51	5.86

(b)

16 when GPUs are used. When multiple executors are deployed, we can see that the performance of the CPU kernel slightly degrades, probably due to sharing computing resources and memory bandwidth.

In the streaming data scenario, we can see that adding extra executors within a computing node results in improved performance even if more than 16 executors are used (Table 21.3a). This shows that, in the static repository scenario, performance was indeed limited by the I/O performance. Tables 21.3a and 21.3b show that the performance of the endmember extraction process now shows close to linear scalability in the number of nodes used.

With the I/O bottleneck eliminated, using GPUs in the endmember extraction leads to a significantly higher throughput. In this scenario, a single GPU executor reaches a higher throughput as an entire CPU node with eight executors. However, Table 21.3b shows that adding extra CPU executors to the nodes that already have a GPU executor does not lead to significant performance improvements.

21.6.1.2 Query Execution Next we evaluated how fast our system is able to execute queries on the endmember data stored in the endmember stores. To that end, we performed queries on stores containing 10,000 and 100,000 elements, using several configurations of query executors (Table 21.4). Our query system is configured to retrieve the 30 most similar elements from the endmember store.

In our experiments with a single query executor and both 10,000 and 100,000 images in the endmember store, roughly 1 ms is needed on average to process a single image. The image comparison kernel itself needs about 0.6 ms to compare

TABLE 21.4 Query Speed: Time (in seconds) to Complete A Query: (a) 10,000 Images; (b) 100,000 Images

		Executors per Node			
		1	2	4	8
Nodes	1	10.24	5.43	3.02	1.91
	2	5.36	3.02	1.79	1.15
	4	2.98	1.89	1.12	0.84

(a)

		Executors per Node			
		1	2	4	8
Nodes	1	100.74	50.83	26.09	14.37
	2	50.03	25.52	13.35	7.54
	4	24.72	12.92	6.96	4.12

(b)

endmember sets with each other. The remainder of the time is spent in sorting the results and overhead of the system runtime itself.

When 100,000 images are used, performance of our system scales linearly with the number of executors used up to about eight executors. In case a total of 32 executors spread over four nodes are used, a speedup of 24.5 is achieved compared to when using a single executor. In case of 10,000 images, scalability of the system is worse, and using 32 executors only leads to a speedup of 12.2.

With 32 executors distributed over four nodes, our system is capable of executing a query within 1 s in case of 10,000 images, and in about 4 s in case of 100,000 images in the endmember store.

21.6.1.3 Jungle Deployment Finally, we show that our system is capable of running in a jungle computing scenario by deploying it on five clusters (VU, UvA, Leiden, Delft, and Astron) of the DAS-4 system, using CPU nodes only. Over these cluster sites, we distributed 40 flight lines of hyperspectral imaging data, for a total of 221.9 GB of data (Table 21.5). The dataset on the VU cluster is the same as used for the static repository experiment.

On each cluster site, we deployed a separate node with a repository master, four nodes with four endmember extractors each, a single node containing the endmember store and a single store executor, and one node with eight query executors to query the endmember store. We deployed the control executor on a node on the DAS-4 cluster at the VU and the GUI on a local workstation.

It took 990 s (16:30 min) to complete the endmember extraction process on all image data, which is similar to the time needed to process the VU dataset in isolation (Table 21.2a), which is the largest dataset. This shows that the CBIR system did not introduce any new overheads when processing multiple repositories concurrently. A query on the full dataset (2464 image tiles) distributed over five endmember stores could be easily completed within 1 s, leading to a smooth and interactive querying experience over the distributed data.

By exploiting several clusters within a single jungle computing system, our CBIR system is able to process the contents of multiple repositories in parallel, without any additional overheads. At the same time, it enables the user to perform queries on the contents of multiple repositories interactively, completely hiding the complexities of the underlying system.

TABLE 21.5 Data Distribution

Cluster Site	Flight Lines	Tiles (512 × 512)	Data Size (GB)
VU Amsterdam	13	784	77.0
Leiden	9	491	38.2
UvA	3	288	24.5
Delft	7	351	30.2
Astron	8	550	52.0
Total	40	2464	221.9

21.7 CONCLUSIONS

In this chapter, we discussed the ever-increasing amounts of data generated by remote sensing instruments, and the challenges that the research domain of remote sensing has to face to process all these data. We argued that the application of jungle computing techniques can help in processing these huge amounts of data.

Next we introduced the Ibis/Constellation, a platform for creating flexible and efficient applications for jungle computing systems. We described how Ibis/Constellation can be used to build a system for CBIR for hyperspectral remote sensing data, which offers a search functionality on the contents of multiple distributed data repositories in a single system.

In this system, we can distinguish two independent processes, linked together by the endmember store. First, in the endmember extraction process, endmembers are extracted of each image present in a repository, and the endmembers are placed in an endmember store. Next, the query process is able to execute queries on the data that is stored in the endmember stores.

We showed that our prototype implementation of the CBIR system can be easily adapted to match the properties of a jungle computing system, and can exploit the available computing resources in an efficient way. Finally, we demonstrated that our prototype system can indeed provide query functionality on multiple distributed data repositories in a transparent and efficient way.

ACKNOWLEDGMENTS

This work was supported by COST Action IC0805 “Open European Network for High Performance Computing on Complex Environments,” and partially funded by the Dutch national research program COMMIT.

REFERENCES

1. D. E. Wojick, W. L. Warnick, B. C. Carroll, and J. Crowe, “The digital road to scientific knowledge diffusion: a faster, better way to scientific progress?” *D-Lib Magazine*, vol. 12, no. 6, 2006.
2. I. Taylor, I. Wang, M. Shields, and S. Majithia, “Distributed computing with Triana on the grid,” *Concurrency and Computation: Practice and Experience*, vol. 17, no. 9, pp. 1197–1214, 2005.
3. J. Maassen and H. E. Bal, “Smartsockets: solving the connectivity problems in grid computing,” in *Proceedings of the 16th International Symposium on High Performance Distributed Computing, HPDC’07*, pp. 1–10. New York: ACM, 2007.
4. R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauvé, “Faults in grids: why are they so bad and what can be done about it?” in *Proceedings of the 4th International Workshop on Grid Computing, GRID’03*, pp. 18–24. Washington, DC: IEEE Computer Society, 2003.

5. F. J. Seinstra, J. Maassen, R. V. van Nieuwpoort, N. Drost, T. van Kessel, B. van Werkhoven, J. Urbani, C. Jacobs, T. Kielmann, and H. E. Bal, "Jungle computing: distributed supercomputing beyond clusters, grids, and clouds," in *Grids, Clouds and Virtualization*, *Computer Communications and Networks* (M. Cafaro and G. Aloisio, eds.), pp. 167–197. London: Springer, 2011.
6. J. Maassen, N. Drost, H. E. Bal, and F. J. Seinstra, "Towards jungle computing with Ibis/Constellation," in *Proceedings of the 2011 Workshop on Dynamic Distributed Data-Intensive Applications, Programming Abstractions, and Systems, 3DAPAS'11*, pp. 7–18. New York: ACM, 2011. http://doi.acm.org/10.1145/1996010.1996013.
7. R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine, "Open MPI: a high-performance, heterogeneous MPI," in *Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, pp. 1–9, 2006.
8. A. J. Plaza, J. Plaza, and A. Paz, "Parallel heterogeneous CBIR system for efficient hyperspectral image retrieval using spectral mixture analysis," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 9, pp. 1138–1159, 2010.
9. N. Drost, J. Maassen, M. A. J. van Meersbergen, H. E. Bal, F. I. Pelupessy, S. P. Zwart, M. Kliphuis, H. A. Dijkstra, and F. J. Seinstra, "High-performance distributed multi-model / multi-kernel simulations: a case-study in jungle computing," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW'12*, pp. 150–162. Washington, DC: IEEE Computer Society, 2012.
10. R. O. Green, M. L. Eastwood, C. M. Sarture, T. G. Chrien, M. Aronsson, B. J. Chippendale, J. A. Faust, B. E. Pavri, C. J. Chovit, M. Solis, M. R. Olah, and O. Williams, "Imaging spectroscopy and the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS)," *Remote Sensing of Environment*, vol. 65, no. 3, pp. 227–248, 1998.
11. A. W. M. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain, "Content-based image retrieval at the end of the early years," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 12, pp. 1349–1380, 2000.
12. C.-I. Chang, *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. New York: Plenum Publishing Co., 2003.
13. A. Plaza, J. A. Benediktsson, J. W. Boardman, J. Brazile, L. Bruzzone, G. Camps-Valls, J. Chanussot, M. Fauvel, P. Gamba, A. Gualtieri, M. Marconcini, J. C. Tilton, and G. Trianni, "Recent advances in techniques for hyperspectral image processing," *Remote Sensing of Environment*, vol. 113, (Suppl. 1), no. 0, pp. S110–S122–, 2009. Imaging Spectroscopy Special Issue.
14. G. Camps-Valls and L. Bruzzone, "Kernel-based methods for hyperspectral image classification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 43, no. 6, pp. 1351–1362, 2005.
15. N. Keshava and J. Mustard, "Spectral unmixing," *IEEE Signal Processing Magazine*, vol. 19, no. 1, pp. 44–57, 2002.
16. D. C. Heinz and C.-I. Chang, "Fully constrained least squares linear spectral mixture analysis method for material quantification in hyperspectral imagery," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 39, no. 3, pp. 529–545, 2001.
17. M. E. Winter, "N-FINDR: an algorithm for fast autonomous spectral end-member determination in hyperspectral data," *Proceedings of SPIE*, vol. 3753, pp. 266–275, 1999.
18. J. A. Richards and X. Jia, *Remote Sensing Digital Image Analysis: An Introduction*. Berlin: Springer, 2006.
19. J. Bioucas-Dias and J. Nascimento, "Hyperspectral subspace identification," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 46, no. 8, pp. 2435–2445, 2008.

20. I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: enabling scalable virtual organizations," *International Journal of High-Performance Computing Applications*, vol. 15, no. 3, pp. 200–222, 2001.
21. J. I. Khan and A. Wierzbicki, "Guest editor's introduction; foundation of peer-to-peer computing," *Computer Communications*, vol. 31, no. 2, pp. 187–189, 2008.
22. "Editorial: Cloud computing: Clash of the clouds," in *The Economist*, 2009, Retrieved from http://www.economist.com/node/14637206.
23. H. E. Bal, J. Maassen, R. V. van Nieuwpoort, N. Drost, R. Kemp, N. Palmer, G. Wrzesinska, T. Kielmann, F. Seinstra, and C. Jacobs, "Real-world distributed computing with Ibis," *Computer*, vol. 43, no. 8, pp. 54–62, 2010.
24. M. Reilly, "When multicore isn't enough: trends and the future for multi-multicore systems," in *Proceedings of the 12th Annual Workshop on High-Performance Embedded Computing (HPEC 2008)*, Lexington, MA, USA, 2008.
25. P. J. Lu, H. Oki, C. A. Frey, G. E. Chamitoff, L. Chiao, E. M. Fincke, C. M. Foale, S. H. Magnus, W. S. Mcarthur Jr., D. M. Tani, P. A. Whitson, J. N. Williams, W. V. Meyer, R. J. Sicker, B. J. Au, M. Christiansen, A. B. Schofield, and D. A. Weitz, "Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station," *Journal of Real-Time Image Processing*, vol. 5, no. 3, pp. 179–193, 2010.
26. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.
27. D. Abramson, R. Sosic, J. Giddy, and B. Hall, "Nimrod: a tool for performing parametised simulations using distributed workstations," in *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing (HPDC'95)*, Washington, DC, USA, pp. 112–121, 1995.
28. K. Verstoep, J. Maassen, H. E. Bal, and J. W. Romein, "Experiences with fine-grained distributed supercomputing on a 10G testbed," in *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid, CCGRID'08*, pp. 376–383. Washington, DC: IEEE Computer Society, 2008.
29. D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience: research articles," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.
30. I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: a Fast and Light-weight tasK executiON framework," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC'07*, Reno, NV, USA, pp. 1–12, 2007.
31. E. Walker, J. P. Gardner, V. Litvin, and E. L. Turner, "Personal adaptive clusters as containers for scientific jobs," *Cluster Computing*, vol. 10, no. 3, pp. 339–350, 2007.
32. N. Drost, R. V. van Nieuwpoort, J. Maassen, F. J. Seinstra, and H. E. Bal, "Zorilla: a peer-to-peer middleware for real-world distributed systems," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 13, pp. 1506–1521, 2011.
33. B. Bode, D. M. Halstead, R. Kendall, Z. Lei, and D. Jackson, "The portable batch scheduler and the maui scheduler on linux clusters," in *Proceedings of the 4th Annual Linux Showcase & Conference—Volume 4, ALS'00*, p. 27. Berkeley, CA: USENIX Association, 2000.