

Jungle Computing: Distributed Supercomputing beyond Clusters, Grids, and Clouds

Frank J. Seinstra, Jason Maassen, Rob V. van Nieuwpoort, Niels Drost, Timo van Kessel, Ben van Werkhoven, Jacopo Urbani, Cerial Jacobs, Thilo Kielmann, and Henri E. Bal

Abstract In recent years, the application of high-performance and distributed computing in scientific practice has become increasingly wide-spread. Among the most widely available platforms to scientists are clusters, grids, and cloud systems. Such infrastructures currently are undergoing revolutionary change due to the integration of many-core technologies, providing orders-of-magnitude speed improvements for selected compute kernels. With high-performance and distributed computing systems thus becoming more heterogeneous and hierarchical, programming complexity is vastly increased. Further complexities arise because urgent desire for scalability, and issues including data distribution, software heterogeneity, and ad-hoc hardware availability, commonly force scientists into *simultaneous use of multiple platforms* (e.g. clusters, grids, and clouds used concurrently). A true *computing jungle*.

In this chapter we explore the possibilities of enabling efficient and transparent use of *Jungle Computing Systems* in every-day scientific practice. To this end, we discuss the fundamental methodologies required for defining programming models that are tailored to the specific needs of scientific researchers. Importantly, we claim that many of these fundamental methodologies already exist today, as integrated in our Ibis high-performance distributed programming system. We also make a case for the urgent need for easy and efficient Jungle Computing in scientific practice, by exploring a set of state-of-the-art application domains. For one of these domains we present results obtained with Ibis on a real-world Jungle Computing System. The chapter concludes by exploring fundamental research questions to be investigated in the years to come.

Frank J. Seinstra, Jason Maassen, Rob V. van Nieuwpoort, Niels Drost, Timo van Kessel, Ben van Werkhoven, Jacopo Urbani, Cerial Jacobs, Thilo Kielmann, Henri E. Bal
Department of Computer Science, Vrije Universiteit, De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands
e-mail: fjseins, jason, rob, niels, timo, ben, jacopo, cerial, kielmann, bal@cs.vu.nl

1 Introduction

It is widely recognized that Information and Communication Technologies (ICTs) have revolutionized the everyday practice of science [6, 54]. Whereas in earlier times scientists spent a lifetime recording and analyzing observations by hand, in many research laboratories today much of this work has been automated. The benefits of automation are obvious: it allows researchers to increase productivity by increasing efficiency, to improve quality by reducing error, and to cope with increasing scale — enabling scientific treatment of topics that were previously impossible to address.

As a direct result of automation, in many research domains the rate of scientific progress is now faster than ever before [11, 14, 59]. Importantly, however, the rate of progress itself puts further demands on the automation process. The availability of ever larger amounts of observational data, for example, directly leads to increasing needs for computing, networking and storage. As a result, for many years the scientific community has been one of the major driving forces behind state-of-the-art developments in supercomputing technologies (e.g., see [58]).

Although this self-stimulating process indeed allows scientists today to study more complex problems than ever before, it has put a severe additional burden on the scientists themselves. Many scientists have to rely on arguably the most complex computing architectures of all — i.e., high-performance and distributed computing systems in their myriad of forms. To effectively exploit the available processing power, a thorough understanding of the complexity of such systems is essential. As a consequence, the number of scientists capable of using such systems effectively (if at all) is relatively low [44].

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 [5, 24]. Worse even, the high-performance and distributed computing landscape is currently undergoing 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 [31, 32]). 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 compute power, and due to additional issues including data distribution, software heterogeneity, and ad-hoc hardware availability, 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*.

In this chapter we explore the possibilities of enabling efficient and transparent use of Jungle Computing Systems in every-day scientific practice. To this end we focus on the following research question:

What are the fundamental methodologies required for defining programming models that are tailored to the specific needs of scientific researchers, and that match state-of-the-art developments in high-performance and distributed computing architectures?

We will claim that many of these fundamental methodologies already exist, and have been integrated in our Ibis software system for high-performance and distributed computing [4]. In other words: Jungle Computing is not just a visionary concept; to a large extent we already adhere to its requirements today.

This chapter is organized as follows. In Section 2 we discuss several architectural revolutions that are currently taking place — leading to the new notion of Jungle Computing. Based on these groundbreaking developments, Section 3 defines the general requirements underlying transparent programming models for Jungle Computing Systems. Section 4 discusses the Ibis programming system, and explores to what extent Ibis adheres to the requirements of Jungle Computing. Section 5 sketches a number of emerging problems in various science domains. For each domain we will stress the need for Jungle Computing solutions that provide transparent speed and scalability. For one of these domains, Section 6 evaluates the Ibis platform on a real-world Jungle Computing System. Section 7 introduces a number of fundamental research questions to be investigated in the coming years, and concludes.

2 Jungle Computing Systems

When 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* [18]. Since then, many other distributed computing paradigms have been introduced, including peer-to-peer computing [25], volunteer computing [57], and — more recently — cloud computing [15]. 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 world-wide 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, etcetera. 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 compute 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 middlewares, programming interfaces, access policies, and protection mechanisms [4].

Apart from the increasing diversity in the distributed computing landscape, the 'basic compute nodes' mentioned above currently are undergoing revolutionary change as well. General purpose CPUs today have multiple compute cores per chip, with an expected increase in the years to come [40]. Moreover, special purpose chips (e.g., GPUs [31, 32]) are now combined or even integrated with CPUs to increase performance by orders-of-magnitude (e.g., see [28]).

The many-core revolution is already affecting the field of high-performance and distributed computing today. One interesting example is the Distributed ASCI Supercomputer 4 (DAS-4), which is currently being installed in The Netherlands. This successor to the earlier DAS-3 system (see www.cs.vu.nl/das3/), will consist of 6 clusters located at 5 different universities and research institutes, with each cluster being connected by a dedicated and fully optical wide-area connection. Notably, each cluster will also contain a variety of many-core 'add-ons' (including a.o. GPUs and FPGAs), making DAS-4 a highly diverse and heterogeneous system. The sheer number of similar developments currently taking place the world over indicates that many-cores are rapidly becoming an irrefutable additional component of high-performance and distributed systems.

With clusters, grids, and clouds thus being equipped with multi-core processors and many-core 'add-ons', systems available to scientists are becoming increasingly hard to program and use. Despite the fact that the programming and efficient use of many-cores is known to be hard [31, 32], 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. The need to access multiple platforms concurrently from within a single application often is due to the impossibility of reserving a sufficient number of compute nodes at once in a single multi-user system. Moreover, additional issues such as the distributed nature of the input

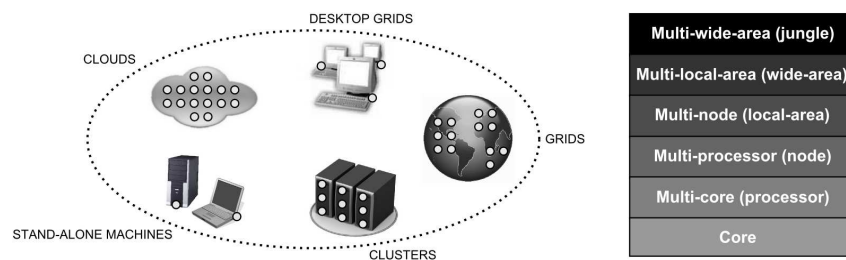


Fig. 1: Left: A 'worst-case' Jungle Computing System as perceived by scientific end-users, simultaneously comprising any number of clusters, grids, clouds, and other computing platforms. Right: Hierarchical view of a Jungle Computing System.

data, the heterogeneity of the software pipeline being applied, and the 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, in this chapter we make the following claim:

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* (see Figure 1).

The abovementioned claim is not new. As part of the European Grid Initiative (EGI [55]), for example, it has been stressed that the integrated application of clusters, grids, and clouds in scientific computing is a key component of the research agenda for the coming years [9]. Similarly, Microsoft Research has advocated the integrated use of grids and clouds [16]. Further European research efforts in this direction are taking place in COST Action IC0805 (ComplexHPC: Open European Network for High-Performance Computing in Complex Environments [56]).

Compared to these related visionary ideas, the notion of a Jungle Computing System is more all-encompassing. It exposes *all* computing problems that scientists today can be (and often are) confronted with. Even though we do not expect most (or even any) research scientists to have to deal with the 'worst case' scenario depicted in Figure 1, we do claim 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.

3 Jungle Computing: Requirements and Methodologies

Although Jungle Computing Systems and grids are not identical (i.e., the latter being constituent components of the former), a generic answer to our overall research question introduced in Section 1 is given by the 'founding fathers of the grid'. In [18], Foster et al. 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 [29, 47] and parameter sweeps [1]), 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 [53].

We ascribe this rather limited use of grids and other distributed systems — or the lack of efficient and transparent programming models — to the intrinsic complex-

ities 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. All these problems limit the acceptance of the many distributed computing technologies available today.

In our research we aim to overcome these problems, and to drastically simplify the programming and deployment of distributed supercomputing applications — without limiting the set of target hardware platforms. Importantly, our philosophy is that Jungle Computing applications should be developed on a local workstation and simply be launched from there. This philosophy directly leads to a number of fundamental requirements underlying the notion of Jungle Computing. In the following we will give a high-level overview of these requirements, and indicate how these requirements are met with in our Ibis distributed programming system.

3.1 Requirements

The abovementioned 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 a.o. by introducing 'intelligent' optimization techniques, application domain-specific knowledge, etcetera.

In our view, for full transparency, the end-user must be shielded from *all* issues that complicate the programming and use of Jungle Computing Systems in comparison with the programming and use of a desktop computer. To this end, methodologies must be available that provide transparent support for:

- **Resource independence.** In large-scale Jungle Computing Systems heterogeneity is omnipresent, to the effect that applications designed for one system are generally guaranteed to fail on others. This problem must be removed by hiding the physical characteristics of resources from end-users.
- **Middleware independence and interoperability.** As many resources already have at least one middleware installed, Jungle Computing applications must be able to use (or: interface with) such *local middlewares*. To avoid end-users having to implement a different interface for each local middleware (and to enhance portability), it is essential to have available a single high-level interface on top of *all* common middleware systems. As multiple distributed resources may use different middlewares, some form of interoperability between these middlewares must be ensured as well.

- **Robust connectivity and globally unique resource naming.** Getting distributed applications to execute at all in a Jungle Computing System is difficult. This is because firewalls, transparent renaming of IP addresses, and multi-homing (machines with multiple addresses), can severely complicate or limit the ability of resources to communicate. Moreover, in many cases no direct connection with certain machines is allowed at all. Despite solutions that have been provided for firewall issues (e.g., NetIbis [10], Remus [46]), integrated solutions must be made available that remove connectivity problems altogether. At the same time, and in contrast to popular communication libraries such as MPI, each resource must be given a globally unique identifier.
- **Malleability.** In a Jungle Computing System, the set of available resources may change, e.g. because of reservations ending. Jungle Computing software must support malleability, correctly handling resources joining and leaving.
- **System-level fault-tolerance.** Given the many independent parts of a large-scale Jungle Computing System, the chance of resource failures is high. Jungle Computing software must be able to handle such failures in a graceful manner. Failures should not hinder the functioning of the entire system, and failing resources should be detected, and if needed (and possible) replaced.
- **Application-level fault-tolerance.** The capacity of detecting resource failures, and replacing failed resources, is essential functionality for any realistic Jungle Computing System. However, this functionality in itself can not guarantee the correct continuation of running applications. Hence, restoring the state of applications that had been running on a failed resource is a further essential requirement. Such functionality is generally to be implemented either in the application itself, or in the run-time system of the programming model with which an application is implemented. Support for application-level fault-tolerance in the lower levels of the software stack can be limited to failure detection and reporting.
- **Parallelization.** For execution on any Jungle Computing system, it is generally up to the programmer to identify the available parallelism in a problem at hand. For the programmer — generally a domain expert with limited or no expertise in distributed supercomputing — this is often an insurmountable problem. Clearly, programming models must be made available that hide most (if not all) of the inherent complexities of parallelization.
- **Integration with external software.** It is unrealistic to assume that a single all-encompassing software system would adhere to all needs of all projected users. In many scientific research domains there is a desire for integrating 'black box' legacy codes, while the expertise or resources to rewrite such codes into a newly required format or language are lacking. Similarly, it is essential to be able to integrate system-level software (e.g. specialized communication libraries) and architecture-specific compute kernels (e.g. CUDA-implemented algorithms for

GPU-execution). While such 'linking up' with existing and external software partially undermines our 'write-and-go' philosophy, this property is essential for a software system for Jungle Computing to be of any use to general scientific researchers.

Our list of requirements is by no means complete; it merely consists of a *minimal* set of methodologies that — in combination — fulfill our high-level requirement of transparency. Further requirements certainly also exist, including support for co-allocation, security, large-scale distributed data management, non-centralized control, runtime adaptivity, the handling of quality-of-service (QoS) constraints, and run-time monitoring and visualization of application behavior. These are secondary requirements, however, and are not discussed further in this chapter.

4 Ibis

The Ibis platform (see also www.cs.vu.nl/ibis/) aims to combine all of the stated fundamental methodologies into a single integrated programming system that applies to *any* Jungle Computing System (see Figure 2). Our open source software system provides high-level, architecture- and middleware-independent interfaces that allow for (transparent) implementation of efficient applications that are robust to faults and dynamic variations in the availability of resources. To this end, Ibis consists of a rich software stack that provides all functionality that is traditionally associated with *programming languages* and *communication libraries* on the one hand, and *operating systems* on the other. More specifically, Ibis offers an integrated, layered solution, consisting of two subsystems: the High-Performance Programming System and the Distributed Deployment System.

4.1 The Ibis High-Performance Programming System

The Ibis High-Performance Programming System consists of (1) the IPL, (2) the programming models, and (3) SmartSockets, described below.

(1) The Ibis Portability Layer (IPL): The IPL is at the heart of the Ibis High-Performance Programming System. It is a communication library which is written entirely in Java, so it runs on any platform that provides a suitable Java Virtual Machine (JVM). The library is typically shipped with the application (as Java jar files), such that no preinstalled libraries need to be present at any destination machine. The IPL provides a range of communication primitives (partially comparable to those provided by libraries such as MPI), including point-to-point and multicast communication, and streaming. It applies efficient protocols that avoid copying and other overheads as much as possible, and uses *bytecode rewriting* optimizations for efficient transmission.

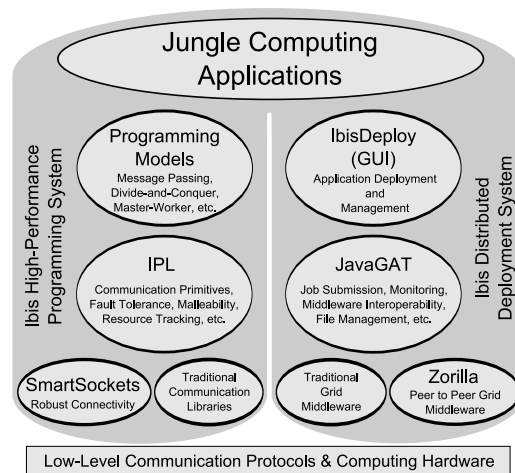


Fig. 2: Integration of the required methodologies in the Ibis software system. See also <http://www.cs.vu.nl/ibis/>

To deal with real-world Jungle Computing Systems, in which resources can crash, and can be added or deleted, the IPL incorporates a globally unique resource naming scheme, as well as a runtime mechanism that keeps track of the available resources. The mechanism, called Join-Elect-Leave (JEL [13]), is based on the concept of signaling, i.e., notifying the application when resources have Joined or Left the computation. JEL also includes Elections, to select resources with a special role. The IPL contains a centralized implementation of JEL, which is sufficient for static (closed-world) programming models (like MPI), as well as a more scalable distributed implementation based on gossiping.

The IPL has been implemented on top of the socket interface provided by the JVM, and on top of our own SmartSockets library (see below). Irrespective of the implementation, the IPL can be used 'out of the box' on any system that provides a suitable JVM. In addition, the IPL can exploit specialized native libraries, such as a Myrinet device driver (MX), if it exists on the target system. Further implementations of the IPL exist, on top of MPI, and on the Android smart phone platform.

(2) Ibis Programming Models: The IPL can be (and has been) used directly to write applications, but Ibis also provides several higher-level programming models on top of the IPL, including (1) an implementation of the MPJ standard, i.e. an MPI version in Java, (2) Satin, a divide-and-conquer model, described below, (3) Remote Method Invocation (RMI), an object-oriented form of Remote Procedure Call, (4) Group Method Invocation (GMI), a generalization of RMI to group communication, (5) Maestro, a fault-tolerant and self-optimizing data-flow model, and (6) Jorus, a user transparent parallel programming model for multimedia applications discussed in Section 6.

Arguably the most transparent model of these is Satin [60], a divide-and-conquer system that automatically provides fault-tolerance and malleability. Satin recursively splits a program into subtasks, and then waits until the subtasks have been completed. At runtime a Satin application can adapt the number of nodes to the degree of parallelism, migrate a computation away from overloaded resources, remove resources with slow communication links, and add new resources to replace resources that have crashed. As such, Satin is one of the few systems that provides transparent programming capabilities in dynamic systems.

(3) SmartSockets: To run a parallel application on multiple distributed resources, it is necessary to establish network connections. In practice, however, a variety of connectivity problems exists that make communication difficult or even impossible, such as firewalls, Network Address Translation (NAT), and multi-homing. It is generally up to the application user to solve such connectivity problems manually.

The SmartSockets library aims to solve connectivity problems automatically, with little or no help from the user. SmartSockets integrates existing and novel solutions, including reverse connection setup, STUN, TCP splicing, and SSH tunneling. SmartSockets creates an overlay network by using a set of interconnected support processes, called *hubs*. Typically, hubs are run on the front-end of a cluster. Using gossiping techniques, the hubs automatically discover to which other hubs they can establish a connection. The power of this approach was demonstrated in a world-wide experiment: in 30 realistic scenarios SmartSockets always was capable of establishing a connection, while traditional sockets only worked in 6 of these [30].

Figure 3 shows an example using three cluster systems. Cluster A is open and allows all connections. Cluster B uses a firewall that only allows outgoing connections. In cluster C only the front-end machine is reachable. No direct communication is possible between the nodes and the outside world. By starting a hub on each of the front-end machines and providing the location of the hub on cluster A to each of them, the hubs will automatically connect as shown in Figure 3. The arrows between the hubs depict the direction in which the connection can be established. Once a connection is made, it can be used in both directions.

The nodes of the clusters can now use the hubs as an overlay network when no direct communication is possible. For example, when a node of cluster B tries to connect to a node of cluster A, a direct connection can immediately be created. A connection from A to B, however, cannot be established directly. In this case, SmartSockets will use the overlay network to exchange control messages between the nodes to reverse the direction of connection setup. As a result, the desired (direct) connection between the nodes of A and B can still be created. The nodes of cluster C are completely unreachable from the other clusters. In this case SmartSockets will create a *virtual connection*, which routes messages over the overlay network.

The basic use of SmartSockets requires manual initialization of the network of hubs. This task, however, is performed automatically and transparently by IbisDeploy, our top-level deployment system, described in the next section.

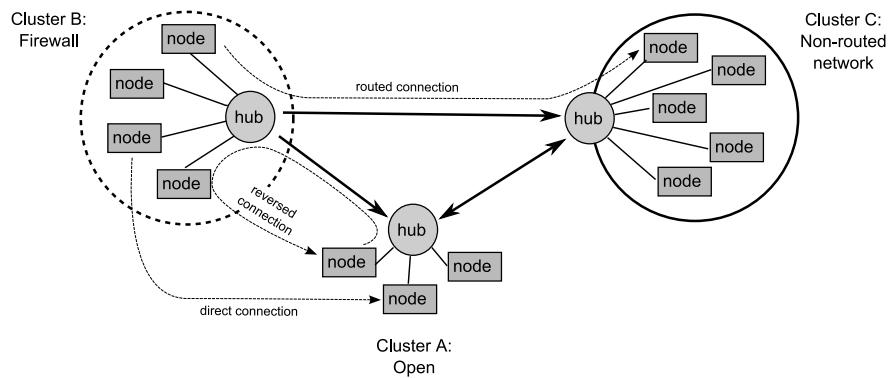


Fig. 3: Example connection setup with SmartSockets.

4.2 The Ibis Distributed Deployment System

The Ibis Distributed Application Deployment System consists of a software stack for deploying and monitoring applications, once they have been written. The software stack consists of (1) the JavaGAT, (2) IbisDeploy, and (3) Zorilla.

(1) The Java Grid Application Toolkit (JavaGAT): Today, distributed system programmers generally have to implement their applications against a grid middleware API that changes frequently, is low-level, unstable, and incomplete [33]. The JavaGAT solves these problems in an integrated manner. JavaGAT offers high-level primitives for developing and running applications, *independent* of the middleware that implements this functionality [51]. The primitives include access to remote data, start remote jobs, support for monitoring, steering, user authentication, resource management, and storing of application-specific data. The JavaGAT uses an extensible architecture, where *adaptors* (plugins) provide access to the different middlewares.

The JavaGAT integrates multiple middleware systems with different and incomplete functionality into a single, consistent system, using a technique called *intelligent dispatching*. This technique dynamically forwards application calls on the JavaGAT API to one or more middleware adaptors that implement the requested functionality. The selection process is done at run-time, and uses policies and heuristics to automatically select the best available middleware, enhancing portability. If an operation fails, the intelligent dispatching feature will automatically select and dispatch the API call to an alternative middleware. This process continues until a middleware successfully performs the requested operation. Although this flexibility comes at the cost of some runtime overhead, compared to the cost of the operations themselves, this is often negligible. For instance, a Globus job submission takes several seconds, while the overhead introduced by the JavaGAT is less than 10 milliseconds. The additional semantics of the high-level API, however, can introduce

some overhead. If a file is copied, for example, the JavaGAT first checks if the destination already exists or is a directory. These extra checks may be costly because they require remote operations. Irrespective of these overheads, JavaGAT is essential to support our 'write-and-go' philosophy: it allows programmers to ignore low-level systems peculiarities and to focus on solving domain-specific problems instead.

The JavaGAT does not provide a new user/key management infrastructure. Instead, its security interface provides generic functionality to store and manage security information such as usernames and passwords. Also, the JavaGAT provides a mechanism to restrict the availability of security information to certain middleware systems or remote machines. Currently, JavaGAT supports many different middleware systems, such as Globus, Unicore, gLite, PBS, SGE, KOALA, SSH, GridSAM, EC2, ProActive, GridFTP, HTTP, SMB/CIFS, and Zorilla.

(2) IbisDeploy: Even though JavaGAT is a major step forward to simplifying application deployment, its API still requires the programmer to think in terms of middleware operations. Therefore, the Ibis Distributed Deployment System provides a

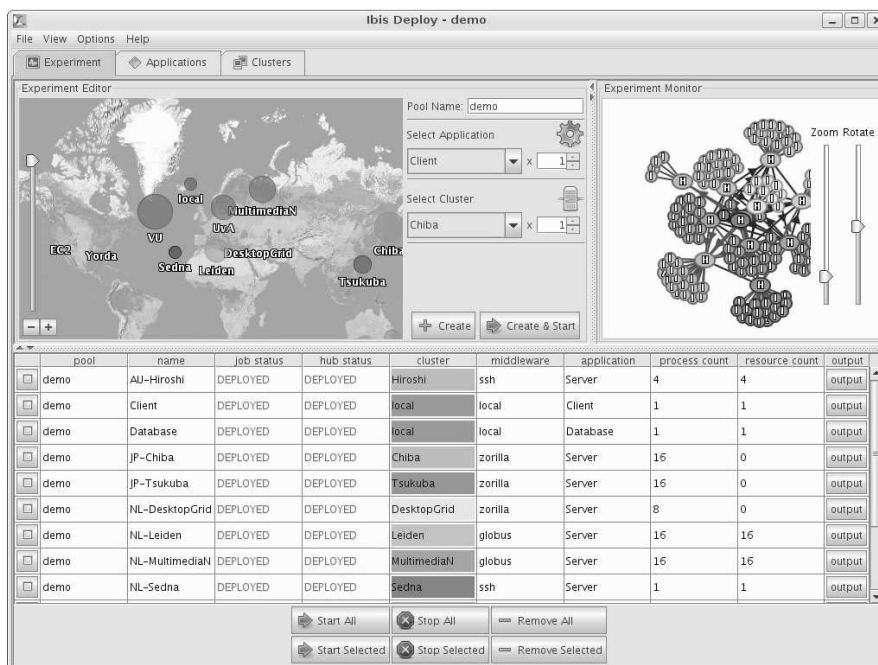


Fig. 4: The IbisDeploy GUI that enables runtime loading of applications and resources (top middle), and keeping track of running processes (bottom half). Top left shows a world map of the locations of available resources; top right shows the SmartSockets network of hubs. See also <http://www.cs.vu.nl/ibis/demos.html>.

simplified and generic API, implemented on top of the JavaGAT, and an additional GUI: the IbisDeploy system.

The IbisDeploy API is a thin layer on top of the JavaGAT API, that initializes JavaGAT in the most commonly used ways, and that lifts combinations of multiple JavaGAT calls to a higher abstraction level. For example, if one wants to run a distributed application written in Ibis, a network of SmartSockets hubs must be started manually. IbisDeploy takes over this task in a fully transparent manner. Also, to run (part of) an Ibis application on a remote machine, one of the necessary steps is to manually upload the actual program code and related libraries to that machine. IbisDeploy transparently deals with such pre-staging (and post-staging) actions as well.

The IbisDeploy GUI (see Figure 4) allows a user to manually load resources and applications at any time. As such, multiple Jungle Computing applications can be started using the same graphical interface. The IbisDeploy GUI also allows the user to add new resources to a running application (by providing contact information such as host address and user credentials), and to pause and resume applications. All run-time settings can be saved and reused in later experiments.

(3) Zorilla: Most existing middleware APIs lack co-scheduling capabilities and do not support fault tolerance and malleability. To overcome these problems, Ibis provides Zorilla, a lightweight peer-to-peer middleware that runs on any Jungle Computing System. In contrast to traditional middleware, Zorilla has no central components and is easy to set up and maintain. Zorilla supports fault tolerance and malleability by implementing all functionality using peer-to-peer techniques. If resources used by an application are removed or fail, Zorilla is capable of automatically finding replacement resources. Zorilla is specifically designed to easily combine resources in multiple administrative domains.

A Zorilla system is made up of a collection of nodes running on all resources, connected by a peer-to-peer network (see Figure 5). Each node in the system is completely independent, and implements all functionality required for a middleware, including the handling of the submission of jobs, running jobs, storing of files, etcetera. Each Zorilla node has a number of local resources. This may simply be the machine it is running on, consisting of one or more processor cores, memory, and data storage. Alternatively, a node may provide access to other resources, for instance to all machines in a cluster. Using the peer-to-peer network, all Zorilla nodes tie together into one big distributed system. Collectively, nodes implement the required global functionality such as resource discovery, scheduling, and distributed data storage, all using peer-to-peer techniques.

To create a resource pool, a Zorilla daemon process must be started on each participating machine. Also, each machine must be given the address of at least one other machine, to set up a connection. Jobs can be submitted to Zorilla using the JavaGAT, or, alternatively, using a command line interface. Zorilla then allocates the requested number of resources and schedules the application, taking user-defined requirements (like memory size) into account. The combination of virtualization and peer-to-peer techniques thus makes it very easy to deploy applications with Zorilla.

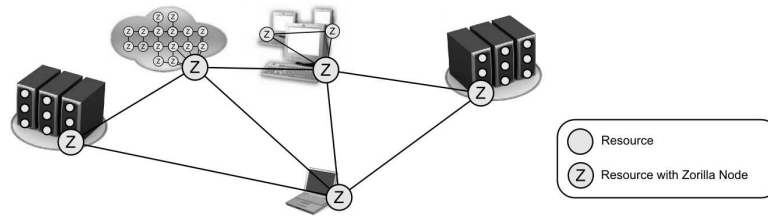


Fig. 5: Example Zorilla peer-to-peer resource-pool on top of a Jungle Computing System consisting of two clusters, a desktop grid, a laptop, and cloud resources (e.g., acquired via Amazon EC2). On the clusters, a Zorilla node is run on the headnode, and Zorilla interacts with the local resources via the local scheduler. On the desktop grid and the cloud, a Zorilla node is running on each resource, since no local middleware capable of scheduling jobs is present on these systems.

4.3 Ibis User Community

Ibis has been used to run a variety of real-life applications like multimedia computing (see Section 6), spectroscopic data processing (by the Dutch Institute for Atomic and Molecular Physics), human brain scan analysis (with the Vrije Universiteit Medical Center), automatic grammar learning, and many others. Also, Ibis has been applied successfully in an implementation of the industry-strength SAT4J SAT-solver. In addition, Ibis has been used by external institutes to build high-level programming systems, such as a workflow engine for astronomy applications in D-grid (Max-Planck-Institute for Astrophysics) and a grid file system (University of Erlangen-Nürnberg), or to enhance existing systems, such as KOALA (Delft University of Technology), ProActive (INRIA), Jylab (University of Patras), and Grid Superscalar (Barcelona Supercomputer Center). Moreover, Ibis has won prizes in international competitions, such as the International Scalable Computing Challenge at CCGrid 2008 and 2010 (for scalability), the International Data Analysis Challenge for Finding Supernovae at IEEE Cluster/Grid 2008 (for speed and fault-tolerance), and the Billion Triples Challenge at the 2008 International Semantic Web Conference (for general innovation).

4.4 Ibis versus the Requirements of Jungle Computing

From the general overview of the Ibis platform it should be clear that our software system adheres to most (if not all) of the requirements of Jungle Computing introduced in Section 3. Resource independence is obtained by relying on JVM virtualization, while the JavaGAT provides us with middleware independence and interoperability. Robust connectivity and globally unique resource naming is taken care of by the SmartSockets library and the Ibis Portability Layer (IPL), respectively. The

need for malleability and system-level fault-tolerance is supported by the resource tracking mechanisms of the Join-Elect-Leave (JEL) model, which is an integral part of the Ibis Portability Layer. Application-level fault-tolerance can be built on top of the system-level fault-tolerance mechanisms provided by the IPL. For a number of programming models, in particular Satin, we have indeed done so in a fully transparent manner. The need for transparent parallelization is also fulfilled through a number of programming models implemented on top of the IPL. Using the Satin model, parallelization is obtained automatically for divide-and-conquer applications. Similarly, using the Jorus model, data parallel multimedia computing applications can be implemented in a fully user transparent manner. Finally, integration with legacy codes and system-level software is achieved through JNI (Java Native Interface) link ups, in the case of system level software through so-called adaptor interfaces (plugins). We will further highlight the linking up with many-core specific compute kernels implemented using CUDA in Section 6.

5 The Need for Jungle Computing in Scientific Practice

As stated in Section 1, the scientific community has automated many daily activities, in particular to *speed up* the generation of results and to *scale up* to problem sizes that better match the research questions at hand. Whether it be in the initial process of data collection, or in later stages including data filtering, analysis, and storage, the desire for speed and scalability can occur in any phase of the scientific process.

In this section we describe a number of urgent and realistic problems occurring in four representative science domains: Multimedia Content Analysis, Semantic Web, Neuroinformatics, and Remote Sensing. Each description focuses on the societal significance of the domain, the fundamental research questions, and the unavoidable need for applying Jungle Computing Systems.

5.1 *Multimedia Content Analysis*

Multimedia Content Analysis (MMCA) considers all aspects of the automated extraction of knowledge from multimedia data sets [7, 45]. MMCA applications (both real-time and off-line) are rapidly gaining importance along with recent deployment of publicly accessible digital TV archives, and surveillance cameras in public locations. In a few years, MMCA will be a problem of phenomenal proportions, as digital video may produce high data rates, and multimedia archives steadily run into Petabytes of storage. For example, the analysis of the TRECVID data set [42], consisting of 184 hours of video, was estimated to take over 10 years on a fast sequential computer. While enormous in itself, this is insignificant compared to the 700,000 hours of TV data archived by the Dutch Institute for Sound and Vision. Moreover, distributed sets of surveillance cameras generate even larger quantities of data.

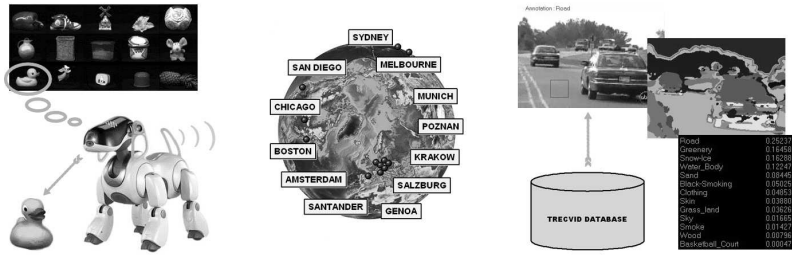


Fig. 6: Real-time (left) and off-line (right) distributed multimedia computing. The real-time application constitutes a visual object recognition task, in which video frames obtained from a robot camera are processed on a set of compute clusters. Based on the calculated scene descriptions, a database of learned objects is searched. Upon recognition the robot reacts accordingly. For this system we obtained a 'most visionary research award' at AAAI 2007. A video presentation is available at <http://www.cs.vu.nl/~fjseins/aibo.html>. The off-line application constitutes our TRECVID system, in which low-level semantic concepts (e.g., 'sky', 'road', 'greenery') are extracted and combined into high-level semantic concepts (e.g., 'cars driving on a highway'), using the same set of compute clusters. For this system we obtained a 'best technical demo award' at ACM Multimedia 2005.

Clearly, for emerging MMCA problems there is an urgent need for speed and scalability. Importantly, there is overwhelming evidence that large-scale distributed supercomputing indeed can push forward the state-of-the-art in MMCA. For example, in [42] we have shown that a distributed system involving hundreds of massively communicating resources covering the entire globe indeed can bring efficient solutions for real-time and off-line problems (see Figure 6). Notably, our Ibis-based solution to the real-time problem of Figure 6 has won First Prize in the International Scalable Computing Challenge at CCGrid in 2008.

5.2 Semantic Web

The Semantic Web [23, 49] is a groundbreaking development of the World Wide Web in which the semantics of information is defined. Through these semantics machines can 'understand' the Web, allowing querying and reasoning over Web information gathered from different sources. The Semantic Web is based on specifications that provide formal descriptions of concepts, terms, and relationships within a given knowledge domain. Examples include annotated medical datasets containing e.g. gene sequence information (linkedlifedata.com), and structured information derived from Wikipedia (dbpedia.org).

Even though the Semantic Web domain is still in its infancy, it already faces problems of staggering proportions [17]. Today, the field is dealing with huge distributed

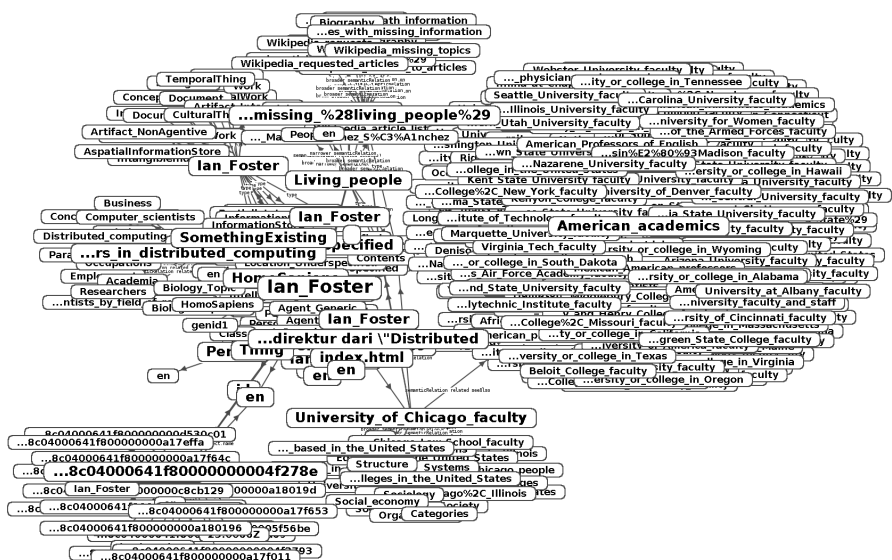


Fig. 7: Visualization of relationships between concepts and terms extracted through automatic reasoning. As the full dataset is far too large, this example visualization focuses on one term only. This is a feat in itself, as the filtering out of such data from the full dataset is a complex search problem. Notably, our prize-winning reasoner is 60 times faster and deals with 10 times more data than any existing approach.

repositories containing billions of facts and relations (see also Figure 7) — with an expected exponential growth in the years to come. As current Semantic Web reasoning systems do not scale to the requirements of emerging applications, it has been acknowledged that there is an urgent need for a distributed platform for massive reasoning that will remove the speed and scalability barriers [17]. Notably, our preliminary work in this direction has resulted in prize-winning contributions to the Billion Triples Challenge at the International Semantic Web Conference in 2008 [2], and the International Scalable Computing Challenge at CCGrid in 2010 [48].

5.3 Neuroinformatics

Neuroinformatics encompasses the analysis of experimental neuroscience data for improving existing theories of brain function and nervous system growth. It is well known that activity dynamics in neuronal networks depend largely on the pattern of synaptic connections between neurons [12]. It is not well understood, however, how neuronal morphology and the local processes of neurite outgrowth and synapse formation influence global connectivity. To investigate these issues, there is a need

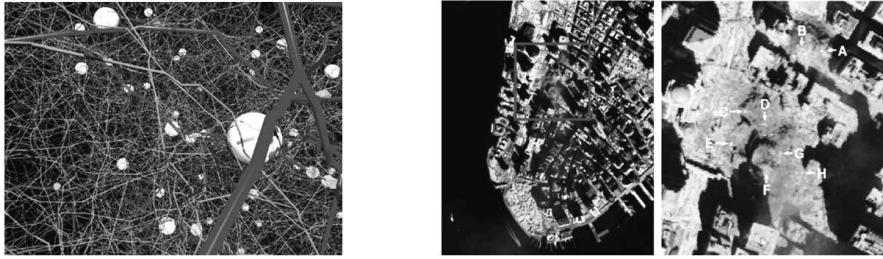


Fig. 8: Left: NETMORPH-generated network, with cell bodies, axons, dendrites and synaptic connections embedded in 3D space. Right: AVIRIS hyperspectral image data, with location of fires in World Trade Center.

for simulations that can generate large neuronal networks with realistic neuronal morphologies.

Due to the high computational complexity, existing frameworks (such as NETMORPH [27]) can simulate networks of up to a few hundred neurons only (Figure 8, left). With the human brain having an estimated 100 billion neurons, a vast scaling up of the simulations is urgent. While the neuroinformatics domain is not new to the use of supercomputer systems (e.g. the Blue Brain Project [21]), speed and scale requirements — as well as algorithmic properties — dictate distributed supercomputing at an unprecedented scale. Our preliminary investigation of the NETMORPH system has shown significant similarities with a specific subset of N-body problems that require adaptive runtime employment of compute resources. We have ample experience with such problems [41, 60], which require many of the advanced capabilities of the Ibis system.

5.4 Remote Sensing

Remotely sensed hyperspectral imaging is a technique that generates hundreds of images corresponding to different wavelengths channels for the same area on the surface of the Earth [20]. For example, NASA is continuously gathering image data with satellites such as the Jet Propulsion Laboratory’s Airborne Visible-Infrared Imaging Spectrometer (AVIRIS [22]). The resulting hyperspectral data cubes consist of high-dimensional pixel vectors representing spectral signatures that uniquely characterize the underlying objects [8]. Processing of these cubes is highly desired in many application domains, including environmental modeling and risk/hazard prediction (Figure 8, right).

With emerging instruments generating in the order of 1 Gbit/s (e.g., ESA’s FLEX [39]) data sets applied in real applications easily approach the petascale range. Given the huge computational demands, parallel and distributed solutions

are essential, at all levels of granularity. As a result, in the last decade the use of compute clusters for applications in remote sensing has become commonplace [38]. These approaches are proven beneficial for a diversity of problems, including target detection and classification [35], and automatic spectral unmixing and endmember extraction [37]. Depending on the complexity and dimensionality of the analyzed scene, however, the computational demands of many remote sensing methodologies still limit their use in time-critical applications. For this reason, there is an emerging trend in mapping remote sensing functionality onto multi- and many-core hardware, and combining the resulting compute kernels with existing solutions for clusters and distributed systems [36].

5.5 A Generalized View

The science domains described above are not unique in their needs for speed and scalability. These domains are simply the ones for which we have gained experience over the years. Certainly, and as clearly shown in [3], the list of domains that we could have included here is virtually endless.

Importantly, however, the set of described domains covers a wide range of *application types*, with some being time-constrained and compute-intensive, and others being off-line and data-intensive. Also, for all of these domains Jungle Computing solutions already exist today (at least to a certain extent) — with each using a variety of distributed computing systems (even at a world-wide scale), and some including the use of many-core hardware. As shown in Section 6, some of these solutions are implemented purely in Ibis, while others constitute a ‘mixed-language’ solution with legacy codes and specialized compute kernels being integrated with Ibis software. As stated, many of these Ibis-implemented solutions have won prizes and awards at international venues, each for a different reason: speed, scalability, fault-tolerance, and general innovation. With our ultimate goal of developing transparent and efficient tools for scientific domain experts in mind, it is relevant to note that several of these results have been obtained with little or no help from the Ibis team.

This brings us to the reasons for the *unavoidable need* for using Jungle Computing Systems in these and other domains, as claimed in this chapter. While these reasons are manifold, we will only state the most fundamental ones here. First, in many cases research scientists need to acquire a number of compute nodes that can not be provided by a single system alone — either because the problem at hand is too big, or because other applications are being run on part of the available hardware. In these cases, concurrently acquiring nodes on multiple systems often is the only route to success. In other cases, calculations need to be performed on distributed data sets that can not be moved — either because of size limitations, or due to reasons of privacy, copyright, security, etcetera. In these cases, it is essential to (transparently) move the calculations to where the data is, instead of vice versa. Furthermore, because many scientists have to rely on legacy codes or compute kernels for special-

purpose hardware, parts of the processing pipeline may only run on a limited set of available machines. In case the number of such specialized codes becomes large, acquiring resources from many different resources simply is unavoidable.

It is important to realize that the use of a variety of distributed resources (certainly at a world-wide scale) is *not* an active desire or end-goal of any domain expert. For all of the above (and other) reasons, research scientists today are simply being *forced* into using Jungle Computing Systems. It is up to the field of high-performance and distributed computing to understand the fundamental research questions underlying this problem, to develop the fundamental methodologies solving each of these research questions, and to combine these methodologies into efficient and transparent programming models and tools for end-users.

6 Jungle Computing Experiments

In this section we describe a number of experiments that illustrate the functionality and performance of the Ibis system. We focus on the domain of Multimedia Content Analysis, introduced in Section 5.1. Our discussion starts with a description of an Ibis-implemented programming model, called Jorus, which is specifically targeted towards researchers in the MMCA domain [42].

For the bulk of the experiments we use the Distributed ASCI Supercomputer 3 (DAS-3, www.cs.vu.nl/das3), a five cluster/272-dual node distributed system located at four universities in The Netherlands. The clusters are largely homogeneous, but there are differences in the number of cores per machine, the clock frequency, and the internal network. In addition, we use a small GPU-cluster, called Lisa, located at SARA (Stichting Academisch Rekencentrum Amsterdam). Although the traditional part of the Lisa cluster is much larger, the system currently has a total of 6 Quad-core Intel Xeon 2.50GHz nodes available, each of which is equipped with two Nvidia Tesla M1060 graphics adaptors with 240 cores and 4 GBytes of device memory. Next to DAS-3 and Lisa, we use additional clusters in Chicago (USA), Chiba and Tsukuba (InTrigger, Japan), and Sydney (Australia), an Amazon EC2 Cloud system (USA, East Region), as well as a desktop grid and a single stand-alone machine (both Amsterdam, The Netherlands). Together, this set of machines constitutes a real-world Jungle Computing System as defined earlier. Most of the experiments described below are supported by a video presentation, which is available at <http://www.cs.vu.nl/ibis/demos.html>.

6.1 High-Performance Distributed Multimedia Analysis with Jorus

The Jorus programming system is an Ibis-implemented *user transparent* parallelization tool for the MMCA domain. Jorus is the next generation implementation of our library-based Parallel-Horus system [42], which was implemented in C++ and MPI.

Jorus and Parallel-Horus allow programmers to implement *data parallel* multimedia applications as fully sequential programs. Apart from the obvious differences between the Java and C++ languages, the Jorus and Parallel-Horus APIs are identical to that of a popular *sequential* programming system, called Horus [26]. Similar to other frameworks [34], Horus recognizes that a small set of *algorithmic patterns* can be identified that covers the bulk of all commonly applied multimedia computing functionality.

Jorus and Parallel-Horus include patterns for functionality such as unary and binary pixel operations, global reduction, neighborhood operation, generalized convolution, and geometric transformations (e.g. rotation, scaling). Recent developments include patterns for operations on large datasets, as well as patterns on increasingly important derived data structures, such as feature vectors. For reasons of efficiency, all Jorus and Parallel-Horus operations are capable of adapting to the performance characteristics of the cluster computer at hand, i.e. by being flexible in the partitioning of data structures. Moreover, it was realized that it is not sufficient to consider parallelization of library operations *in isolation*. Therefore, the programming systems incorporate a run-time approach for communication minimization (called *lazy parallelization*) that automatically parallelizes a fully sequential program at run-time by inserting communication primitives and additional memory management operations whenever necessary [43].

Earlier results obtained with Parallel-Horus for realistic multimedia applications have shown the feasibility of the applied approach, with data parallel performance consistently being found to be optimal with respect to the abstraction level of message passing programs [42]. Notably, Parallel-Horus was applied in earlier NIST TRECVID benchmark evaluations for content-based video retrieval, and played a crucial role in achieving top-ranking results in a field of strong international competitors [42, 45]. Moreover, and as shown in our evaluation below, extensions to Jorus and Parallel-Horus that allow for services-based distributed multimedia computing, have been applied successfully in large-scale distributed systems, involving hundreds of massively communicating compute resources covering the entire globe [42]. Finally, while the current Jorus implementation realizes data parallel execution on cluster systems in a fully user transparent manner, we are also working on a cluster-implementation that results in combined data and task parallel execution [50].

6.2 Experiment 1: Fine-grained Parallel Computing

To start our evaluation of Ibis for Jungle Computing, we first focus on Ibis communication on a single traditional cluster system. To be of any significance for Jungle Computing applications, it is essential for Ibis' communication performance to compare well to that of the MPI message passing library — the de facto standard for high-performance cluster computing applications. Therefore, in this first experiment we will focus on fine-grained data-parallel image and video analysis, implemented

using our Jorus programming model. For comparison, we also report results obtained for Parallel-Horus (in C++/MPI) [42].

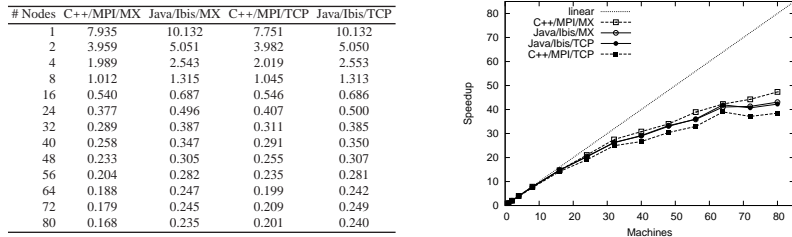
In particular we investigate the data-parallel analysis of a single video frame in a typical MMCA application (as also shown in the left half of Figure 6). The application implements an advanced object recognition algorithm developed in the Dutch MultimediaN project [42]. At runtime, so-called *feature vectors* are extracted from the video data, each describing local properties like color and shape. The analysis of a single video frame is a data-parallel task executed on a cluster. When using multiple clusters, data-parallel calculations over consecutive frames are executed concurrently in a task-parallel manner.

The Jorus implementation intentionally mimics the original Parallel-Horus version as close as possible. Hence, Jorus implements several collective communication operations, such as scatter, gather, and all-reduce. Other application specific communication steps, such as the exchange of non-local image data between nodes (known as *BorderExchange*) are implemented in a manner resembling point-to-point communication. More importantly, the Ibis run-time environment has been set up for *closed-world* execution, meaning that the number of compute nodes is fixed for the duration of the application run. This approach voids all of Ibis' fault-tolerance and malleability capabilities, but it shows that *Ibis can be used easily to mimic any MPI-style application*.

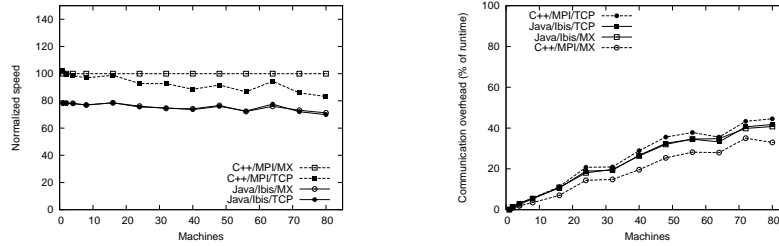
By default, Ibis uses TCP for communication, but the user can indicate which communication protocol to use. In our experiments the MX (Myrinet Express) protocol is available for the DAS-3 Myri-10G high-speed interconnects. In the following we will therefore report results for four different versions of the multimedia application using a single physical network: two versions in Java/Ibis (one communicating over MX and one over TCP) and two versions in C++/MPI (again, one for each protocol). Of these four versions, the C++/MPI/MX version is expected to be the fastest, as C++ is perceived to be generally faster than Java and MX is designed specifically for communication over Myri-10G.

Figure 9(a) presents the results for the DAS-3 cluster at the Vrije Universiteit, obtained for a video frame of size 1024×768 pixels. The sequential Java version is about 20% slower than the sequential C++ version. This performance drop is well within acceptable limits for a 'compile-once, run everywhere' application executing inside a virtual machine. Also, MX does not significantly outperform TCP. Clearly, the communication patterns applied in our application do not specifically favor MX. More importantly, the two Ibis versions are equally fast and show similar speedup characteristics compared to their MPI-based counterparts.

Further evidence of the feasibility of Ibis for parallel computing is shown in Figure 9(b). The graph on the left shows the normalized speed of three versions of the application compared to the fastest C++/MPI/MX version. It shows that the relative drop in performance is rather stable at 20 to 25%, which is attributed to JVM overhead. The graph on the right presents the cost of communication relative to the overall execution time. Clearly, the relative parallelization overheads of Ibis and MPI are almost identical. These are important results, given the increased flexibility and much wider applicability of the Ibis system.



(a) Performance and speedup characteristics of all application versions.



(b) Normalized speed and communication overhead of all application versions.

Fig. 9: Results obtained on DAS-3 cluster at Vrije Universiteit, Amsterdam.

6.3 Experiment 2: User Transparent MMCA on GPU-clusters

Essentially, Jorus extends the original sequential Horus library by introducing a thin layer right in the heart of the small set of algorithmic patterns, as shown in Figure 10. This layer uses the IPL to communicate image data and other structures among the different nodes in a cluster. In the most common case, a digital image is *scattered* throughout the parallel system, such that each compute node is left with a *partial image*. Apart from the need for additional pre- and post-communication steps (such as the common case of *border handling* in convolution operations), the *sequential compute kernels* as also available in the original Horus system are now applied to each partial image.

From a software engineering perspective, the fact that the IPL extensions 'touch' the sequential implementation of the algorithmic patterns in such a minimal way provides Jorus with the important properties of sustainability and easy extensibility. In the process of extending Jorus for GPU-based execution, we have obtained a similar minimal level of intrusiveness: we left the thin communication layer as it is, and introduced CUDA-based alternatives to the sequential compute kernels that implement the algorithmic patterns (see bottom half of Figure 10). In this manner, Jorus and CUDA are able to work in concert, allowing the use of multiple GPUs on the same node, and on multiple nodes simultaneously, simply by creating one Jorus process for each GPU. In other words, with this approach we obtain a system

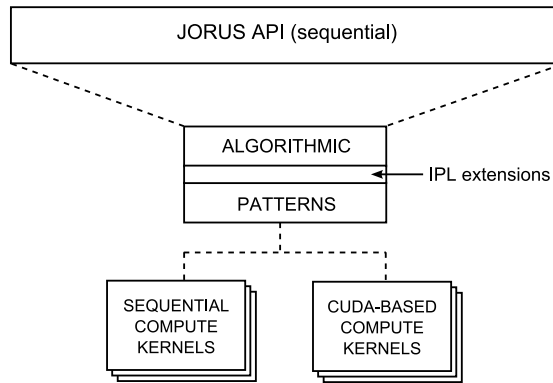


Fig. 10: General overview of the Jorus system and its CUDA-based extensions.

that can execute sequential Jorus applications in data parallel fashion, while exploiting the power of GPU hardware. The details of the CUDA-implemented compute kernels are beyond the scope of this chapter, and are discussed further in [52].

Our second experiment comprises the computationally demanding problem of *line detection* in images and video streams [19]. Although not identical to our object recognition problem discussed above, the set of algorithmic patterns applied in the Jorus implementation is quite similar. We have executed this GPU-enabled application on the Lisa cluster. In our experiments we use many different configurations, each of which is denoted differently by the number of nodes and CPUs/GPUs used. For example, measurements involving one compute node and one Jorus process are denoted by 1x1. Likewise, 4x2 means that 4 nodes are used with 2 Jorus processes executing on each node. For the CUDA-based executions, the latter case implies the concurrent use of 8 GPUs.

Figure 11(a) shows the total runtimes for several configurations using either CPUs or GPUs for the execution of the original and CUDA-implemented compute kernels. The presented execution times include the inherently sequential part of the application, which consists mainly of reading and writing the input and output images. For CPU-only execution, the execution time reduces linearly; using 8 CPUs gives a speedup of 7.9. These results are entirely in line with earlier speedup characteristics reported in [44] for much larger cluster systems.

The GPU-extensions to the Jorus system show a dramatic performance improvement in comparison with the original version. Even in the 1x1 case, the total execution time is reduced by a factor of 61.2. The speedup gained from executing on a GPU cluster compared to a traditional cluster, clearly demonstrates why traditional clusters are now being extended with GPUs as accelerators. As shown in Figure 11(b), our application executing on 4 nodes with 2 Jorus processes per node, experiences a speedup of 387 with GPU-extensions. Notably, when using the Jorus programming model, these speedup results are obtained without requiring *any* parallelization effort from the application programmer.

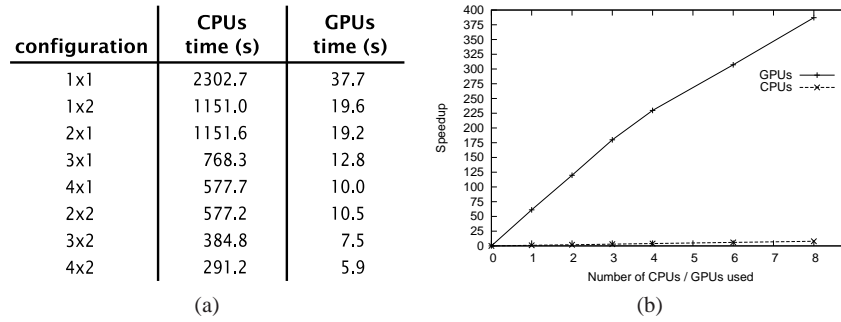


Fig. 11: (a) Runtimes in seconds for different configurations in the number of CPUs and GPUs; (b) User-perceived speedup compared to single CPU execution. Results obtained on Lisa cluster, SARA, Amsterdam.

6.4 Experiment 3: Jungle Computing at a World-Wide Scale

After having discussed some the capabilities and performance characteristics of the Ibis system for traditional cluster systems as well as emerging GPU-clusters, we will now turn our attention to the use of Ibis for world-wide execution on a large variety of computing resources. For this purpose, we reconsider our object recognition problem of Section 6.2.

As stated, when using multiple distributed resources, with Jorus it is possible to concurrently perform multiple data-parallel calculations over consecutive video frames in a task-parallel manner. This is achieved by wrapping the data parallel analysis in a *multimedia server* implementation. At run-time, client applications can then upload an image or video frame to such a server, and receive back a recognition result. In case multiple servers are available, a client can use these simultaneously for subsequent image frames, in effect resulting in task-parallel employment of data-parallel services.

As shown in our demonstration video (see www.cs.vu.nl/ibis/demos.html), we use IbisDeploy to start a client and an associated database of learned objects on a local machine, and to deploy four data-parallel multimedia servers — each on a different DAS-3 cluster (using 64 machines in total). All code is implemented in Java and Ibis, and compiled on the local machine. No application codes are initially installed on any other machine.

Our distributed application shows the simultaneous use of multiple Ibis environments. Whilst the data-parallel execution runs in a closed-world setting, the distributed extensions are set up for open-world execution to allow resources to be added and removed at run-time. For this application the additional use of resources indeed is beneficial: where the use of a single multimedia server results in a client-side processing rate of approximately 1.2 frames per second, the simultaneous use of

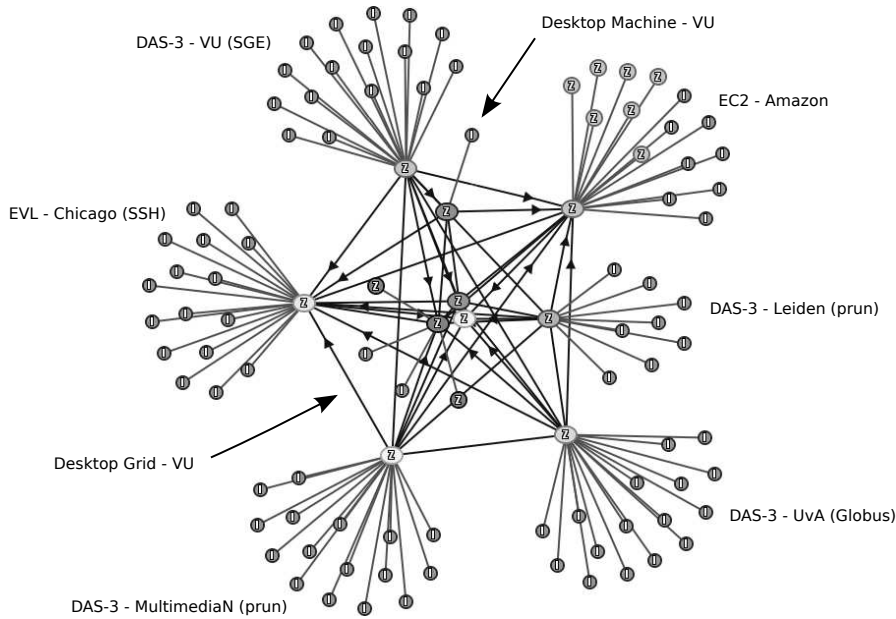


Fig. 12: Visualization of resources used in a world-wide Jungle Computing experiment, showing all nodes and the SmartSockets overlay network between these. Nodes marked *Z* represent Zorilla nodes running on either a frontend machine, or on a resource itself. Nodes marked *I* represent instances of a running (Ibis) application.

2 and 4 clusters leads to linear speedups at the client side with 2.5 and 5 frames/sec respectively.

We continue the experiment by employing several additional clusters, an Amazon EC2 cloud, a local desktop grid, and a local stand-alone machine. With this world-wide set of machines we now use a variety of middlewares simultaneously (i.e., Globus, Zorilla, and SSH) from within a single application. Although not included in the demonstration, at a lower video resolution the maximum obtained frame-rate is limited only by the speed of the camera, meaning that (soft) real-time multimedia computing at a world-wide scale has become a reality.

When running at this world-wide scale a variety of connectivity problems is automatically circumvented by SmartSockets. As almost all of the applied resources have more than one IP address, SmartSockets automatically selects the appropriate address. Also, SmartSockets automatically creates SSH tunnels to connect with the clusters in Japan and Australia. These systems would be unreachable otherwise. Finally, the SmartSockets network of hubs avoids further problems due to firewalls and NATs. In case we would remove the hub network, we only would have access to those machines that allow direct connections *to* and *from* our client application. As our client is behind a firewall, two-way connectivity is possible only within the same

administrative domain, which includes the local desktop grid, the stand-alone machine, and one of the DAS-3 clusters. Clearly, the absence of SmartSockets would by-and-large reduce our world-wide distributed system to a small set of local machines. A visualization of a SmartSockets overlay network is shown in Figure 12.

To illustrate Ibis' fault-tolerance mechanisms, the video also shows an experiment where an entire multimedia server crashes. The Ibis resource tracking system notices the crash, and signals this event to other parts of the application. After some time, the client is notified, to the effect that the crashed server is removed from the list of available servers. The application continues to run, as the client keeps on forwarding video frames to all remaining servers.

The demonstration also shows the use of the multimedia servers from a smartphone running the Android operating system. For this platform we have written a separate Ibis-based client, which can upload pictures taken with the phone's camera and receive back a recognition result. Running the full application on the smartphone itself is not possible due to CPU and memory limitations. Based on a stripped down, yet very inaccurate, version that *does* run on the smartphone we estimate that recognition for 1024×768 images would take well over 20 minutes. In contrast, when the smartphone uploads a picture to one of the multimedia servers, it obtains a result in about 3 seconds. This result clearly shows the potential of Ibis to open up the field of mobile computing for compute-intensive applications. Using IbisDeploy, it is even possible to deploy the entire distributed application as described from the smartphone itself.

7 Conclusions and Future Work

In this chapter we have argued that, while the need for speed and scalability in everyday scientific practice is omnipresent and increasing, the resources employed by end-users are often more diverse than those contained in a single cluster, grid, or cloud system. In many realistic scientific research areas, domain experts are being forced into concurrent use of multiple clusters, grids, clouds, desktop grids, stand-alone machines, and more. Writing applications for such *Jungle Computing Systems* has become increasingly difficult, in particular with the integration of many-core hardware technologies.

The aim of the Ibis platform is to drastically simplify the programming and deployment of Jungle Computing applications. To achieve this, Ibis integrates solutions to many of the fundamental problems of Jungle Computing in a single modular programming and deployment system, written entirely in Java. Ibis has been used for many real-world Jungle Computing applications, and has won awards and prizes in very diverse international competitions.

Despite the successes, and the fact that — to our knowledge — Ibis is the only integrated system that offers an efficient and transparent solution for Jungle Computing, further progress is urgent for Ibis to become a viable programming system for everyday scientific practice. One of the foremost questions to be dealt with is

whether it is possible to define a set of fundamental building blocks that can describe *any* Jungle Computing application. Such building blocks can be used to express both generic programming models (e.g., pipelining, divide-and-conquer, MapReduce, SPMD), as well as domain-specific models (e.g. the Jorus model described earlier). A further question is whether all of these models indeed can yield efficient execution on various Jungle Computing Systems. In relation to this is the question whether it is possible to define generic computational patterns that can be re-used to express a variety of domain-specific programming models. The availability of such generic patterns would significantly enhance the development of new programming models for unexplored scientific domains.

As we have shown in Section 6.3, multiple kernels with identical functionality but targeted at different platforms (referred to as *equi-kernels*) often are available. Such kernels are all useful, e.g. due to different scalability characteristics, or ad-hoc hardware availability. Therefore, an important question is how to transparently integrate (multiple) domain-specific kernels with Jungle Computing programming models and applications. Moreover, how do we transparently decide to schedule a specific kernel when multiple equi-kernels can be executed on various resources? Initially, it would be sufficient to apply random or heuristic-based approaches. For improved performance, however, further solutions must be investigated, including those that take into account the potential benefits of coalescing multiple subsequent kernels, and scheduling these as a single kernel.

Mapping kernels to resources is a dynamic problem. This is because resources may be added or removed, and the computational requirements of kernels may fluctuate over time. Moreover, the mapping may have to take into account optimization under multiple, possibly conflicting, objectives (e.g., speed, productivity, financial costs, energy use). Hence, a further research question is to what extent it is possible to provide run-time support for transparent and dynamic re-mapping and migration of compute kernels in Jungle Computing Systems. Also, what basic metrics do we need for making self-optimization decisions, and how can we apply these using existing theories of multi-objective optimization?

Despite the need for solutions to all of these (and other) fundamental research questions, the Ibis system was shown to adhere to most (if not all) of the necessary requirements of Jungle Computing introduced in Section 3. As a result, we conclude that Jungle Computing is not merely a visionary concept; with Ibis we already deploy Jungle Computing applications on a world-wide scale virtually every day. Anyone can do the same: the Ibis platform is fully open source and can be downloaded for free from <http://www.cs.vu.nl/ibis>.

References

1. Abramson, D., Sasic, R., Giddy, J., Hall, B.: Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. In: Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing (HPDC'95), pp. 112–121. Pentagon City, USA (1995)

2. Anadiotis, G., Kotoulas, S., Oren, E., Siebes, R., van Harmelen, F., Drost, N., Kemp, R., Maassen, J., Seinstra, F., Bal, H.: MaRVIN: A Distributed Platform for Massive RDF Inference. In: Semantic Web Challenge 2008, held in conjunction with the 7th International Semantic Web Conference (ISWC 2008). Karlsruhe, Germany (2008)
3. Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K.: A View of the Parallel Computing Landscape. *Communications of the ACM* **52**(10), 56–67 (2009)
4. Bal, H., Maassen, J., van Nieuwpoort, R., Drost, N., Kemp, R., van Kessel, T., Palmer, N., Wrzesińska, G., Kielmann, T., van Reeuwijk, K., Seinstra, F., Jacobs, C., Verstoep, K.: Real-World Distributed Computing with Ibis. *IEEE Computer* (2010). (to appear)
5. Butler, D.: The Petaflop Challenge. *Nature* **448**, 6–7 (2007)
6. Carley, K.: Organizational Change and the Digital Economy: A Computational Organization Science Perspective. In: E. Brynjolfsson, B. Kahin (eds.) *Understanding the Digital Economy: Data, Tools, Research*, pp. 325–351. MIT Press (2000)
7. Carneiro, G., Chan, A., Moreno, P., Vasconcelos, N.: Supervised Learning of Semantic Classes for Image Annotation and Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **29**(3), 394–410 (2007)
8. Chang, C.I.: *Hyperspectral Data Exploitation: Theory and Applications*. John Wiley & Sons, New York (2007)
9. D. Kranzlmüller: Towards a Sustainable Federated Grid Infrastructure for Science (2009). Keynote Talk, Sixth High-Performance Grid Computing Workshop (HPGC'08), Rome, Italy
10. Denis, A., Aumage, O., Hofman, R., Verstoep, K., Kielmann, T., Bal, H.: Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems. In: *Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC'04)*, pp. 97–106. Honolulu, HI, USA (2004)
11. Dijkstra, E.: *On the Phenomenon of Scientific Disciplines* (1986). Unpublished Manuscript EWD988; E.W. Dijkstra Archive
12. Douglas, R., Martin, K.: Neuronal Circuits in the Neocortex. *Annual Review of Neuroscience* **27**, 419–451 (2004)
13. Drost, N., van Nieuwpoort, R., Maassen, J., Seinstra, F., Bal, H.: JEL: Unified Resource Tracking for Parallel and Distributed Applications. *Concurrency and Computation: Practice & Experience* (2010). (to appear)
14. Editorial: The Importance of Technological Advances. *Nature Cell Biology* **2**, E37 (2000)
15. Editorial: Cloud Computing: Clash of the Clouds (2009). *The Economist*
16. F. Gagliardi: Grid and Cloud Computing: Opportunities and Challenges for e-Science (2008). Keynote Speech, International Symposium on Grid Computing 2008 (ISCG 2008), Taipei, Taiwan
17. Fensel, D., van Harmelen, F., Andersson, B., Brennan, P., Cunningham, H., Valle, E.D., Fischer, F., Zhisheng, H., Kiryakov, A., Lee, T.I., I. Schooler, Tresp, V., Wesner, S., Witbrock, M., Ning, Z.: Towards LarKC: A Platform for Web-Scale Reasoning. In: *Proceedings of the Second International Conference on Semantic Computing (ICSC 2008)*, pp. 524–529. Santa Clara, CA, USA (2008)
18. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High-Performance Computing Applications* **15**(3), 200–222 (2001)
19. Geusebroek, J., Smeulders, A., Geerts, H.: A Minimum Cost Approach for Segmenting Networks of Lines. *International Journal of Computer Vision* **43**(2), 99–111 (2001)
20. Goetz, A., Vane, G., Solomon, J., Rock, B.: Imaging Spectrometry for Earth Remote Sensing. *Science* **228**, 1147–1153 (1985)
21. Graham-Rowe, D.: Mission to Build a Simulated Brain Begins. *New Scientist* (2005)
22. Green, R., Eastwood, M., Sarture, C., Chrien, T., Aronsson, M., Chippendale, B., Faust, J., Pavri, B., Chovit, C., Solis, M., Olah, M.: Imaging Spectroscopy and the Airborne Visible/Infrared Imaging Spectrometer (AVIRIS). *Remote Sensing of Environment* **65**(3), 227–248 (1998)

23. Hendler, J., Shadbolt, N., Hall, W., Berners-Lee, T., Weitzner, D.: Web Science: An Interdisciplinary Approach to Understanding the Web. *Communications of the ACM* **51**(7), 60–69 (2008)
24. Hey, T.: The Social Grid (2007). Keynote Talk, OGF20 2007, Manchester, UK
25. Khan, J., Wierzbicki, A.: Guest Editor's Introduction; Foundation of Peer-to-Peer Computing. *Computer Communications* **31**(2), 187–189 (2008)
26. Koelma, D., Poll, E., Seinstra, F.: Horus C++ Reference. Tech. rep., University of Amsterdam, The Netherlands (2002)
27. Koene, R., Tijms, B., van Hees, P., Postma, F., de Ridder, A., Ramakers, G., van Pelt, J., van Ooyen, A.: NETMORPH: A Framework for the Stochastic Generation of Large Scale Neuronal Networks with Realistic Neuron Morphologies. *Neuroinformatics* **7**(3), 195–210 (2009)
28. Lu, P., Oki, H., Frey, C., Chamitoff, G., Chiao, L., E.M. Fincke C.M. Foale, S.M., Jr., W.M., Tani, D., Whitson, P., Williams, J., Meyer, W., Sicker, R., Au, B., Christiansen, M., Schofield, A., Weitz, D.: Order-of-magnitude Performance Increases in GPU-accelerated Correlation of Images from the International Space Station. *Journal of Real-Time Image Processing* (2009)
29. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E., Tao, J., Zhao, Y.: Scientific Workflow Management and the KEPLER System. *Concurrency and Computation: Practice & Experience* **18**(10), 1039–1065 (2005)
30. Maassen, J., Bal, H.: 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. Monterey, USA (2007)
31. Manual: Advanced Micro Devices Corporation (AMD). AMD Stream Computing User Guide, Revision 1.1 (2008)
32. Manual: NVIDIA CUDA Complete Unified Device Architecture Programming Guide, v2.0 (2008)
33. Medeiros, R., Cirne, W., Brasileiro, F., Sauv e, J.: 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*, pp. 18–24. Phoenix, AZ, USA (2003)
34. Morrow, P., Crookes, D., Brown, J., McAleese, G., Roantree, D., Spence, I.: Efficient Implementation of a Portable Parallel Programming Model for Image Processing. *Concurrency: Practice & Experience* **11**, 671–685 (1999)
35. Paz, A., Plaza, A., Plaza, J.: Comparative Analysis of Different Implementations of a Parallel Algorithm for Automatic Target Detection and Classification of Hyperspectral Images. In: *Proceedings of SPIE Optics and Photonics - Satellite Data Compression, Communication, and Processing V*. San Diego, CA, USA (2009)
36. Plaza, A.: Recent Developments and Future Directions in Parallel Processing of Remotely Sensed Hyperspectral Images. In: *Proceedings of the 6th International Symposium on Image and Signal Processing and Analysis*, pp. 626–631. Salzburg, Austria (2009)
37. Plaza, A., Plaza, J., Paz, A.: Parallel Heterogeneous CBIR System for Efficient Hyperspectral Image Retrieval Using Spectral Mixture Analysis. *Concurrency and Computation: Practice & Experience* **22**(9), 1138–1159 (2010)
38. Plaza, A., Valencia, D., Plaza, J., Martinez, P.: Commodity Cluster-based Parallel Processing of Hyperspectral Imagery. *Journal of Parallel and Distributed Computing* **66**(3), 345–358 (2006)
39. Rasher, U., Gioli, B., Miglietta, F.: FLEX - Fluorescence Explorer: A Remote Sensing Approach to Quantify Spatio-Temporal Variations of Photosynthetic Efficiency from Space. In: J. Allen, et al. (eds.) *Photosynthesis. Energy from the Sun: 14th International Congress on Photosynthesis*, pp. 1387–1390. Springer (2008)
40. Reilly, M.: When Multicore isn't enough: Trends and the Future for Multi-Multicore Systems. In: *Proceedings of the Twelfth Annual Workshop on High-Performance Embedded Computing (HPEC 2008)*. Lexington, MA, USA (2008)
41. Seinstra, F., Bal, H., Spoelder, H.: Parallel Simulation of Ion Recombination in Nonpolar Liquids. *Future Generation Computer Systems* **13**(4-5), 261–268 (1998)

42. Seinstra, F., Geusebroek, J., Koelma, D., Snoek, C., Worring, M., Smeulders, A.: High-Performance Distributed Video Content Analysis with Parallel-Horus. *IEEE Multimedia* **14**(4), 64–75 (2007)
43. Seinstra, F., Koelma, D., Bagdanov, A.: Finite State Machine-Based Optimization of Data Parallel Regular Domain Problems Applied in Low-Level Image Processing. *IEEE Transactions on Parallel and Distributed Systems* **15**(10), 865–877 (2004)
44. Seinstra, F., Koelma, D., Geusebroek, J.: A Software Architecture for User Transparent Parallel Image Processing. *Parallel Computing* **28**(7–8), 967–993 (2002)
45. Snoek, C., Worring, M., Geusebroek, J., Koelma, D., Seinstra, F., Smeulders, A.: The Semantic Pathfinder: Using an Authoring Metaphor for Generic Multimedia Indexing. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **28**(10), 1678–1689 (2006)
46. Tan, J., Abramson, D., Enticott, C.: Bridging Organizational Network Boundaries on the Grid. In: *Proceedings of the 6th IEEE International Workshop on Grid Computing*, pp. 327–332. Seattle, WA, USA (2005)
47. Taylor, I., Wang, I., Shields, M., Majithia, S.: Distributed Computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience* **17**(9), 1197–1214 (2005)
48. Urbani, J., Kotoulas, S., Maassen, J., Drost, N., Seinstra, F., van Harmelen, F., Bal, H.: WebPIE: A Web-Scale Parallel Inference Engine. In: *Third IEEE International Scalable Computing Challenge (SCALE2010)*, held in conjunction with the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2010). Melbourne, Australia (2010)
49. van Harmelen, F.: Semantic Web Technologies as the Foundation of the Information Infrastructure. In: P. van Oosterom, S. Zlatanove (eds.) *Creating Spatial Information Infrastructures: Towards the Spatial Semantic Web*. CRC Press, London (2008)
50. van Kessel, T., Drost, N., Seinstra, F.: User Transparent Task Parallel Multimedia Content Analysis. In: *Proceedings of the 16th International Euro-Par Conference (Euro-Par 2010)*. Ischia - Naples, Italy (2010)
51. van Nieuwpoort, R., Kielmann, T., Bal, H.: User-friendly and Reliable Grid Computing Based on Imperfect Middleware. In: *Proceedings of the ACM/IEEE International Conference on Supercomputing (SC'07)*. Reno, NV, USA (2007)
52. van Werkhoven, B., Maassen, J., Seinstra, F.: Towards User Transparent Parallel Multimedia Computing on GPU-Clusters. In: *Proceedings of the 37th ACM IEEE International Symposium on Computer Architecture (ISCA 2010) - First Workshop on Applications for Multi and Many Core Processors (A4MMC 2010)*. Saint Malo, France (2010)
53. Verstoep, K., Maassen, J., Bal, H., Romein, J.: Experiences with Fine-grained Distributed Supercomputing on a 10G Testbed. In: *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'08)*, pp. 376–383. Lyon, France (2008)
54. Waltz, D., Buchanan, B.: Automating Science. *Science* **324**, 43–44 (2009)
55. Website: EGI - Towards a Sustainable Production Grid Infrastructure; <http://www.eu-egi.eu>
56. Website: Open European Network for High-Performance Computing on Complex Environments; http://w3.cost.esf.org/index.php?id=177&action_number=IC0805
57. Website: SETI@home; <http://setiathome.ssl.berkeley.edu>
58. Website: Top500 Supercomputer Sites; <http://www.top500.org>; Latest Update (2009)
59. Wojcik, D., Warnick, W., Carroll, B., Crowe, J.: The Digital Road to Scientific Knowledge Diffusion: A Faster, better Way to Scientific Progress? *D-Lib Magazine* **12**(6) (2006)
60. Wrzesnińska, G., Maassen, J., Bal, H.: Self-Adaptive Applications on the Grid. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pp. 121–129. San Jose, CA, USA (2007)