

Ibis: Real-World Problem Solving using Real-World Grids

H.E. Bal, N. Drost, R. Kemp, J. Maassen,
R.V. van Nieuwpoort, C. van Reeuwijk, and F.J. Seinstra
Department of Computer Science, Vrije Universiteit,
De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands
{bal, ndrost, rkemp, jason, rob, reeuwijk, fjseins}@cs.vu.nl

Abstract

Ibis is an open source software framework that drastically simplifies the process of programming and deploying large-scale parallel and distributed grid applications. Ibis supports a range of programming models that yield efficient implementations, even on distributed sets of heterogeneous resources. Also, Ibis is specifically designed to run in hostile grid environments that are inherently dynamic and faulty, and that suffer from connectivity problems.

Recently, Ibis has been put to the test in two competitions organized by the IEEE Technical Committee on Scalable Computing, as part of the CCGrid 2008 and Cluster/Grid 2008 international conferences. Each of the competitions' categories focused either on the aspect of scalability, efficiency, or fault-tolerance. Our Ibis-based applications have won the first prize in all of these categories. In this paper we give an overview of Ibis, and — to exemplify its power and flexibility — we discuss our contributions to the competitions, and present an overview of our lessons learned.

1 Introduction

In 2001, grid experts Foster, Kesselman, and Tuecke published one of the most influential and defining papers in the field of grid computing [3]. The authors indicate that the fundamental problem that underlies the grid concept is "flexible, secure, coordinated resource sharing and problem solving in dynamic collections of individuals, institutions, and resources". Also, the field is "distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance orientation". With "grid-enabled programming systems" available that "enable familiar programming models to be used", this definition holds a promise, i.e. *the promise of efficient and easy-to-use wall-socket computing over a distributed set of resources* [11].

The promise of abstractions that would allow large sets of computing and storage resources to be seen as a single virtual supercomputer has raised high expectations — especially in domains that have an immediate need for large-scale distributed compute power (e.g., astronomy, multimedia). Despite rewarding results that have been obtained for specific application types (i.e., parameter sweeps [1], workflow driven problems [5, 13]), the field of grid computing at large has not yet been able to live up to its promise. This is unfortunate, because — in contrast to common belief — many grid systems allow for a much larger class of communication-intensive and distributed applications to be executed effectively [14].

We ascribe this rather limited use of grid systems to the intrinsic complexity of writing and deploying distributed applications. Grid programmers often are required to use low-level programming interfaces that change frequently. Programmers also must deal with system- and software-heterogeneity, connectivity problems, and resource failures. Also, managing a running application is complicated, because the execution environment may change dynamically as resources come and go. All of these problems limit the acceptance of grid computing technology.

The Ibis project aims to overcome these problems, and to drastically simplify the programming and deployment process of (high-performance) grid applications. The Ibis philosophy is that grid applications should be developed on a local workstation and simply be launched from there. This 'write-and-go' philosophy is directly derived from the abovementioned 'promise of the grid': it requires minimalistic assumptions about the execution environment, and expects most of the environment's software (e.g., libraries) to be sent 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.

This paper focuses on real-world problems that were defined in two competitions organized by the IEEE Technical Committee on Scalable Computing (see www.ieeetcsc.org): (1) The First IEEE International Scalable Computing Challenge (SCALE 2008, held in conjunction with CCGrid 2008 in Lyon, France), and (2) The First International Data Analysis Challenge for Finding Supernovae (DACH 2008, held in conjunction with IEEE Cluster/Grid 2008 in Tsukuba, Japan). The objective of these competitions was to highlight and showcase real-world problem solving and large-scale data analysis, by applying scalable, efficient, and fault-tolerant computing techniques using real-world grid systems. Our winning results, in all competition categories, are a clear indication of the broad applicability of the Ibis system.

This paper is organized as follows. Section 2 gives an overview of Ibis. Section 3 presents the SCALE 2008 challenge, as well as our winning contribution. Section 4 explains the two competition categories of DACH 2008, and presents our contribution to each of these. Related work is discussed in Section 5. Lessons learned, as well as conclusions, are given in Section 6.

2 The Ibis Project

The Ibis project (see also www.cs.vu.nl/ibis/) aims to provide transparent solutions to all inherent grid computing problems, and — as such — to make significant steps forward towards realizing the 'promise of the grid'. To this end, Ibis provides a rich software stack that provides all functionality that is traditionally (e.g., for sequential computers) associated with *programming languages* 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 Application Programming System and the Distributed Application Deployment System (see Figure 1).

2.1 The Ibis High-Performance Application Programming System

The Ibis High-Performance Application 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 High-Performance Application Programming System. It is a communication library that 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.

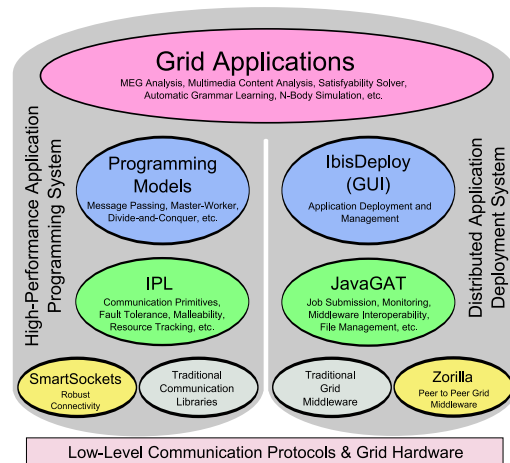


Figure 1. Ibis: high level overview.

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.

To deal with real-world grid systems, in which resources can crash, and can be added or deleted, the IPL incorporates a runtime mechanism that keeps track of the available resources. The mechanism, called Join-Elect-Leave (JEL), 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 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).

(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.

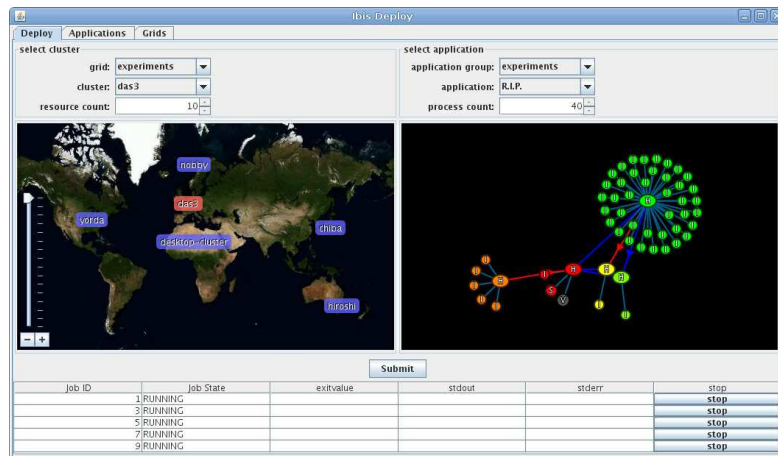


Figure 2. The IbisDeploy GUI that enables runtime loading of grids and applications (top), and keeping track of running processes (bottom). Center left shows the locations of available resources; center right shows SmartSockets connections between these resources.

The most transparent model of these is Satin [15], 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 grid 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 was always capable of establishing a connection, while traditional sockets only worked in 6 of these [6].

2.2 The Ibis Distributed Application 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, described below.

(1) The Java Grid Application Toolkit (JavaGAT): Today, grid programmers generally have to implement their applications against a grid middleware API that changes frequently, is low-level, unstable, and incomplete [7]. 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. It does so by integrating multiple middlewares into a single coherent system. Extensions are frequently added by ourselves and others. JavaGAT further defines a powerful API for middleware developers to allow experiments with various middleware designs without hindering the application programmers.

JavaGAT allows grid applications to transparently access remote data and spawn off jobs. It also provides support for monitoring, steering, user authentication, resource discovery, and storing of application-specific data. JavaGAT calls are dynamically forwarded to one or more middlewares that implement the requested functionality. For example, JavaGAT supports file operations through GridFTP, SSH, HTTP, and SMB/CIFS, and resource management with GRMS, Globus/(WS) GRAM, SSH, PBS, SGE, ProActive, and Zorilla. If a grid operation fails, it automatically dispatches the API call to an alternative middleware.

(2) IbisDeploy: 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 grid application written in Ibis, a network of Smart-Sockets hubs must be started manually. IbisDeploy takes over this task in a fully transparent manner.

The IbisDeploy GUI (see Figure 2) allows a user to manually load grids and applications at any time. As such, multiple grid applications can be started using the same graphical user interface. More interestingly, the IbisDeploy GUI allows the user to add new resources to a running application. In addition, the GUI also allows a certain level of application steering.

(3) Zorilla: As traditional grid middlewares lack co-scheduling capabilities, Ibis provides its own middleware implementation. The system, called Zorilla, is a light-weight peer-to-peer (P2P) middleware system that runs on clusters, grids, clouds (Amazon), and desktop machines. In contrast to traditional middlewares, it has no central components. It supports both fault-tolerance and malleability and it is easy to set up and maintain. Also, Zorilla was specifically designed to allow parallel applications to run concurrently on resources in multiple administrative domains.

As Zorilla is the only part of Ibis that was not used in our contributions to the SCALE and DACH challenges, we will not discuss it further.

3 SCALE 2008

SCALE 2008, or the First IEEE International Scalable Computing Challenge, is a competition organized by the IEEE Technical Committee on Scalable Computing (TCSC, see also www.ieeetcsc.org), and endorsed by the IEEE Technical Committee on Parallel Processing (tcpp.computer.org). The objective of the competition, held in conjunction with the CCGrid 2008 international conference in Lyon, France, was to highlight and showcase real-world problem solving using scalable computing techniques. All participants were expected to identify significant real-world problems where scalable computing can be effectively used, and to design, implement, evaluate and demonstrate their solutions.

Six finalists, from a total of 11 teams, were invited to participate in the challenge and to present their competition contribution. The participating teams were judged by a panel of experts based on the following criteria: (1) the significance and potential impact of the application and its appropriateness for scalable computing, (2) novelty, complexity and correctness of

the presented solution and the results achieved, (3) extent to which the presented solution pushes the envelope in scalable computing, (4) demonstration and presentation by the team. The panel of experts selected two of the participating teams (i.e. Konrad-Zuse-Zentrum für Informationstechnik, Berlin, and Vrije Universiteit, Amsterdam) as shared first prize winner. Our Ibis-based contribution to the SCALE 2008 competition is described below.

3.1 Wall-Socket Multimedia Computing

With the increasing omnipresence of multimedia data, automatic *multimedia content analysis* (MMCA) is rapidly becoming a problem of phenomenal proportions. At SCALE 2008 we presented a scalable distributed supercomputing solution for the MMCA domain. Specifically, we developed an application in which a digital camera is capable of real-time 'recognition' of objects from a set of learned objects, while being connected to a world-wide grid system (see Figure 3). Object recognition is a computationally demanding problem that involves a non-trivial tradeoff between specificity of recognition and invariance (e.g., to different lighting conditions).

With our application we demonstrated true *wall-socket grid computing*. The entire application, including all required libraries, was stored on a memory stick, which could be plugged into any Linux or Windows laptop with an appropriate JVM installed. From there, the application was compiled and started (using IbisDeploy), with the world-wide set of available grid resources being employed entirely transparently.

For our SCALE participation we have put most effort in the implementation of a transparent programming model for multimedia computing, directly on top of the IPL. The model, called Jorus, is a full Ibis implementation of the Parallel-Horus system [10]. As such, Jorus is a cluster programming model that allows its

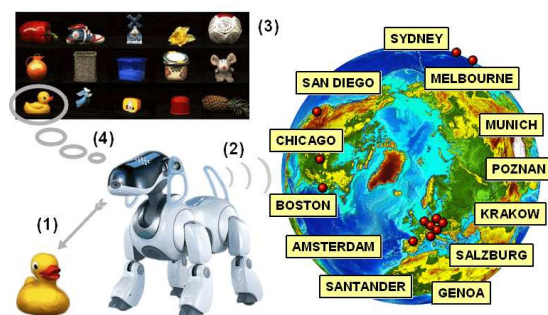


Figure 3. Overview of the MMCA application. A video of is available at www.cs.vu.nl/~fjseins/aibo.shtml

users to implement parallel multimedia applications as fully sequential programs, using a carefully designed set of building block operations that hide all complexities of parallelization behind a familiar sequential API.

For reasons of efficiency, Jorus applies an advanced runtime optimization approach that automatically parallelizes a sequential program by inserting communication primitives and memory management operations whenever necessary. Earlier results obtained with Parallel-Horus have shown the feasibility of the approach, with data parallel performance consistently being found to be optimal with respect to the abstraction level of message passing programs.

In our SCALE application, the processing of a single video frame is a data-parallel Jorus task (executed on a cluster) and calculations over consecutive frames are executed in a task-parallel manner on a grid. Because Jorus aimed to mimic Parallel-Horus as much as possible, the underlying IPL was configured to run using a *closed-world* pool of resources: like MPI, Jorus is not fault-tolerant at the level of single compute nodes, as no resources can be added or removed at application run-time. Still, for the full distributed application cluster-level fault-tolerance and malleability was obtained easily, by letting multiple Jorus instances run together under an open-world model.

During our SCALE demonstration we concurrently used up to 20 clusters in Europe, USA, and Australia (commonly employing 500-800 cores in total), with a mix of grid middlewares (including Globus, Zorilla, and even SSH), and under a variety of connectivity problems solved by SmartSockets. We often obtained recognition at a rate of 10 to 15 frames per second, where the original sequential code took 30 seconds per frame on a single core of a fast desktop computer. Clearly, without the benefits of Ibis, moving the application from a controlled laboratory setting to a real-world system would have been close to impossible.

4 DACH 2008

DACH 2008, the First International Data Analysis Challenge for Finding Supernovae, was held in conjunction with the IEEE Cluster/Grid 2008 international conference in Tsukuba, Japan. The competition was organized by the IEEE Technical Committee on Scalable Computing, the Japan MEXT grant-in-aid for priority area research called Info-Plosion, and the Special Interest Group on High Performance Computing (SIGHPC) of the Information Processing Society of Japan (IPJS). The competition was driven by the observation that the importance of large scale data analysis increases every year, not only in the scientific domain (e.g. astronomy, biology), but also in industry.

In the DACH challenge a large distributed database (of several hundreds of GBytes) of scientific data, gathered by the Subaru telescope in Hawaii, had to be searched to find new and unknown ‘supernova candidates’. For the calculations, the participants were given access to a supercomputer system comprising of 12 compute clusters distributed over Japan — i.e. a combination of the InTrigger Info-Plosion platform and Tokyo Tech Presto III, with a combined total of 1163 cores and over 140 TByte of storage.

To find all supernova candidates, it was necessary to compare image pairs at an interval of one month. The data analysis program itself, a sequential program implemented in C called ‘SuperFind’, was provided by the competition organizers. The participants were allowed to optimize the SuperFind program, and to embed it in any kind of distributed software environment, as long as the calculated results would be identical to the results that were pre-calculated by the organizers. With all these preconditions being identical, the competition was split into two different categories: (1) The Basic Category (BS), in which the goal was to perform all of the necessary calculations as fast as possible, (2) The Fault-Tolerant Category (FT), which had the added complexity of artificial node failures.

Fourteen teams participated in DACH 2008. The contest was considered difficult: not more than 9 teams submitted a result in the BS category, and only 1 team completed the FT category. Our Ibis-based contributions are described below.

4.1 Hierarchical Task Farming with Ibis

Initial runs of the SuperFind program on a small sample data set showed that the execution time was widely varying. For some input image pairs the execution took only 30 seconds, while for other (similar sized) input pairs the program took over 40 minutes to complete. When calculating in a distributed fashion, such wide fluctuations in execution times easily result in load imbalances - and a longer overall execution time. To overcome this problem, and to speed up the program as a whole, we have parallelized the original SuperFind code. Although there is no fundamental reason why the SuperFind program could not be implemented in Java, due to the sheer complexity of the original code, and the need to present *identical* results even under fluctuations in the internal storage of floating point numbers by different programming languages, we decided to keep C as the language of choice. This part of our solution is, therefore, outside the scope of this paper. The following description indicates, however, the ease with which ‘legacy’ codes in languages other than Java are integrated with Ibis.

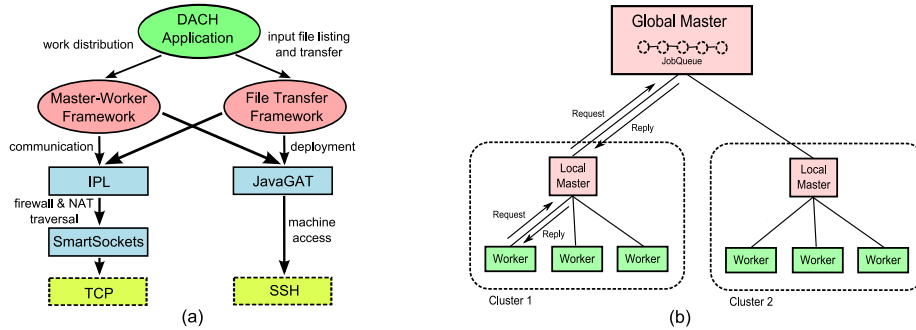


Figure 4. (a) Design of the DACH application. (b) The hierarchical master-worker model.

For the BS problem, we developed a new Master-Worker Framework and a File Transfer Framework (see Figure 4(a)). Our application starts by obtaining a directory where the input files can be found. The application then uses the Ibis File Transfer library to obtain a list of files available on the different clusters. To perform this operation, a simple File Server is started on each of the sites. This File Server is implemented in Java and is based on the IPL, which — in turn — uses SmartSockets. The JavaGAT is used to start these File Servers over SSH. Note that, on different systems, this could be any other middleware, such as Globus, with little or no change to the application code.

Once the DACH application has obtained a list of all input files, it generates a set of Jobs. Each job contains a pair of files that need to be compared using the SuperFind program. The jobs also contain a list of locations where each of the files can be found. These jobs are passed on to the Master-Worker Framework, which uses the JavaGAT to start a single Local Master on each of the participating sites. In turn, each Local Master uses the JavaGAT to start a worker on each of the compute nodes of its local cluster. The framework itself acts as the Global Master. This results in a hierarchical master-worker setup, as shown in Figure 4(b).

Whenever a worker is idle, it requests a job from its Local Master, who forwards the request to the Global Master. If a job is available, it will be returned using the reverse path. This approach is chosen because the input files of some jobs are replicated on multiple clusters. By using a single centralized job queue, we prevent these jobs from being run more than once. Also, by routing all worker request through a Local Master on each cluster, the number of required network connections is significantly reduced.

Once a worker receives a job, it calls a DACH application specific function to process the job. This function copies the input files to a temporary directory on the local disk, either using NFS (in case of local files), or using the Ibis File Transfer library (in case of remote files). It then calls the SuperFind program to

process the files. The result of the comparison is then sent to the Global Master (again via the Local Master), which returns it to the DACH application. When all results have been received, they are checked against the results that were pre-calculated by the organizers.

In initial experiments, we (and other teams) found that data transfer between sites occasionally took an excessive amount of time. This behavior is shown in Figure 5, plotting the total time and related data transfer for each job. As a result, a complete run, consisting of 1052 jobs to be processed, would spend half of its time waiting for the last 100 jobs. To overcome this problem, we let each cluster process local files only. As a consequence, some clusters finished earlier than others, but no time was lost in data transfer.

Our Ibis-based solution obtained by far the fastest result. While we required only 36 minutes for all calculations, the second best team used more than 1 hour. All other teams obtained run-times of 3 to even more than 25 times longer. Our result was partially due to our parallelization of the SuperFind program. Without parallelization, our solution runs in under 50 minutes — still significantly faster than all other teams. More importantly, we benefited from the flexibility of Ibis. This allowed us to implement multiple solutions in a short time frame, using a grid system that we — in contrast to many other teams — had no experience with until the start of the competition.

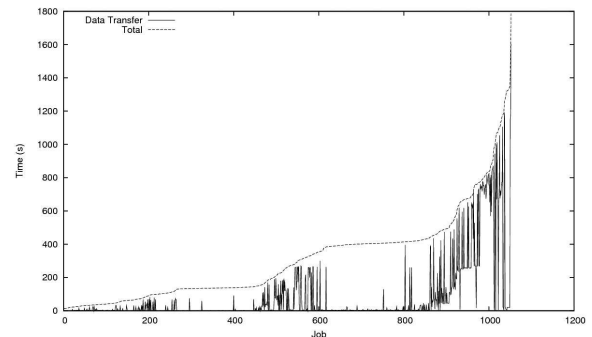


Figure 5. Total time vs. data transfer time.

4.2 Maestro: Self-Organizing Data-Flows

The purpose of also participating in the fault tolerant challenge was to test a new Ibis programming model: Maestro, a system for data-flow computations in real-world grid systems. An important feature of Maestro is that it is self-organizing. In particular, distribution of computations over nodes is based on local decisions made by each of the nodes independently, not by a central coordinator. The only special nodes in the system are the ‘Maestro’ nodes that put input data into the system and that handle the final result.

Each node has two queues: a submission queue and a work queue. The tasks in the submission queue are submitted to the ‘best’ node in the system for execution, either the local node, or a remote one. To select the best node, the capabilities and performance of all nodes are continuously ‘gossiped’ by a background process on each node. To avoid idle times, the work queue on each node contains tasks to be executed on the local node only. To allow rapid reaction to changing circumstances, the work queue is kept small.

Maestro is designed for data-flow systems, where computations ‘flow’ through a number of distinct processing steps. By interpreting the DACH computations as a flow of size one, we simply reduced Maestro to a master-worker configuration. More problematic was the central assumption in Maestro that a computation consists of repeated tasks, each with comparable execution time. Nodes use these properties to learn the best allocation of computations to nodes based on previous experiences. The DACH computations do not match these assumptions, since the SuperFind program has widely varying computation times. Moreover, since the DACH challenge was relatively small-scale, there is limited opportunity to learn from earlier calculations. In fact, for best performance the system cannot afford to learn at all; from the start it must assign the longest running jobs in an optimal manner. We solved this problem by introducing a new type of tasks in Maestro: unpredictable tasks. For such tasks Maestro runs a benchmark on each node of the system on startup. Maestro then favors task submission to nodes with fast benchmark results.

Since the SuperFind program had such widely varying computation times, it was important to identify the longest computations, and start these as soon as possible. As it is reasonable to assume that there is a correlation between image file size and the duration of the computation, the master first assigns the image pair with the largest total file size to the best node, then the second largest to the best remaining node, and so on.

With this adapted Maestro framework we participated in the FT challenge. We could do so thanks

to the resource tracking mechanisms provided by the Join-Elect-Leave (JEL) model, discussed earlier. Whenever a task is terminated involuntarily, Maestro is notified of this event, such that appropriate action (i.e. the restarting of the same task) can be taken.

Maestro was the only system capable of participating in the FT challenge; it finished all computations in 5.5 hours. This execution time is not very relevant, as it is largely determined by the high cost of file transfer, and partially also by the number of processes being killed. For faster execution, it would have helped to use the parallel version of SuperFind, but since our effort was for the purpose of testing Maestro, we used the standard version. Maestro used a total of 92 nodes, 34 (i.e. more than one third) of which were killed, and one of which crashed by itself. Maestro automatically restarted about 10 percent of all SuperFind comparisons, before returning the correct result.

5 Related Work

ProActive [2] is one of the few systems that, like Ibis, follows a dual-subsystem approach. It contains several grid programming models and provides grid deployment and virtualization components. Whereas Ibis has a strong focus on performance, the core of ProActive is more heavy-weight. ProActive further differs from Ibis in that it requires the user to manually handle all connection setup problems and to manually (and statically) select the appropriate middleware. Finally, ProActive is one integrated system, from which individual components cannot be used separately.

Phoenix [12] also follows a dual-subsystem approach. It consists of a message passing model that, like the Ibis IPL, allows compute nodes to be added to and removed from a running application. Phoenix automatically deals with several connection setup problems (e.g., firewalls), but our SmartSockets solution is more extensive. Phoenix also provides easy-to-use tools that handle common grid operations. Unlike the JavaGAT system, however, it cannot automatically exploit different grid middlewares at the same time.

Genesis II [8] is an open source, and standards-based grid system for compute and data intensive applications. Its main focus is on grid deployment, under the motto “*by default the user should not have to think*”. Genesis II extensively uses the file-system paradigm, allowing transparent access through familiar mechanisms (such as drag-and-drop remote job execution), giving functionality similar to the Ibis JavaGAT. It does not focus on high-performance (distributed) computing, however, and lacks a grid-centric communication system (such as our IPL) or higher level grid programming models.

The Open Grid Forum is currently standardizing the next generation grid programming toolkit: the Simple API for Grid Applications (SAGA) [4]. The goal is to provide a "grid counterpart to MPI" (at least in impact), that would supply developers with a simple, uniform, and standard programming interface for distributed applications. SAGA will support several programming languages and will provide consistent semantics and style for different grid functionality. Notably, the Java Reference Implementation of SAGA is implemented directly on top of Ibis JavaGAT.

Finally, SyD [9] is a Java-based dual-subsystem middleware for collaborative and distributed applications over mobile and other devices. While SyD seems to be no longer under active development, it is a relevant reference given the fact that an Ibis implementation is available for the Android smart phone platform.

6 Lessons Learned and Conclusions

Contests such as described in this paper can play an important role in pushing forward the state-of-the-art in the field. Having to solve real problems on real grid systems forces the development process of applications, runtime systems, and middlewares to be more than a theoretical exercise. Performing grid research on 'hostile' systems is essential, as many grid users come from scientific and industrial domains. These users need to solve real problems using real grids.

SCALE, DACH, and other contests have helped us to identify and fix shortcomings in Ibis over the years, and inspired us to develop solutions for problems in *real-world* grids. As a result, Ibis now solves most of the inherent grid complexities transparently, allowing more time to be spend on the actual problem at hand.

We have also learned that Ibis can provide real-time (SCALE) as well as off-line (DACH) solutions for different domains (i.e., multimedia and astronomy). Also, from the fact that our SCALE solution was a pure Java/Ibis application, while our DACH solutions comprised of a mix of Java, Ibis, and 'legacy' codes, we conclude that Ibis is likely to have broad applicability, in many application areas, and under many domain- and application-specific requirements.

Clearly, we also encountered difficulties, due to a lack of a sufficiently large set of available programming models, and due to shortcomings in earlier implementations. Apart from the lack of support for unpredictable tasks, Maestro suffered from a lack of support for locality-aware execution, to favor computations to be performed where data resides. Also, Ibis lacked a hierarchical master-worker model, and a library for distributed file access. These problems have been fixed, making Ibis as a whole much more mature.

To conclude: participation in the two contests was a great experience, and we will continue to embrace these and other competitions in the future. We sincerely hope that other research projects in the domains of scalable and high-performance grid computing will do the same. When this happens, there is a realistic chance that such competitions indeed will push forward the overall state-of-the-art in our research field.

References

- [1] D. Abramson et al. Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. In *HPDC'95*, Pentagon City, USA, Aug. 1995.
- [2] L. Baduel et al. Programming, Deploying, Composing, for the Grid. In *Grid Computing: Software Environments and Tools*. Springer-Verlag, Jan. 2006.
- [3] I. Foster et al. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. of High-Perform. Comput. Appl.*, 15(3):200–222, 2001.
- [4] T. Goodale et al. SAGA: A Simple API for Grid Applications, High-Level Application Programming on the Grid. *Comp. Meth. Sci. Tech.*, 12(1):7–20, 2006.
- [5] B. Ludäscher et al. Scientific Workflow Management and the KEPLER System. *Concur. Comput.: Pract. Exp.*, 18(10):1039–1065, Aug. 2005.
- [6] J. Maassen et al. SmartSockets: Solving the Connectivity Problems in Grid Computing. In *Proc. HPDC'07*, pages 1–10, Monterey, USA, June 2007.
- [7] R. Medeiros et al. Faults in Grids: Why are they so bad and What can be done about it? In *Proc. 4th International Workshop on Grid Computing*, pages 18–24, Phoenix, AZ, USA, Nov. 2003.
- [8] M. Morgan and A. Grimshaw. Genesis II - Standards Based Grid Computing. In *Proceedings of CCGrid'07*, pages 611–618, Rio de Janeiro, Brazil, May 2007.
- [9] S. Prasad et al. SyD: A Middleware Testbed for Collaborative Applications over Small Heterogeneous Devices and Data Stores. In *Proc. Middleware'04*, pages 352–371, Toronto, Canada, Oct. 2004.
- [10] F. Seinsträ et al. High-Performance Distributed Video Content Analysis with Parallel-Horus. *IEEE Multimedia*, 15(4):64–75, Oct. 2007.
- [11] V. Sunderam. True Grid: What Makes a Grid Special and Different? Keynote Lecture ICCS'04, Cracow, Poland, June 2004.
- [12] K. Taura et al. Phoenix: Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. In *PPoPP'03*, San Diego, USA, June 2003.
- [13] I. Taylor et al. Distributed Computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, Aug. 2005.
- [14] K. Verstoep et al. Experiences with Fine-grained Distributed Supercomputing on a 10G Testbed. In *Proc. CCGrid'08*, pages 376–383, Lyon, France, May 2008.
- [15] G. Wrzesińska, J. Maassen, and H. Bal. Self-Adaptive Applications on the Grid. In *Proc. PPoPP'07*, pages 121–129, San Jose, CA, USA, Mar. 2007.