Experiences Deploying Parallel Applications on a Large-scale Grid

Rob V. van Nieuwpoort, Jason Maassen, Andrei Agapi, Ana-Maria Oprescu, and Thilo Kielmann
Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
email: {rob,jason,aagapi,aoprescu,kielmann}@cs.vu.nl

Abstract— **We describe our experiences with integrating several Grid software components into a single coherent system that is used to write and run parallel applications on the Grid. The integrated components are the Grid Application Toolkit (GAT), ProActive, Satin and Ibis. We experimented with this (Java-based) system by participating in the N-Queens contest of the Grids@work event in October 2005. In addition to integrating available components, we wrote a ProActive plugin for the GAT, a parallel N-Queens solver, and an application to manage Grid deployment of N-Queens. We identified several connectivity issues and scalability problems in the components we use. We show how we modified some of the components to solve of these problems. We successfully ran experiments on 960 processors across Grid'5000, with an efficiency of around 85%, winning the prize for the largest number of nodes deployed during the contest.**

I. Introduction

The Grids@work event held in October 2005 in Sophia Antipolis, France [7] was composed of a series of conferences and tutorials including the 2nd Grid Plugtests. The objective was to bring together Grid users and to present and discuss current and future features of the ProActive Grid platform, and to test the deployment and interoperability of Grid applications on various Grids. A part of the 2nd Grid Plugtests consisted of an N-Queens contest, where the aim was to find the number of solutions to the N-Queens problem, N being as big as possible, in a limited amount of time.

We participated in this contest with a parallel N-Queens application. We used this application and the Grid testbed that was provided to integrate many software components, and to evaluate the integration, functionality and performance. The global structure of the system we used is shown in Figure 1. For portability reasons, all software components are written in Java. The N-Queens application itself is written in Satin, our Java-based divide-and-conquer programming model [9]. Satin is implemented on top of the Ibis [11] communication library, while deployment of the application was done with a manager application that was written specifically for this contest. The manager uses the Java Grid Application Toolkit [2] (GAT) to access the Grid. The Java GAT in turn uses the ProActive [5] middleware for Grid deployment.

This paper describes our experiences with integrating all different software components we used, and identifies some problems in our software packages, Ibis and Satin in particular, that we discovered during the contest. Some are related to scaling a parallel programming system up to 1000 machines that are distributed over a large geographical area, others were related to typical Grid issues such as firewall problems and network misconfigurations. Although we did encounter some difficulties, we were still able to run the parallel N-Queens application on 961 CPUs scattered across different Grid'5000 sites in France. Finally, we suggest possible solutions for the problems encountered. After identifying and solving the problems we describe in this paper, we won the prize for largest number of nodes deployed in parallel during the contest.

The remainder of this paper is structured as follows. First, we discuss the deployment tools GAT and ProActive (Section II), followed by Ibis and Satin (Section III), and then the N-Queens application itself (Section IV) and the testbed (Section V). Section VI will discuss the issues we encountered using a large-scale Grid, along with the solutions we applied. Sections VII and VIII summarize results and our conclusions, respectively.

II. Deployment of Grid Applications

As shown in Figure 1, we can compile and deploy applications from any workstation directly onto the Grid. Thanks to Java's "write once, run anywhere" portability, recompilation or configuration is not necessary, we can run all our code on any Grid site and architecture that supports Java.

The Grid Application Toolkit (GAT) enables us to deploy software from a workstation to the Grid. The GAT allows applications to abstract from the underlying Grid middleware layers. A GAT application simply uses the GAT API. The GAT, in turn, could internally use Globus, SSH or ProActive. In this paper, we used the ProActive plugin (adaptor) for the GAT. For the contest, we wrote a separate deployment and management application that uses the GAT. This manager runs on a workstation, and deploys the parallel (in this case N-

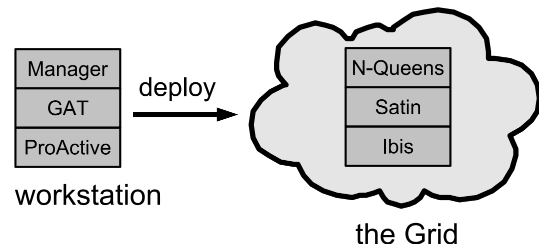

Fig. 1. Structure of the experiment setup.

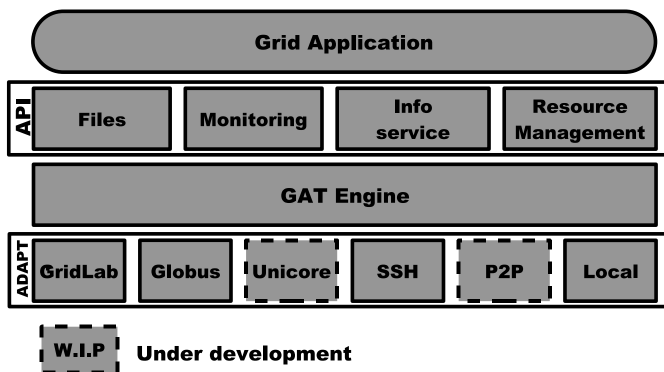

Fig. 2. GAT Design. The various adaptor modules can be loaded dynamically, using run time class loading.

Queens) application on the Grid. In this section we focus on the manager application (Figure 1, left). Sections III and IV will describe the N-Queens application as deployed on the testbed (Figure 1, right).

A. GAT: The Grid Application Toolkit

While Grid technologies are maturing rapidly, there still remains a shortage of real Grid applications. One important reason is the lack of simple and high-level application programming interfaces to the Grid, bridging the gap between existing Grid middleware and application-level needs. The Grid Application Toolkit (GAT), as has been developed by the EC-funded GridLab project [1], provides a unified, simple programming interface to the Grid infrastructure, tailored to the needs of Grid application programmers and users. This section briefly outlines the general language independent GAT architecture and its Java implementation. A more detailed presentation can be found in [2].

In Grid platforms, or Virtual Organizations (VO's), applications access a variety of more or less heterogeneous resources. For most applications, the important resources are computers and data files. For computers, heterogeneity stems from different hardware architectures and operating systems. For (remote) files, different access protocols exist. Moreover, the administrative autonomy of the member sites in a VO leads to a diversity of access policies and installed middleware packages and package versions. Unfortunately, even minor differences between versions of the same middleware package often lead to prohibitive incompatibilities. Access to data files suffers from similar problems. Hiding this heterogeneity and even the individual entities in a Grid, in favor of providing a unified distributed computing platform, is important for writing Grid-enabled applications. The GAT's main objective is to provide a single, easy-to-use Grid API, while hiding the complexity and diversity of the actual Grid resources and their middleware layers. The GAT provides abstractions for computers, data, and application instances (running jobs).[1]

[1]Within the Global Grid Forum (GGF) we are currently designing the Simple API for Grid Applications (SAGA) which is intended as a standardized version of the GAT API.

Figure 2 shows the GAT architecture. The top layer, the GAT API, is the only part that is visible to Grid applications. The GAT adaptors are Grid middleware-specific, and implement the actual Grid operations. The GAT engine is a lightweight layer that dispatches GAT API calls to Grid service invocations via GAT adaptors. The GAT engine typically loads adaptors dynamically at runtime, whenever a certain service is needed. An engine may load multiple adaptors implementing the same subsystem at the same time. For example, one adaptor can give access to a local file system while another one gives access to files via GridFTP, and a third one is using SSH. The application simply uses the GAT API, and does know anything about the underlying middleware and protocols.

The GAT API has been designed using an object-based approach, trying to avoid dependencies on any specific language binding. The Java version of the GAT was implemented by our group at the Vrije Universiteit in Amsterdam. Java has several properties making it attractive for Grid computing, notably it's "write-once, run anywhere" portability. Java code can run without recompilation on any Grid site that has a JVM installed. The Java GAT is truly object-oriented. Java's exception handling mechanism considerably simplifies error-handling code. The Java GAT engine implements the dispatching of API calls to the adaptors. Although it would be possible to re-use adaptors written for the C GAT via the Java Native Interface (JNI), we chose to pursue a pure Java implementation to retain Java's portability. This way, applications only need the GAT library (a jar file) and the adaptors (also jar files). No recompilation or configuration is necessary. We currently have adaptors for local operations, for the GridLab services, and for many other middlewares, such as Globus, SSH, FTP, HTTP, and ProActive.

An interesting property of the Java GAT is that it uses late binding: the actual adaptor to execute a GAT operation is selected only when the operation is invoked, and not when the corresponding GAT object is created. Late binding is more robust and flexible than static binding. For example, if a file is to be copied from site A to the sites B and C, different transfer mechanisms can be used for the transfer from A to B and for the transfer A to C. For example, the first transfer could use SSH, while the second uses GridFTP. Late binding also simplifies adaptor writing. For example, a file adaptor might only support a highly optimized implementation of file.copy, but no file.delete operation. The Java GAT engine will automatically fall back to another adaptor that can do the delete. This feature is convenient, as many Grid services do not provide the complete functionality that is offered by the GAT API. The Java GAT will then automatically use multiple services to implement the functionality of a single GAT object.

One of the strong points of the Java GAT is that users can develop Grid applications on their workstation or laptop. GAT will automatically use the local adaptors, for instance for file access and resource management (which forks jobs on the local machine). The user can write, test and debug the application without ever using another machine or a Grid. Next, he can deploy the application directly on the real Grid

```
public class RemoteCopy {
  public static void main(String[] args) {
    GATContext context = new GATContext();
    URI src    = new URI(args[0]);
    URI dest   = new URI(args[1]);
    File file = GAT.createFile(context, src);
    file.copy(dest);
  }
}
```

This program can be executed as follows:

Local file copy:
java RemoteCopy /bin/echo foo

Copy remote file to local system with GSIFTP:
java RemoteCopy gsiftp://remote.host//bin/echo foo

Third party copy, GAT is allowed to choose the best protocol:
java RemoteCopy any://remote.host.one//bin/echo any://remote.host.two/foo

Fig. 3. *RemoteCopy*: an example Java GAT program.

from his own workstation, no code change or recompilation is necessary. Figure 3 illustrates this. The *RemoteCopy* program can copy local and remote files, using any protocol desired. Starting jobs on the Grid is done in a similarly simple way.

B. ProActive

ProActive [5] is a framework based on transparently accessible remote objects built for the purpose of providing support for seamless, object-oriented, distributed computing under Java, targeting portable execution of unmodified, distributed application code over networks of workstations/clusters, multiprocessors or any combination of these in a transparent manner. Besides this generic framework, the ProActive library also features other characteristics, such as support for group communication, mobility and components.

The deployment of ProActive objects can be done using a mechanism based on XML deployment descriptors, in which an administrator specifies the characteristics of the deployment site (e.g. number of nodes, cluster scheduler, access protocol, possible RMI registries etc). The ProActive application can subsequently be transparently deployed on the respective site. The ProActive team provided XML deployment descriptors for the Grid we used for this paper. For a more detailed description of the ProActive deployment mechanism, please see [3].

C. Interfacing GAT and ProActive

For the N-Queens contest, we could use a Grid that consisted of many sites, all using the ProActive middleware. In order to deploy our N-Queens application on this testbed, we had to write a ProActive adaptor for the JavaGAT. We use a separate management application to deploy the parallel N-Queens code. The management application itself does not know anything about ProActive, it simply uses GAT API calls.

The implementation of the ProActive GAT Adaptor was straightforward. In order to execute jobs on a Grid enabled with the ProActive middleware, our GAT adaptor first retrieves all ProActive XML deployment descriptors that completely describe the testbed that we can use to launch our jobs on.

At initialization time, the adaptor initializes ProActive, and starts one active object, called a *ProActiveLauncher* on each node of the Grid. The *ProActiveLauncher* is an Active remote object, which main functionality is to continuously accept remote service calls from our adaptor. One such call is the *launch* method, which takes as a parameter a class to be launched, the corresponding command-line arguments and possibly some JVM parameters to be set in the executing JVM. The launch method then executes the class in a Java Virtual Machine on the same node. Depending on a user-supplied parameter the JVM in which the class is executed can be either the same JVM in which the ProActiveLauncher runs (to reduce possibly unnecessary overhead when this is feasible) or a separate JVM that is specifically spawned for the purpose of running the job.

Job launch requests received through GAT API calls are served by a separate thread in our ProActive adaptor that picks an initialized node (i.e. a node on which a ProActiveLauncher has been successfully started) from the pool of currently available nodes. The adaptor copies the files that are needed to execute the job (input files and the executable) to the selected machine, using the SSH protocol. Next, it launches the job on that particular node by using the *launch* remote service invocation. We found that once the deployment descriptors are given, deploying Grid applications with ProActive is almost trivial.

D. Deployment and Management Application

For the contest, we wrote a "management application" that runs on a workstation. This application deploys and monitors the parallel N-Queens application on the Grid. The management application itself uses GAT. Thanks to the simplicity of the GAT interface, our management application is straightforward, hiding all implementation and deployment complexity. In addition, it is highly portable.

The management application uses the GAT API to build a job description. A job description contains the executable to be launched, the job's command-line arguments, files that must be pre- and post-staged, and *stdout*, *stdin* and *stderr* files used by the job. In our specific case, the job parameters would be the parameters needed for Java to correctly execute an Ibis N-Queens solver worker, including the required classpath and needed ibis setup parameters.

III. The Ibis Grid Programming System

This section describes the structure of the parallel N-Queens application and the layers it is built upon.

A. Ibis, flexible and efficient Java-based Grid programming

The overall structure of the Ibis system is shown in Figure 4. The central part of the system is the Ibis Portability Layer (IPL) which consists of a number of well-defined interfaces. The IPL can have different implementations, that can be selected and loaded into the application *at run time*. Although

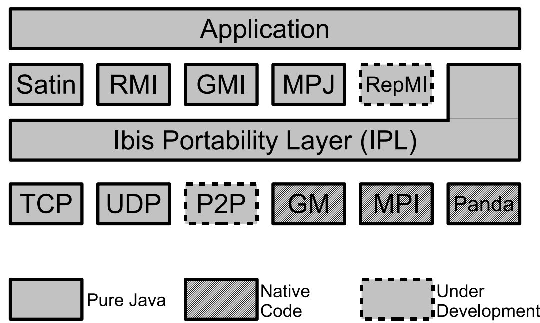

Fig. 4. Design of Ibis. The various modules can be loaded dynamically, using run time class loading.

it is possible to use the IPL directly from an application, Ibis also provides more high-level programming models. Currently, we have implemented four. Ibis RMI [11] provides Remote Method Invocation, using the same interface as Sun RMI, but with a more efficient wire protocol. MPJ is Java's language binding for MPI. GMI provides MPI-like collective operations, cleanly integrated into Java's object model. In this paper, we focus on our fourth programming model, Satin [9].

1) Combining high Performance with Portability: Key to the design of Ibis is to achieve a system that obtains high communication performance while still adhering to Java's "write once, run anywhere" model. The Ibis strategy to achieve both goals simultaneously is to develop reasonably efficient solutions using standard techniques that work "anywhere", supplemented with highly optimized but non-standard solutions for increased performance in special cases. We apply this strategy to both computation and communication. Ibis is designed to use any standard JVM, but if a native, optimizing compiler (e.g., Manta [8]) is available for a target machine, Ibis can use it instead. Likewise, Ibis can use standard communication protocols, e.g., TCP/IP or UDP, as provided by the JVM, but it can also plug in an optimized low-level protocol for a high-speed interconnect, like GM or MPI, if available. Moreover, Ibis offers a highly efficient object serialization mechanism to improve communication performance.

2) Ibis Firewall Traversal Support: Grid computing applications are challenged by current wide-area networks: firewalls, private IP addresses and network address translation (NAT) hamper connectivity, the TCP protocol can hardly exploit the available bandwidth, and security features like authentication and encryption are usually difficult to integrate. Existing systems (like GridFTP, JXTA, SOCKS) each address only one of these issues. However, applications need to cope with all of them, at the same time.

In Ibis, we have an integrated solution for these problems. Ibis lets applications span multiple sites of a Grid, and copes with firewalls, local IP addresses, and TCP bandwidth problems. In this paper, we focus on connectivity problems, as most problems we encountered during deployment occurred in that area. Ibis uses a flexible way of setting up connections between Grid sites, that automatically adapts to firewalls, if present. Different mechanisms to setup a connection are tried consecutively. All mechanisms use a timeout, making sure that if a connection setup mechanism does not work in a particular case, the next method can be tried.

The mechanism works as follows:

- First, Ibis tries to establish a direct connection. This works if no firewalls are present between the Grid sites. If there is a firewall that blocks incoming traffic, but there is an open port range, the ports will be allocated within this range, and a direct connection is still possible.
- Second, if the above does not work or results in a timeout, Ibis tries to reverse the connection establishment. Again, this mechanism takes open port ranges into account. A reverse connection setup is likely to work if only one of the two Grid sites uses a NAT setup or a firewall.
- Third, Ibis tries a technique called TCP splicing [6], which can setup a direct connection, even a site is behind a firewall. In addition to the asymmetrical client/server handshake, the TCP standard defines simultaneous initiation, also called simultaneous SYN or TCP splicing in the literature. This way of establishing a connection is symmetrical. At user level, both sides invoke connect at the same time to connect to each other. Neither of them invokes listen or accept. This mechanism requires a negotiation between both endpoints, however. The nodes have to invoke connect simultaneously, and both act as clients and do not know whether the other is ready. TCP splicing fails if the firewall blocks outgoing traffic.
- Finally, if all of the above fails or times out, Ibis tries to route all traffic through a third point that is not behind a firewall. Of course, this may be slower than a direct connection.

3) The Ibis Nameserver: Ibis supports malleability: nodes participating a parallel computation can join and leave the run at any time. To keep track of the nodes that participate, and to register communication endpoints, Ibis uses a centralized nameserver. If a machine joins the computation, the others are notified. A new machine contacts the Ibis nameserver, which then sends a join message to all machines already participating. The procedure is the same for a node that leaves the computation. Finally, as explained in the previous section, the nameserver can also route messages between clusters if a direct connection is not possible.

B. Satin

Satin's programming model is an extension of the single-threaded Java model. To achieve parallel execution, Satin programs do not have to use Java's threads or Remote Method Invocations (RMI). Instead, they use much simpler divide-and-conquer primitives. Satin does allow the combination of its divide-and-conquer primitives with Java threads and RMIs. Additionally, Satin provides shared objects.

Satin expresses divide-and-conquer parallelism entirely in the Java language itself, without requiring any new language constructs. Satin uses so-called *marker interfaces* to indicate that certain method invocations need to be considered for

potentially parallel (so called spawned) execution, rather than being executed synchronously like normal methods. Furthermore, a mechanism is needed to synchronize with (wait for the results of) spawned method invocations.

Satin's bytecode rewriter generates the necessary code. Conceptually, a new thread is started for running a spawned method upon invocation. Satin's implementation, however, eliminates thread creation altogether. A spawned method invocation is put into a local work queue. From the queue, the method might be transferred to a different CPU where it may run concurrently with the method that executed the spawned method. The sync method waits until all spawned calls in the current method invocation are finished; the return values of spawned methods are undefined until a sync is reached.

Spawned method invocations are distributed across the processors of a parallel Satin program by work stealing from the work queues mentioned above. In [9], we presented a new work stealing algorithm, *Cluster-aware Random Stealing* (CRS), specifically designed for cluster-based, wide-area (Grid computing) systems. CRS is based on the traditional Random Stealing (RS) algorithm that has been proven to be optimal for homogeneous (single cluster) systems [4].

IV. The Application: N-Queens

Initially, the N-Queens problem has been introduced in 1848 by chess player Max Bezzel, for eight queens on a eight-by-eight chessboard. It has been then generalized to arranging N queens on a N-by-N chessboard, such that none of them is able to successfully attack any other using standard chess queen's moves. A solution of the problem is a board configuration for which no two queens share the same row, column or diagonal. There are several different ways to formulate the problem: find one solution, find all solutions or count all solutions. Furthermore, given the symmetry operations (rotations and reflections) which can be applied to a board configuration, there is another discriminant on the solution set: uniqueness. Apart from its algorithmic appeal, the N-Queens problem has several practical applications, such as parallel memory storage design, traffic control and deadlock prevention.

An efficient approach to solving the problem is constructing a row-wise search tree; a node at depth k in the search tree represents a N-by-N board with k non-attacking queens. The search tree can be further pruned when considering symmetry properties. For the contest, we are not interested in storing solutions, but rather in efficiently counting them. Therefore we use a symmetry-based breadth-first approach [10], which finds only unique solutions and determines the total number of solutions using symmetry properties: each unique solution that has the first queen in the corner would generate seven more distinct solutions, each unique solution that does not have the first queen in the corner, would generate one, three or seven more distinct solutions, depending on the equality between the original configuration and a 90, 180 or 270-degree rotation.

Our implementation is a direct translation of Takahashi's C code in Java. The original code is already written in a divide-and-conquer style, so parallelizing it with Satin is just a matter

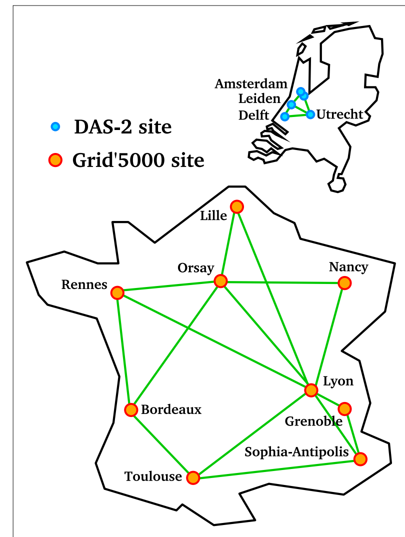

Fig. 5. The testbed we used during the N-Queens contest.

of indicating which methods can run in parallel. This is done with a marker interface that contains the methods that can be spawned. The resulting code still is normal Java, and can be compiled with any Java compiler, and run sequentially on any JVM. If compiled with the Satin compiler, the code can run in parallel.

V. The Testbed

During the five days of the Grid Plugtest N-Queens competition, a large testbed was available which consisted of approximately 846 machines (1518 CPUs) distributed across 27 locations worldwide. After the five day event, the teams where allowed to use any Grid infrastructure at their disposal to try and improve their results. New results could be submitted until one month after the start of the competition.

During the competition, our team used two distributed cluster systems to perform the experiments; the DAS-2 system which consists of 200 machines (400 CPUs) distributed over 5 clusters located in the Netherlands, and the Grid'5000 system, consisting of more than 1200 CPUs distributed across 9 clusters in France. Figure 5 shows the physical location and number of machines of each cluster.

The DAS-2 system is an older cluster in which each machine consist of a dual 1 Ghz Pentium-3 with 1 GB of memory. Grid'5000 is a more heterogeneous system that uses a wide range of processors, such as AMD Opterons, Intel Xeons and Itaniums, and IBM PowerPCs. Machines typically have 2 GB of memory, although some offer only 1 GB.

VI. Lessons Learned

The problems we have encountered and the lessons we have learned can be categorized into two parts: network connectivity problems, and issues regarding scalability of the parallel application. We will describe these two categories below.

A. Network Problems and Peculiarities

During the experiments, we encountered several network related problems: some did not significantly affect our application but mainly caused some inconvenience or confusion, others did affect our application but could be solved by manual intervention, and some required an extension of Ibis.

1) Peculiarities: At the time of the experiments, the compute nodes of the Grid'5000 system did not offer DNS resolution. As a result, the application was not able to resolve a hostname of a machine into an IP address. This caused some inconvenience, since the hostname of the machine running the Ibis name server is passed as a command line parameter when the application is started on a compute node. This problem could simply be solved by directly providing the IP address of the nameserver instead instead of its hostname. However, as will be explained below, for some machines finding the correct IP address was non-trivial.

The IP addresses used in Grid'5000's clusters were the cause of some confusion. Many clusters use IP addresses in a site-local range (as defined in RFC 1918), such as '192.168.x.y' or '10.x.y.z', which are normally only usable from within the cluster. Contrary to our expectations, however, these machines are directly reachable from the other Grid'5000 sites. The routing tables of all Grid'5000 machines contain the information necessary to create direct connections to the machines in those clusters.

2) Problems: As is often the case in cluster systems, many of the machines we used contain multiple network interfaces. The machines in the DAS-2 system, for example, are all connected to at least two networks: Fast Ethernet and Myrinet. Some machines even have a third network, Gigabit Ethernet, at their disposal. The network configuration of the machines in the Grid'5000 system is even more varied. This led to an interesting problem during our experiments, the *reverse routing problem*, where the sending machine must select one out of multiple IP addresses for a receiver. The problem is illustrated below.

When an application is started on a large number of machines, distributed over multiple sites, each machine registers itself at the nameserver by sending it information on how it can be contacted (i.e., an IP address and port number). The nameserver then forwards this information to the other participants, which use it to contact the machine. Some machines, however, may have multiple IP addresses at their disposal (e.g., one for each network interface), and the choice of which IP address to forward to the nameserver is non-trivial.

On the DAS-2 machines, the choice is relatively straightforward, since Fast Ethernet is the only network reachable from outside the clusters. The other networks (Myrinet and Gigabit Ethernet) are only usable when communicating within each cluster. This is reflected by the IP address, which are in the public range for Fast Ethernet and in the site-local range for Myrinet and Gigabit Ethernet. This is what one would expect, conforming to RFC 1918. Therefore, when running a single application on multiple DAS-2 sites, Ibis automatically selects the public IP-addresses of the machines, and publishes them using the nameserver. This ensures that all machines are able to communicate.

On Grid'5000, however, the situation is more complicated. There, some machines have multiple public IP address at their disposal, often in combination with one or more site-local addresses. Some machine have no public IP addresses at all, but only have one or more site-local addresses.

To complicate things further, not all combinations of addresses are able to connect. For example, the frontend machine of the Sophia Antipolis cluster has three public and two site-local IP addresses, while the compute nodes of the Orsay cluster only offers two site-local addresses. When trying to connect an Orsay compute node to the Sophia Antipolis frontend, we found that only one of the networks in the compute node was capable of reaching only two of the public addresses of the frontend machine. Interestingly, the public address of the frontend that was not reachable from the compute node was the address that is registered in DNS. The second network on the compute node could only be used to reach other compute nodes in the same cluster.

The example above illustrates the *reverse routing problem*. Normally, when a machine is trying to connect to a single IP address, a routing table decides which network is used for the outgoing connection. However, when the target machine has multiple IP addresses, the machine must also decide which IP address to connect to. As the example shows, this can be far from trivial, since there is generally not enough information available to make this decision.

During our PlugTest experiments, we where able to manually configure the Ibis system to reflect the network configuration of Grid'5000. This approach is very labour intensive, however. We are therefore improving the Ibis system such that it is capable of automatically discovering the network configuration of the testbed it is running on. This will allow applications to be run on complex system such as Grid'5000 with minimal help from the user.

In our final experiment, we attempted to combine the computational power of the DAS-2 and Grid'5000 systems, but failed. In general, machines in the Grid'5000 system are behind a firewall and are only capable of communicating within Grid'5000. Communication with the outside world, both incoming and outgoing, is not allowed. The only exceptions are some of the frontend machines, which allow incoming SSH connections and (very rarely) outgoing traffic.

As described in Section III-A.2, Ibis has support for firewall and NAT traversal, and is also capable of routing all communication through an external machine which is not behind a firewall. Unfortunately, all of these mechanisms failed in this experiment, because they all make the assumption that outgoing network traffic is allowed, which makes it possible to have an single external point which is reachable by all machines. This external point then acts as an "alternative communication channel" and is used by the machines to initiate reverse connection setup or TCP splicing, or, if these mechanisms fail, to forward application data itself.

Unfortunately, when combining the DAS-2 and Grid'5000

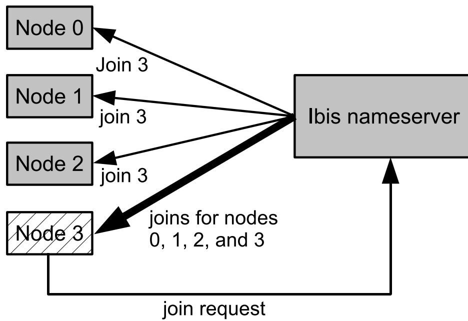

Fig. 6. Node 3 joins a running parallel computation.

systems this assumption failed. Most of the Grid'5000 machines do not allow network traffic out of the Grid'5000 system. As a result, no external machine could be found that was reachable by all participants. No machine outside of Grid'5000 could be used, since it would not be reachable by the compute nodes of Grid'5000. Similarly, no machine inside of Grid'5000 could be used, since it would not allow incoming traffic from the DAS-2. As a result, we were unable to complete the experiment.

A possible solution for this problem would be to allow multiple 'routing points', which are (partially) interconnected. For example, a routing point on the Sophia Antipolis frontend machine is reachable from all Grid'5000 machines, while a routing point on the DAS-2 frontend at the Vrije Universiteit is be reachable from all DAS-2 machines. These two routing points can be connected using an direct connection from the Sophia Antipolis frontend to the frontend at Vrije Universiteit (which is allowed) or an SSH tunnel in the opposite direction (which is also allowed). Grid'5000 and DAS-2 machines would then be able to communicate by passing all their data over these two 'routing points'. More complex scenarios which require three or more routing points are also possible. We are currently extending Ibis to support multiple routing points.

B. Scalability

1) Scalability Issues in the Ibis Nameserver: As explained above, Ibis uses a centralized nameserver to implement malleability support. For the contest, we ran N-Queens on 960 CPUs in parallel. We found that a centralized solution for the nameserver can easily support these numbers of machines. However, the malleability support in the nameserver was not efficient enough. Figure 6 illustrates what happens if a new node joins the computation (the mechanism is identical for nodes leaving). In this case, *node 3* is started and wants to participate. Therefore, it sends a message to the Ibis nameserver, requesting to join the computation. If the join is allowed, an acknowledgment of the join is sent to all machines that already participate, so their administration can be updated to include the newly joined machine (*node 3*). The machine that joined gets the acknowledgments for all machines that joined earlier, and for itself. This way, all machines have an identical administration of the machines participating, in the order they joined. Thus, joins (and leaves) are globally ordered.

It is clear that this straightforward implementation of malleability support does not scale well. With N nodes, N^2 join messages have to be sent in total. Moreover, a new connection is set up for each join message. The nameserver cannot keep the connections open, because that would not scale. A connection (in this case a TCP socket) consumes resources (file descriptors).

Thanks to the experience we gathered during the contest, we were able identify and solve this problem. We have implemented two improvements in the nameserver. Together, they completely solved the scalability problems we had.

First, we implemented message combining for join and leave notifications. If a node joins the computation, we don't immediately broadcast the join message. Instead, we delay sending the message for some (limited) time. During the delay period, more nodes might join the computation, and the join request messages can then be aggregated into a single message that is broadcast. We found that this mechanism works extremely well in practice. Typically, all nodes in a cluster join almost simultaneously, because a reservation system schedules the parallel job. These joins are now sent as a single message instead of N messages. With distributed supercomputing the number of clusters is typically much smaller than the total number of machines, so this optimization improves scalability considerably. For the contest, we used five clusters, with 960 CPUs in total. Thanks to the message combining optimization, only five (the number of clusters) instead of 960 join messages are broadcast.

Second, we implemented a mechanism that sends join messages to different machines in parallel. In the original implementation, we sent the join messages sequentially to all machines. As mentioned before, sending of a join message also includes connection setup. During the contest, we found that the join messages were sometimes stalled due to a slow or overloaded processor or network. Especially the connection setup could sometimes take a long time. Because the joins were sent sequentially, one slow node or network would also slow down the join messages to all other machines. The solution is straightforward: use multiple threads to forward the join messages asynchronously.

2) Scalability Issues in Satin: Satin uses a variant of random work stealing to do load balancing of the computation. Random work stealing has been proven to be optimal in space, time and communication for divide-and-conquer applications. However, using this algorithm, all machines communicate with all other machines, because they select random machines to steal work from. Whenever a new node joins the computation, the Satin runtime system gets a join notification. Satin sets up a connection to the new node, because it might be a candidate for work stealing in the future.

During the contest, we found that this solution results in a scalability problem. All machines in the computation receive a join notification at approximately the same time, and try to connect to the new machine. When many machines participate,

TABLE I
MACHINES USED FOR THE N-QUEENS CONTEST.

Site	CPUs
Orsay	426
Bordeaux	92
Rennes Opteron cluster	120
Rennes Xeon cluster	128
Sophia Antipolis	196
total	960

the joining machine is overwhelmed with connection requests. Moreover, many connection attempts fail, because the joining machine runs out of network resources. With TCP, a server socket that accepts connections has a limited *backlog*. This refers to the maximum queue length for incoming connection indications (a request to connect). If a connection indication arrives when the queue is full, the connection is refused. A typical value for the backlog is 50. So if many machines try to connect to the same machine at the same time, many connection attempts will fail, and have to be retried later.

Once the problem was identified, the solution was quite simple: we modified Satin to implement connection setup on demand. Connections are initiated when a machine becomes idle and starts stealing work from randomly selected other nodes. If there is no connection to the selected victim node yet, it is set up on the fly. We found that this relatively simple enhancement was enough to solve the scalability problems we encountered in Satin itself.

VII. RESULTS

We compared our parallel N-Queens (Java) program with the fastest known C version by Takahashi [10]. We found that our parallel version was on average only 0.5 % slower than the C version. The portability we gained by writing the program in Java thus has almost no measurable impact on performance.

During the contest, we were able to run the N-Queens application with on 960 nodes of Grid'5000 in parallel. As mentioned before, a parallel run using both Grid'5000 and DAS-2 failed due to connectivity problems. We ran N-Queens with N=22 in 1516 seconds (25 minutes). The list of machines we used is shown in Table I. We used 2 workers per node (1 for each CPU) and an additional node on the Orsay cluster as the Ibis name server.

The parallel Satin application spawned 4.7 million jobs in the 25 minutes it ran, and the Satin runtime system sent about 800,000 messages between the nodes to balance the load in the parallel computation. We measured the overhead of deploying and starting the jobs on all machines, as well as the communication cost in the Satin runtime system. In total, these overheads add up to 15% of the total run time. Therefore, 85% of the total run time is usefully spent in the application itself. Although our system is fault-tolerant, we encountered no crashed or failures during the contest.

VIII. CONCLUSIONS

We have described our experiences with integrating several Grid software components. The integration itself was relatively straightforward. Although the resulting system is quite complex and contains many components, writing and deploying parallel Grid applications is relatively easy thanks to the high level of abstraction that was achieved. Deployment is facilitated by the fact that all components are Java-based. Therefore, we were able to use a heterogeneous testbed without any recompilation. Moreover, the ProActvive deployment mechanism, which uses XML descriptions of the Grid resources, makes it very easy to start new jobs on the Grid.

We successfully ran experiments on 960 processors across Grid'5000, with a parallel N-Queens application written in Java. The performance of the application itself is virtually the same as the best known parallel C implementation of the algorithm. The parallel efficiency achieved was around 85%.

During the experiment, we identified several connectivity issues and problems related to parallel scalability. We have shown how we modified Ibis and Satin to solve most of these problems. We are currently working on solving one class of connectivity problems we discovered during the contest, the reverse routing problem. Important to note is that it was not necessary to modify the application itself, thanks to the high-level programming model we used.

REFERENCES

[1] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, "Enabling Applications on the Grid - A GridLab Overview," *International Journal of High Performance Computing Applications*, vol. 17, no. 4, pp. 449–466, 2003.

[2] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer, "The grid application toolkit: Towards generic and easy application programming interfaces for the grid," *Proceedings of the IEEE*, vol. 93, no. 3, pp. 534–550, march 2005.

[3] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere, "Interactive and descriptor-based deployment of object-oriented grid applications," in *The 11th International Symposium on High Performance Distributed Computing*, July 2002, pp. 93–102.

[4] R. D. Blumofe and C. E. Leiserson, "Scheduling Multithreaded Computations by Work Stealing," in *35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, November 1994, pp. 356–368.

[5] D. Caromel, W. Klauser, and J. Vayssiere, "Towards seamless computing and metacomputing in java," *Concurrency Practice and Experience*, vol. 10, no. 11–13, pp. 1043–1061, September-November 1998.

[6] A. Denis, O. Aumage, R. F. H. Hofman, K. Verstoep, T. Kielmann, and H. E. Bal, "Wide-area communication for grids: An integrated solution to connectivity, performance and security problems." in *Proc. of the 13th International Symposium on High-Performance Distributed Computing (HPDC)*, June 2004, pp. 97–106.

[7] "Grids@work – 2nd GRID Plugtests," 2005, http://www.etsi.org/plugtests/History/2005GRID.htm.

[8] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient Java RMI for Parallel Programming," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 6, pp. 747–775, 2001.

[9] R. V. v. Nieuwpoort, T. Kielmann, and H. E. Bal, "Efficient Load Balancing for Wide-area Divide-and-Conquer Applications," in *Proceedings Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, June 2001, pp. 34–43.

[10] K. Takahashi, "N Queens Problem (number of Solutions)," 2003, http://www.ic-net.or.jp/home/takaken/e/queen.

[11] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal, "Ibis: an Efficient Java-based Grid Programming Environment," in *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, USA, November 2002, pp. 18–27.