A High Performance Java Middleware with a Real Application *

Fabrice Huet
Vrije Universiteit
Amsterdam
The Netherlands
fhuet@cs.vu.nl

Denis Caromel
INRIA-I3S-CNRS
Sophia-Antipolis
France
Denis.Caromel@inria.fr

Henri E. Bal
Vrije Universiteit
Amsterdam
The Netherlands
bal@cs.vu.nl

Abstract

Previous experiments with high-performance Java were initially disappointing. After several years of optimization, this paper investigates the current suitability of such object-oriented middleware for High-Performance and Grid programming.

Using a middleware offering high level abstractions (ProActive), we have replaced the standard Java RMI layer with the optimized Ibis RMI interface. Ibis is a grid programming environment featuring efficient communications. Using a 3D electromagnetic application (an object-oriented time domain finite volume solver for 3D Maxwell equations) we have first conducted benchmarks on single clusters, including comparisons with the same application in Fortran MPI. Finally, Grid experiments have been conducted simultaneously on up to 5 different clusters.

Overall, the paper reports extremely promising results. For instance, a speed up of 12 on 16 machines (vs. 13.8 for Fortran), a speedup of 100 on 150 machines on a Grid.

1 Introduction

Our work falls in the trend of building high performance middleware with high levels of abstractions, especially in the context of objects, and the Java language.

The performance issues related to the implementation of RMI have been numerously reported and some solutions have been proposed. KaRMI [12] pioneered some work, with its fully integrated middleware, JavaParty [13]. Some important improvement were already achieved.

The solution proposed in this paper goes one step further. It is first based on a grid programming environment featuring an optimized communication layer, Ibis [16]. Both portable (in Java) and native protocols are proposed by Ibis, taking advantage of a portable layer (Ibis Portability Layer, IPL). The second element of our experiment is a high-level environment for the Grid, ProActive [4], featuring object-oriented group communications, and interactive XML deployments. Both these middleware can be considered as "application-level tools" which hide the low level details from the user. They can both use services provided by Globus[7] or other Grid middlewares. Such environment made it possible to develop a real and demanding application: an object-oriented solver for 3D Maxwell equations. This computational electromagnetism program relies on a time

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 0-7695-2153-3/04 $20.00 (c)2004 IEEE

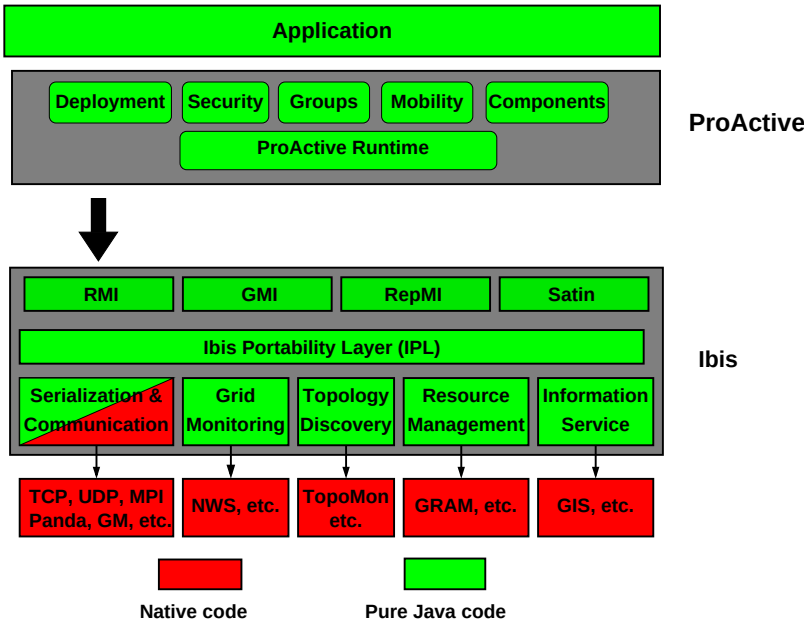

Figure 1. Global architecture of ProActive/Ibis

domain finite volume method [14]. The paper first reports the experiment on a single cluster, comparing the performance and speed up of the Java version with an equivalent program in Fortran using MPI. Multi-clusters benchmarks are also presented, executing on the DAS-2 Grid.

In the past, some MPI bindings for Java have been proposed [3, 10] as an alternative solution to benefit from both the object paradigm and the high-performance achievements. It is clear that the approach being used here could also be experimented with such MPI binding, just replacing the RMI layer with an optimized one, however, we will in this paper only consider a fully portable solution. Other previous work focused on the ease of deployment and usability. In [15], the authors uses a modified tuple space to offer an approach with various advantages like class downloading and platform independence through the use of Java. The tuple space paradigm, however, offers less imperative control.

Section 2 presents the overall middleware architecture with the ProActive functionality, the Ibis optimizations, and their integration. Section 3 provides a quick overview of the 3D electromagnetism application, focusing on the properties needed to understand its parallel performances. Section 4 details the experiments. After first analyzing the application memory usage, we compare the execution on ProActive/RMI, ProActive/Ibis, Fortran/MPI on a single cluster. Finally, Grid experiments on multi-clusters are reported.

2 Building a High level-High performance middleware

Rather than writing a new middleware which would offer a high level of abstraction while providing high performance, we decided to stack two existing one with respective good records in their fields of expertise. The main question was whether or not the overall result, shown in Figure 1, would meet our expectations.

In the remaining of the paper, we will refer to RMI as the Sun JDK implementation of RMI, and to Ibis as the Ibis RMI implementation.

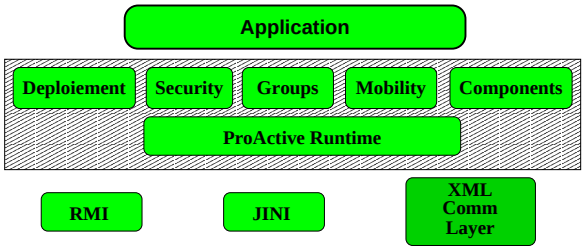

Figure 2. Application running on top of ProActive

2.1 The ProActive library

ProActive is a Java middleware for parallel, distributed and concurrent programming which features high level services like weak migration, group communication, security, deployment and components. The current implementation can use 3 different communication layers: RMI, Jini (for environment discovery) and a XML based protocol.

2.1.1 Base model

A distributed or concurrent application built using ProActive is composed of a number of medium-grained entities called *active objects*. Each active object has its own thread of control, and decides in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [6]. All active object running in a JVM belong to a *Node* which provides an abstraction for their physical location. At any time, a JVM hosts one or several nodes.

2.1.2 Group communications

The group communication mechanism[1] achieves asynchronous remote method invocation for a group of remote objects, with automatic gathering of replies.

Given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation: it is *typed group communications*. Furthermore, groups are automatically constructed to handle the result of collective operations, providing an elegant and effective way to program gather operations.

Using a standard Java class `A`, here is an example of a typical group creation:

```
// A group of type "A" is created,
// all its member are created at once on the specified nodes
// we specify here the parameters to be passed to the
// constructor of the objects
Object[][] params = {{...}, ... {...}};
A ag = (A) ProActiveGroup.newGroup("A", params, {node1, ..., node2});
```

Elements can be dynamically included into a typed group only if their class equals or extends the class specified at the group creation. Note that we do allow and handle *polymorphic* groups. For example, an object of class `B` (`B` extending `A`) can be included to a group of type A. However based on Java typing, only the methods defined in the class A can be invoked on the group.

A method invocation on a group has a syntax similar to a standard method invocation:

```
ag.foo(...); // A group communication
```

Such a call is asynchronously propagated to all members of the group in parallel using multithreading. Like in the ProActive basic model, a method call on a group is non-blocking and provides a transparent future object to collect the results. A method call on a group yields a method call on each of the group members.

An important specificity of the group mechanism is: the *result* of a typed group communication *is also a group*. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results, thus gathering results. The *wait-by-necessity* mechanism is also valid on groups: if all replies are awaited the caller blocks, but as soon as one reply arrives in the result group the method call on this result is executed. For instance in

```
V vg = ag.bar(); // A method call on a group with result
...              // vg is a typed group of "V",
vg.f(); // Also a collective operation, subject to wait-by-necessity
```

a new `f()` method call is automatically triggered as soon as a reply from the call `ag.bar()` comes back in the group `vg` (dynamically formed). The instruction `vg.f()` completes when `f()` has been called on all members.

2.1.3 Deployment Descriptors

The deployment descriptors [4] provide a mean to abstract from the source code of the application any reference to software or hardware configuration. It also provides an integrated mechanism to specify external processes that must be launched and the way to do it. The goal is to be able to deploy an application anywhere without having to change the source code, all the necessary information being stored in an XML *descriptor file*. An application using deployment descriptors has access to an API enabling it to queries its runtime environment for informations like the number of nodes available. A deployment file is made of three parts. The first one, VirtualNode, is used to declare nodes name that will be used in the source code of the application. The second part, Mapping, describes how the virtual nodes are to be mapped to virtual machines. Finally, in the Infrastructure part, we describe how these virtual machines will be created.

```
VirtualNode:
  jem3DNode

Mapping:
  jem3DNode --> VM1 , VM2

Infrastructure:
  VM1   -->  Local Virtual Machine
  VM2   -->  SSH host1  then RemoteVM
  RemoteVM -->  Local Virtual Machine
```

Figure 3. A simple deployment descriptor

An example of deployment file is given in Figure 3. For the sake of clarity, we have used a pseudo-code syntax instead of the less readable XML one. We have indicated in *italic* the symbolic names which are used as references in the file. These names are used to structure the descriptor and can be of arbitrary value. In **bold** references are the actual classes provided by ProActive. The application which will use this file will be able to use the symbolic name *jem3DNode* in the source code to access these resources. When used, this virtual node will be mapped onto two virtual machines, *VM1* and *VM2*, specified in the infrastructure part. The creation of these virtual machines is as

follows. The first one will be created locally. The second one will trigger a ssh connection to host1 and then perform the creation of a new local virtual machine there. In this part it is possible to specify various environment variables such as CLASSPATH to be used for the creation of the virtual machine.

2.2 The Ibis library

Ibis is a Java-based grid programming environment that allows efficient communication, using standards techniques that work "everywhere" and optimized solutions for special cases. It can use networks protocol provided by the JVM, like TCP/IP or UDP, but can also rely on low level protocols for high speed interconnect like GM, if available. The core of the library is the Ibis Portability Layer which ensures the interfacing between high level programming models and low level communication protocols or services. It is possible to write applications using either a source compatible RMI implementation or other models like Group Method Invocation (GMI[11]) or divide and conquer parallel programming (Satin[17]).

As noted in previous work [16], in order to increase the performance of RMI, it is possible to work on two different parts: the RMI protocol and the serialization. The current Sun implementation of RMI resends type information for each remote call, whereas Ibis caches this information on a connection basis, reducing the amount of data sent over the network. Two optimizations can be performed in order to reduce the cost of object serialization. First, the standard Java serialization uses reflection to convert objects to byte to be sent over the wire. It is possible to avoid this overhead by generating serialization code for each class that can be serialized, moving most of the cost from runtime to compile time. This generation is performed by a bytecode rewriter and does not require the source code of the application. Second, when rebuilding an object graph from a stream, one has to create objects without calling user defined constructors to avoid side effects. In standard Java, this task is performed by a private native call which turns out to be more expensive than a standard *new*. Ibis associates to each serializable class a *generator class* which role is to create objects by invoking a specially generated constructor, greatly reducing the object creation overhead.

We will, in this paper, focus on the 100% Java version of Ibis in order to retain portability.

2.3 Integration

One of the key decisions in the design of ProActive was that it should be independent from the communication layer. There are two issues that need to be tackled to do so. First, the references onto remote objects should not be dependent on the communication layer. From the application point of view, and to a lower extent from most of the middleware point of view, communicating using Ibis or RMI should not change anything in the source code. Second, the creation of remote references should be handled in a way such as, once again, it is fully transparent for the application or the middleware. The communication related parts of remote references are isolated using the proxy pattern [9], shielding other parts from communication code like exception handling (Figure 4 left). The creation of the references is handled through the abstract factory pattern [9]. Obtaining a remote reference requires requesting it from a factory whose implementation is loaded based on the requested communication protocol (Figure 4 right). This enables us to delay until runtime the choice of a communication layer. As a consequence, switching from RMI to Ibis based communications does not require any source code modification but simply the setting of a Java *property*.

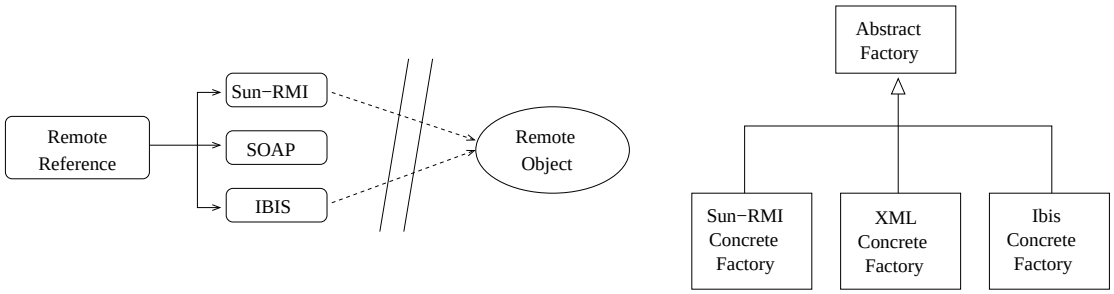

Figure 4. Proxied remote reference and concrete factories

3 A 3D computational electromagnetism application

In order to demonstrate the performance of our middleware, we have conducted extensive experiments using a parallel object oriented numerical simulation tool, Jem3D [2], for 3D electromagnetism applications. It is written on top of ProActive to benefit from high level features like deployment and group communication. In this section we will review the architecture of the application and its basics principle. We will also carefully study its distribution, drawing basic properties useful for the understanding of its parallel behavior.

3.1 Overview

Jem3D numerically solves the 3D Maxwell equations modeling time domain electromagnetic wave propagation phenomena. It relies on a finite volume approximation method designed to deal with unstructured tetrahedral discretization of the computation domain (see [14] for more details).

A standard test case for which an exact solution of the Maxwell equations exists (therefore allowing a precise validation of Jem3D with regards to both the numerical kernels and the parallelization aspects) consists in the simulation of the propagation of an eigenmode in a cubic metallic cavity. For this test case, the underlying tetrahedral mesh is automatically built in a pre-processing phase of a typical run of Jem3D. This mesh is simply obtained by first defining a Cartesian grid discretization of the cube and then, dividing each element of this grid in six tetrahedra. Ongoing Jem3D developments aim at handling general (irregular) tetrahedral meshes and complex geometries. In the sequential version of the application, the cube is divided into tetrahedra where local calculation of the electromagnetic fields is performed. More precisely, the balance flux is evaluated as the combination of the elementary fluxes computed through all 4 facets of the tetrahedron. After each step, the local values calculated in a tetrahedron are passed to its neighbors and a new local calculation starts. The general algorithm is given in Figure 5 and displays two phases in the running of the application: the *initialization* and the *calculation*. The complexity of the calculation can be changed by modifying the number of tetrahedra in the cube, which is done through the mesh size.

Definition 3.1 (Mesh size) *The mesh size of a calculation is a triple* $(m_1 \times m_2 \times m_3)$ *fixing the number of points on the* x*,* y *and* z *axis used for the building of the tetrahedral mesh.*

As a consequence, the higher the mesh size, the higher the number of tetrahedra and the more memory and computation intensive the application.

In the distributed version, the cube is divided into *subdomains* which can be placed over different machines.

Definition 3.2 (Domain, Subdomain and Border Face) *A domain is the overall volume of the calculation of the sequential application. A subdomain defines a part of the distributed applica-*

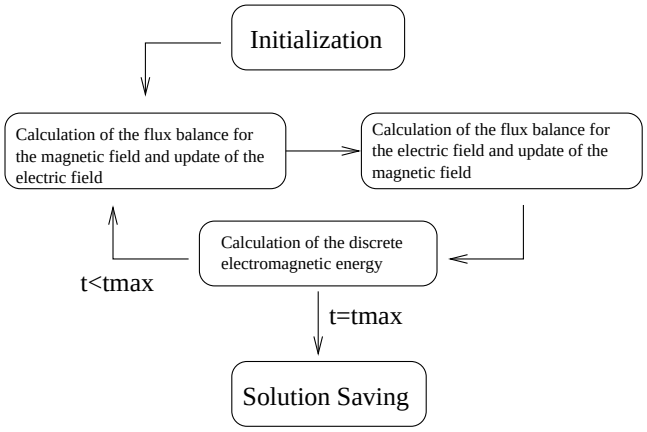

Figure 5. Simpli£ed view of the algorithm used in the jem3D application

tion whose elements are all located in the same address space. The faces of tetrahedra located on the boundary of a subdomain will be called border faces.

Inside a subdomain, the calculation behaves like in a domain except that tetrahedra located on the boundary of a subdomain have to communicate their results to ones located in a different address space through their border faces, using remote calls. We have decided to use, in our experiments, a cube as opposed to a more complex structure because it can be easily divided into identical volumes, thus avoiding the creation of a bottleneck between two particular subdomains. This gives us more control over the execution of our experiments. The division is defined as follow:

Definition 3.3 (Division of the calculation) *The division into subdomains is done by specifying a triplet* (a, b, c) *which indicates the number of subdomains on the* x*,* y *and* z *axis of the cube.*

Given the triplets $(m1 \times m2 \times m3)$ and (a, b, c), the code automatically builds a Cartesian grid decomposition of the cube (i.e a subdomain is a subcube) such that the interface between two neighboring subdomains is composed of triangular faces. Thus the resulting decomposition is a non-overlapping one such that vertices and triangular faces located on an interface are duplicated in the definition of each of the neighboring subdomains.

3.2 Architecture of the application

Jem3D is written completely in Java on top of ProActive and does not use either third-party or native libraries to perform the calculation. It can run on any standard Java platform where ProActive is available. Subdomains are *active objects* and communication between them is done through group communication mechanisms. Each communication between subdomains consists of a linked list whose elements, one for each border face, contain one array of 3 double. A special object, called the *collector* ensures the initial synchronization of the subdomains and can perform monitoring and steering of the application, if needed.

The application works in two steps: first it initialize itself and then the calculation starts. The initialization part consists in the deployment of the remote JVMs on the nodes of the cluster. Once the JVMs started, a subdomain is created on each of them. When the creation is over, each subdomain is linked to its neighbors through a group reference and reports back to the collector which then starts the computation.

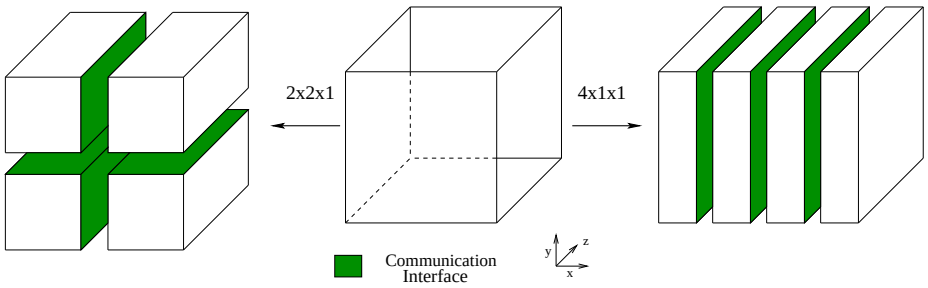

Figure 6. Two possible division of a cube into 4 subdomains

3.3 Optimal division of the calculation

As noted previously, the network communications take place on the edge of the subdomains so, assuming that the network is the bottleneck, we need to ensure that these frontiers are minimal, i.e. we need to find an optimal division for a given mesh size and a given number of nodes.

Definition 3.4 (Communication interface) *We will call communication interface the side of a subdomain where communication with another subdomain takes place. It is made of the set of half the border faces of all subdomains.*

We only take half of all the border faces because each border face located in a subdomain will be linked to a symmetric one in another subdomain, creating a single interface.

As illustrated in Figure 6, the division into $(2, 2, 1)$ will generate 4 interfaces whereas (4,1,1) only 3.

Proposition 3.1 *Given a cube with edge size n and a division (a, b, c), the number of communication interfaces between subdomains will be*

$$C = 3abc - bc - a(c + b)$$

and the area of all the interfaces will be

$$A = (a + b + c - 3) \times n^2$$

Proof : Considering a cube with edge size n with a $a \times b \times c$ divisions then there will be $(a-1) \times b \times c$ interfaces on the x-axis, $a \times (b - 1) \times c$ on the y-axis and $a \times b \times (c - 1)$ on the z-axis. Thus the total number will be $3abc - bc - ac - ab$. To evaluate the area of the interfaces, we will consider the three axis independently. There will be $(a - 1)$ divisions on the x-axis which will generate a total area of $(a - 1)n^2$. Applying the same reasoning to the other axis concludes the proof. ■

Although the area of the interfaces is an abstraction which is not used in the calculation, it can be used to calculate the number of tetrahedron communicating through the interface when the mesh is uniform (same value for all three axis).

Proposition 3.2 (Border faces on the interface) *Considering a cube with a mesh (m, m, m), and a division $a \times b \times c$, the total number of border faces (tetrahedra) located on the interface will be*

$$N = 2(a + b + c - 3) \times (m - 1)^2$$

Proof : Using the previous proposition, we are able to devise the area of the interface and it only remains to compute the number of tetrahedra per unit of area. Considering an area of size $m \times m$, calculating the number of border faces boils down to computing the number of triangles that can be constructed by dividing squares of size 1×1, which is $2(m-1)(m-1)$. Replacing this result in the area equation concludes the proof. ■

As an example, consider the following two divisions: $1 \times 1 \times 16$ and $2 \times 2 \times 4$. Although both of them will create 16 subdomains, the first one will require $30(m-1)^2$ border faces, as opposed to only $10(m-1)^2$ for the second.

Proposition 3.3 (Optimal division) *An optimal division of the calculation will be one which will minimize the number of border faces.*

In order to evaluate the effect of distribution over the calculation, we need to be able to link the sequential and parallel executions. In other word, given a sequential problem, what is the parallel version whose subdomains are equal in size to the sequential one.

Remark 3.1 (Relation between sequential and distributed problem size) *Considering a sequential experiment using a mesh of size $m_{s1} \times m_{s2} \times m_{s3}$, its domain will be equivalent to a subdomain of a distributed application with a mesh size of*

$$[a(m_{s1}-1)+1] \times [b(m_{s2}-1)+1] \times [c(m_{s3}-1)+1]$$

and a division (a,b,c), $\forall a,b,c \in \mathbb{N}$.

Proof : We proceed as with the previous demonstrations by considering first the x-axis. There are m_{s1} points used to build the tetrahedra. Consider an encapsulating domain using 2 identically sized subdomains, and recall from Definition 3.3 that points located on the interfaces will be common to both subdomains and so need to be accounted for only once. It is then clear that this domain should have $2 \times m_{s1} - 1$ points on the x-axis. Using induction, we can show that $\forall a \in \mathbb{N}$ the equivalent domain will have $a \times m_{s1} - (a-1)$ on the x-axis. Applying the same reasoning to the remaining axis concludes the proof. ■

4 Experiments

We will, in this section, study the time taken by our application to perform 100 loops of the *calculation* (see Figure 5) using different mesh size and number of nodes. We will compare the results obtained when using RMI and Ibis. As a reference, we will use a Fortran version of the same algorithm.

4.1 Experimental test bed

We have conducted our experiments on the Distributed ASCI Supercomputer 2 (DAS-2[1]). The nodes are composed of Dual Pentium III CPU running at 1Ghz with 1GB or more of memory, running RedHat Linux and linked with fast-ethernet. Two remote sites of the DAS-2 are connected through 10Gbits/s connections and the latency between them varies from $1ms$ to $3ms$. We have used the Sun JDK 1.4.2 for all our experiments. Although IBM's JDK one was found to be faster by up to 20%, its Garbage Collector did not work well with our application, sometimes inducing pauses time of more than 200 seconds. We have thus decided not to use it in our experiments until we could get smoother and more deterministic executions.

Because of the high memory usage, it was not possible to run Jem3D on a single node for high mesh values. Nonetheless, in order to be able to calculate the speed-up, we needed to extrapolate such a value, which could be done using the following remark:

[1] A detailed description of its architecture can be found at http://www.cs.vu.nl/das2/.

Remark 4.1 *When running on* 1 *node, the following relation holds*

$$\frac{Calculation\,Length}{Number\,of\,Tetrahedrons} \approx constant$$

Because all operations conducted on the list of tetrahedrons are of linear complexity.

Knowing the value of this ratio and the number of tetrahedrons for a given mesh size, we can calculate the calculation length for any mesh size. The duration we obtain would then be very close to a real experiment on a node with a *sufficient* amount of memory.

4.2 Memory usage

Before studying the impact of communications on the application, we first study its memory usage to know whether it is relying on virtual memory which decreases its performance. We have instrumented the source code to report the memory used at some key points of the execution, using the Java commands *Runtime.getRuntime().totalMemory()* and *Runtime.getRuntime().freeMemory()*. We wanted to measure first the total amount of memory used for a given mesh size and then, the overhead of the distribution, using the Remark 3.1.

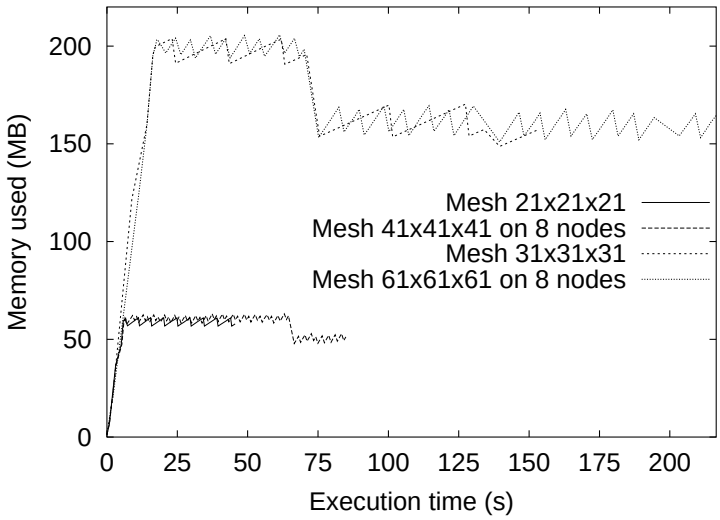

Figure 7. Memory used by Jem3D on 1 **node**

The results, depicted in Figure 7, clearly shows the two phases of our application. During a short period the memory used greatly increases because of the initialization and then, when performing the computation, the memory oscillates around an average value. Comparing the sequential (e.g. $21 \times 21 \times 21$) and distributed versions (e.g. $41 \times 41 \times 41$), we can see that there is only little memory overhead added by the distribution. The main difference being that since the experiments are longer because of the influence of the remote communications, the garbage collector has a stronger impact, especially after 60 seconds of execution. Using Ibis instead of RMI (not shown on the figure) led to an increase of 3% in the memory usage which we do not consider as significant.

4.3 ProActive/Ibis vs ProActive/RMI

First, we have conducted experiments on a part of the DAS-2 cluster in order to check the importance of the communication layers in an homogeneous environment. All experiments were run with the same version of the application on top of ProActive, using either RMI or Ibis. The

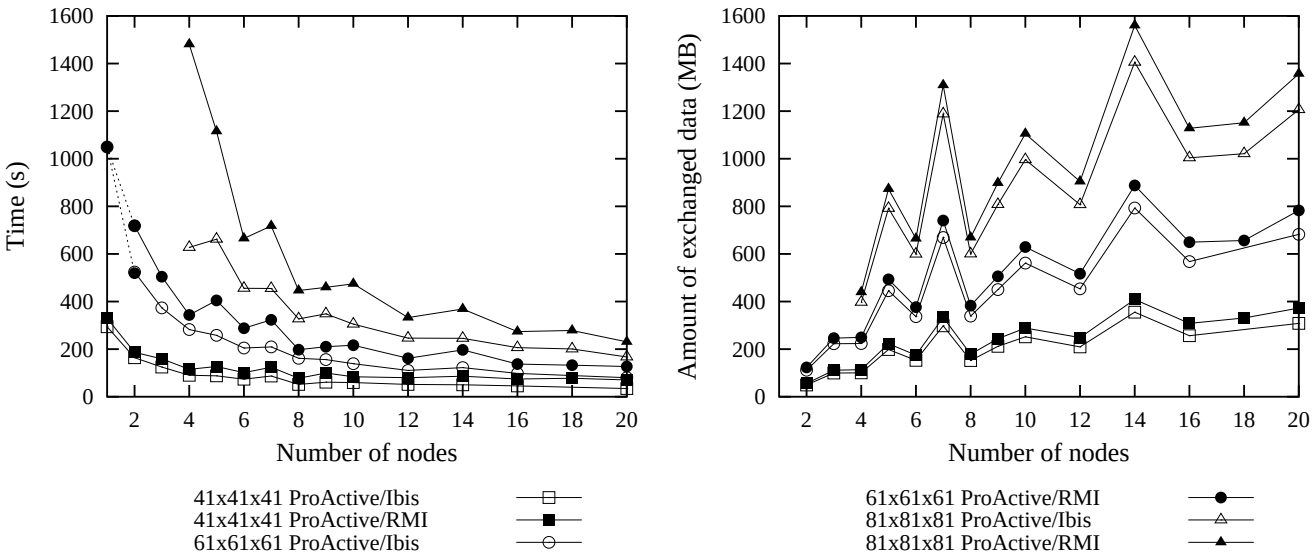

Figure 8. ProActive/RMI vs ProActive/Ibis

results are depicted in Figure 8. The left hand part shows the execution time as a function of the number of nodes used in the computation, for various mesh values. We have indicated the interpolated time for 1 node using a dashed line. The time for $81 \times 81 \times 81$ on 1 node, which is $2488s$, is not depicted on the figure. The right hand side of the figure shows the total amount of data sent over the network as a function of the number of nodes. Sometimes, increasing the number of nodes will increase the execution time and the amount of data exchanged. Although we use the optimal division for the calculation, as described in Proposition 3.3, it does not mean that increasing the number of nodes will have a matching effect on the number of border faces. Indeed, for $41 \times 41 \times 41$ on 4 nodes, the optimal division will be $(1, 2, 2)$ which will generates 6400 border faces. Considering 5 nodes, the optimal division will be $(1, 1, 5)$ with 12800 border faces. In this case, increasing the number of nodes involved in the computation will almost double the amount of data sent over the network, which is clearly visible on the figure. As we can see, there is an important difference in the execution time when using Ibis or RMI. Using the former, the programs executes between 20 and 50% faster. The amount of data sent over the network is 10% lower using Ibis, however, this, in itself, cannot explain the difference in execution and we believe this result is also achieved because of the faster serialization, which remains an important bottleneck in RMI.

We have indicated the obtained speedup for the application on Figure 9, using the interpolated value for 1 node when not available. Again, using Ibis instead of RMI increases dramatically the speedup from around 8 for $81 \times 81 \times 81$ on 20 nodes to 14.

4.4 Comparison with the Fortran/MPI version

The original algorithm used in Jem3D was actually exploited first in a Fortran/MPI version called EM3D. We wanted to perform a comparison in order to provide us with some insight on the relative performance of these two versions. The aim is not to compare Java with Fortran but rather two implementations of the same algorithm using different architectures and see whether the distribution of the calculation can be as efficient in Java than in Fortran/MPI.

We only had access to a compiled version of EM3D and thus were not able to instrument it to perform fine measurements like memory usage. One important difference in the design was that

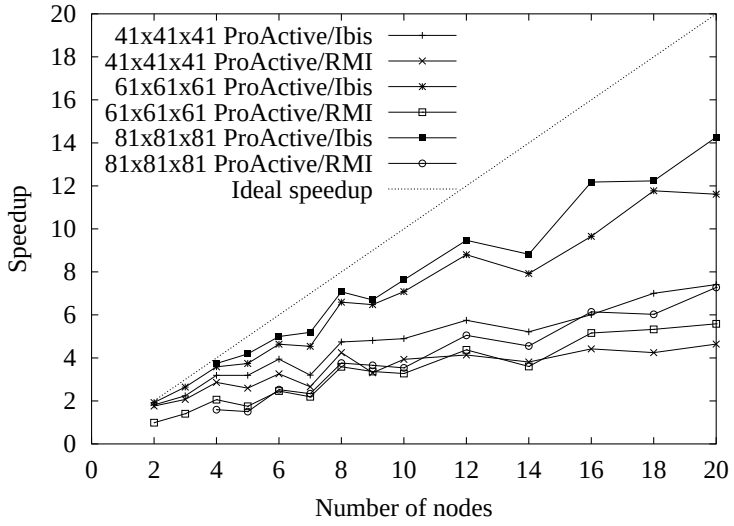

Figure 9. Speed-up using ProActive/RMI and ProActive/Ibis on a single DAS-2 cluster

EM3D used fixed size data structures to be fine tuned to the available memory and avoid swapping. As a result, source code modification is needed to reach arbitrary high mesh values. Moreover we were not able to run it on the DAS-2 cluster because of libraries issues. The following experiments were thus run on a cluster made of Dual Pentium III running at 933Mhz with 512MB of memory and linked through 100Mb/s switched network, located in Sophia Antipolis, France. The starting

	Time		Memory		Java/Fortran	
Mesh	Java	Fortran	Java	Fortran	Time	Memory
21x21x21	45s	18.9s	78M	59M	2.38	1.32
31x31x31	150s	65s	224MB	164MB	2.30	1.36
41x41x41	387s	156s	483MB	366MB	2.48	1.31

Table 1. Java vs Fortran on 1 **node**

point for our comparison will be the execution time and memory usage of the sequential versions of the algorithm. Since we couldn't modify the Fortran version, the memory, for both applications was measured using *top* under Linux and the results are shown in Table 1. The ratio between the execution time of the Java and Fortran versions is around 2.38 which we believe is a good, taking into account the differences existing between the two versions and previously published benchmarks [5] that reported a ratio between 1.24 and 2.70 for some applications. The memory usage ratio is around 1.33 which, while still being acceptable, can probably be improved since it has not been, to the best of our knowledge, a goal in the design of Jem3D. We have performed some simple optimizations on the Java code, like creating dynamic data structures with the final size instead of having them grow when needed, thus avoiding the creation of temporary variables and overprovisionned structures.

Considering now the distributed version, whose results can be seen in Figure 10, the Java version is, to no surprise, still slower than the Fortran one. However, where the ratio of execution time (Java/Fortran) reached with RMI was almost all the time around 3.5, we achieved a much better performance with Ibis, lowering it to around 2.5, very close to the sequential one. The speedup achieved is, in both case, lower than the Fortran one, however, as we increase the mesh size, Ibis scales better, increasing its speedup to as much as 12 on 16 nodes, compared to 8.8 with RMI and 13.8 with Fortran.

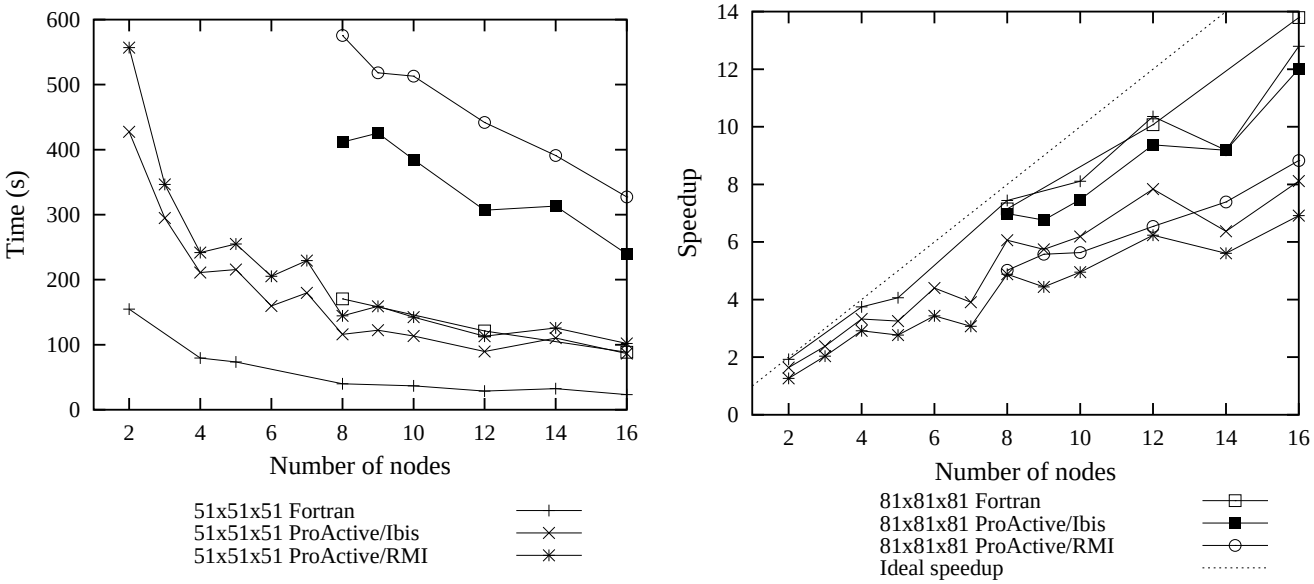

Figure 10. Execution time and speedup for the Java and Fortran versions

The Sun implementation of RMI seems to be a limiting factor for the performance of distributed applications. Using a different communication layer, it is possible to increase the performance and to achieve very good speedups, very close to those obtained using Fortran/MPI.

4.5 Multi-cluster experiments

Following [8] stating that a Grid is a system that coordinates resources without centralized control, using standard protocols and interfaces to achieve a non trivial quality of service, we experimented on the DAS-2 Grid. Taking full advantage of it, we were able to perform multi-cluster experiments by requesting nodes on each of its parts. As an example, the distribution for an experiment using 8 subdomains distributed on 4 clusters (fs0, fs1, fs2, fs3) is shown in Figure 11. Each subdomain has 3 neighbors with which it exchanges locally calculated values. They all report to the Collector but, for the sake of clarity, we did not indicate all the communications in the figure. In the current version we do not take advantage of the network architecture and it might happen that the repartition of the calculation is sub-optimal, having multiple communications over slow-links. However we did not find any conclusive proof in our experiments that such a situation occurred. Indeed, the application demonstrated a very good speedup, even when distributed over multiple clusters.

In order to have the application run, we had to perform only the following two steps: (*i*) install the needed classes on each file system [2], (*ii*) write a descriptor file which describes the architecture used for the experiments. The descriptor files basically states that: Jem3D will run on 5 cluster, 1 local (fs0), 4 remotes (fs1, fs2, fs3 and fs4). The deployment process will start from fs0. There, a PBS command will be issued to book some nodes. To request nodes on the other clusters, first an ssh connection will be established with each one of them and then, a PBS command will be issued. Once the nodes granted by the scheduler, JVMs will be created as specified in the *descriptor file*, along with a ProActive Runtime which will allow the application to deploy itself.

[2]It is not necessary to install the application classes since they can be downloaded using dynamic class loading at the expense of a slower startup, only the middleware ones are needed.

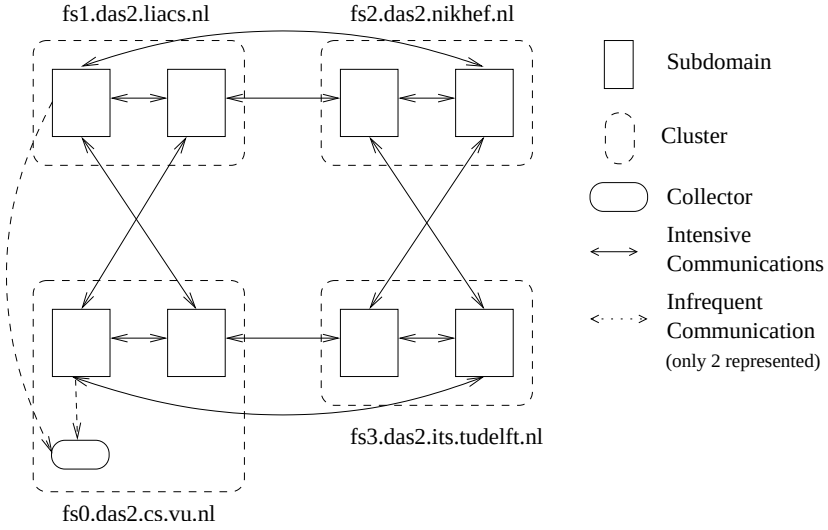

Figure 11. Distribution of the calculation on the DAS-2 Grid with 8 subdomains

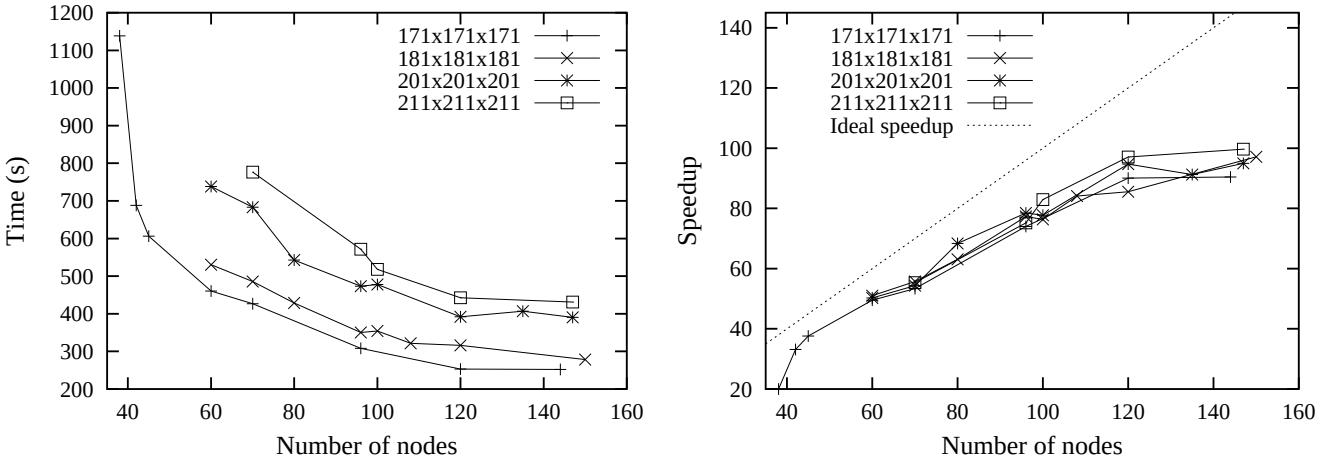

Figure 12. Benchmarks with all available nodes on the DAS-2 Grid

All these operations are usually handled in scripts or various programs and often requires some human intervention. Using the deployment mechanism, this can be done automatically from *inside* the application which, as a side effect, provides a very useful feature. Since the application can check, using some ProActive API, the number of nodes which have been allocated, it can adapt its execution to the available resources. As an example, if Jem3D requests 20 nodes on each of the four clusters but get only 60 out of 80 because of co-allocation issues, it could either reduce the size of its calculation or postpone it until enough resources are available, making it self-adaptive to its environment.

The results of this experiments are shown in Figure 12. Running on a single cluster, the maximum efficiency reached was 96% on 2 nodes with a mesh of $61 \times 61 \times 61$ whereas, on multiple clusters, we had 85.4% on 80 nodes with $201 \times 201 \times 201$, which is very good considering that we did not have any control on the distribution of the elements of the calculation.

5 Conclusion

In this paper we have demonstrated how to take advantage of two complementary middlewares to offer a platform with high-level abstractions and high performance communications while still remaining 100% Java. The result is an efficient and highly flexible programming environment suited for cluster and grid programming.

Extensive experiments on the DAS-2 cluster have shown that, in real applications, the current implementation of Sun-RMI greatly limits the overall speed and scalability, even with simple data structures sent over the network like linked lists. Although this had been previously demonstrated in micro-benchmarks, we believe this is the first time it has been exhibited in a full-scale application. Comparing the ProActive and a Fortran/MPI versions of Jem3D, we have shown the scalability of Java, achieving, on 16 nodes, a speedup of 13.8 for Fortran, 12 for ProActive/Ibis and only 8.8 for ProActive/RMI.

The use of a deployment mechanism accessible to the application through both XML and an API allows the construction of complex configurations spawning multiple clusters. Moreover, the configuration can be very dynamic, adapting the deployment to the actual number of nodes obtained. It was indeed possible to run the 3D electromagnetic application on 150 nodes, obtaining a speedup of around 100.

Although we have, in this paper, advocated a Java middleware to benefit from the "run everywhere" portability, we believe that the use of native code in limited and carefully chosen parts of the middleware, like low level communication drivers, can bring important benefits with limited drawbacks. We plan to address this, using Ibis' multiple protocols, in a future work.

Acknowledgments

We would like to thank Ceriel J.H. Jacobs and Rutger Hofman, from the Vrije Universiteit (The Netherlands), for their help with this work. We are also grateful to Stephane Lanteri and Said El Kasmi, from INRIA Sophia-Antipolis (France), for their earlier comments. This work was partly funded by an INRIA grant.

References

[1] L. Baduel, F. Baude, and D. Caromel. Efficient, Flexible, and Typed Group Communications in Java. In *Joint ACM Java Grande - ISCOPE Conference*, pages 28–36, Seattle, 2002. ACM Press.

[2] L. Baduel, F. Baude, D. Caromel, C. Delbe, S. El Kasmi, N. Gama, and S. Lanteri. A parallel object-oriented application for 3D electromagnetism. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.

[3] M. Baker, B. Carpenter, S. Ko, and X. Li. mpijava: A java interface to mpi. Presented at First UKWorkshop on Java for High Performance Network Computing, Europar, 1998.

[4] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications. In *11th IEEE International Symposium on High Performance Distributed Computing HPDC-11*, 2002.

[5] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking java against C and Fortran for scientific applications. In *Java Grande*, pages 97–105, 2001.

[6] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. In *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM, Lyon, France*, August 1996.

[8] Ian Foster. What is the grid? a three point checklist. GridToday, July 2002.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] V. Getov, P. Gray, and V. Sunderam. MPI and Java-MPI: Contrasts and comparisons of low-level communication performance. In *Proceedings of Supercomputing '99, Portland, Oregon*, 1999.

[11] Jason Maassen, Thilo Kielmann, and Henri E. Bal. GMI: Flexible and efficient group method invocation for parallel programming. LCR '02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computer, 2002.

[12] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for java. *Concurrent: Practrice and Experience*, 12(7):495–518, 2000.

[13] M. Philippsen and M. Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.

[14] S. Piperno, M. Remaki, and L. Fezoui. A nondiffusive finite volume scheme for the three-dimensional maxwell's equations on unstructured meshes. *SIAM Journal on Numerical Analysis*, 39(6):2089–2108, 2002.

[15] H. De Sterck, R. S. Markel, T. Phol, and U. Rde. A lightweight java taskspaces framework for scientific computing on computational grids. In *Proceedings of the 2003 ACM symposium on Applied computing, Melbourne, Florida*, 2003.

[16] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency and Computation: Practice and Experience*, to appear.

[17] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Thilo Kielmann, and Henri E. Bal. Satin: Simple and efficient java-based grid programming. *Accepted for publication in Journal of Parallel and Distributed Computing Practices*, 2004.