International Journal of Parallel Programming, Vol. 35, No. 6, December 2007 (© 2007)
DOI: 10.1007/s10766-007-0044-3

Asynchronous Typed Object Groups for Grid Programming

Laurent Baduel,[1,2] Françoise Baude,[1] and Denis Caromel[1]

Received May 31, 2006; accepted March 23, 2007

This article presents an object-oriented mechanism to achieve group communication in large scale grids. Group communication is a crucial feature for high-performance and grid computing. While previous work on collective communications imposed the use of dedicated interfaces, we propose a scheme where one can initiate group communications using the standard public methods of the class by instantiating objects through a special object factory. The object factory utilizes casting and introspection to construct a "parallel processing enhanced" implementation of the object which matches the original class' interface. This mechanism is then extended in an evolution of the classical SPMD programming paradigm into the domain of clusters and grids named "Object-Oriented SPMD". OOSPMD provides interprocess (inter-object) communications via transparent remote method invocations rather than custom interfaces. Such typed group communication constitutes a basis for improvement of component models allowing advanced composition of parallel building blocks. The typed group pattern leads to an interesting, uniform, and complete model for programming applications intended to be run on clusters and grids.

KEY WORDS: Java middleware; Group communication; object-oriented parallelism; SPMD programming.

[1]INRIA – I3S CNRS, University of Nice Sophia-Antipolis, 2004 route des Lucioles, B.P. 93, Sophia-Antipolis Cedex, 06902, France. E-mail: laurent.baduel@sophia.inria.fr
[2]To whom correspondence should be addressed.

1. INTRODUCTION

1.1. Context

Many distributed applications deal with intensive computations and management of huge amounts of data which have to be transferred and processed on multiple resources, e.g. on computing grids, in order to improve the performance or scale the number of processing units involved in the application. Typical examples include simulations applied to scientific and engineering fields, or data acquisition and analysis from distributed measurement instrumentation and sensors.

In recent years many grid middleware platforms and toolkits have been developed (e.g. Globus,[1] Legion,[2] Unicore,[3] Condor-G,[4] HiMM[5]). These middleware platforms typically use unicast communication mechanisms implemented on top of reliable protocols. Performance of programs on distributed memory parallel hardware infrastructures are highly dependent of the efficiency of interprocess communications, and performance could be improved by relying on one-to-many or many-to-many communication mechanisms.[6] In the particular case of grids, some experiments have already proven that applications can strongly benefit from collective communication mechanisms.[7,8] Overall, this requires that the decision to apply collective communication mechanisms is taken by the programmers (be they programming the end-user application or the middleware itself). The challenge is thus to provide both expressive and efficient mechanisms for collective communications, while minimizing the burden on programmers in using them. This is difficult in a grid environment as they are heterogeneous by nature. In the past few years the interest in using Java for both portable and high-performance computing has increased. Java provides an object-oriented programming model with support for concurrency, garbage collection, and security. It features multithreading and *Remote Method Invocation* (RMI)[9] (an object-oriented version of *Remote Procedure Call* (RPC)[10]). In the Java world the RMI mechanism is the standard point-to-point communication mechanism and is adequate for client–server interactions via synchronous remote method calls. In a high-performance computing context, however, asynchronous and collective communication mechanisms are necessary, making RMI unsuitable. Programming high-performance applications requires the definition and the coordination of parallel activities, hence a library for parallel programming should provide not only point-to-point but collective communication primitives on groups of activities. This calls for a way to represent and manipulate such groups. For instance, according to the Object Group design pattern,[11] a group is a local surrogate for a set of objects distributed across networked machines to which the execution

of a task can be assigned. The Object Group pattern specifies that when a method is invoked on a group, the runtime system sends the method invocation request to the group members, waits for one or more member-replies on the basis of a policy, and returns the result back to the client. Groups are usually dynamic, meaning the set of group members can continuously change.

1.2. Contribution and Structure of the Article

This article advances the current state of the art with a new *high level* and *transparent* group communication mechanism for object-oriented parallel and distributed programming environments. Other attempts to provide a solution for both parallel and distributed programming within an object-oriented approach exist.[12–14] Our design, even if it has been achieved in the context of the *ProActive* library goes farthest with respect to the fulfillment with object-orientation. Shortly put, any distributed object can become a member of a typed group, without additional constraint; a communication towards a group is designed and appears as a smooth extension of a point-to-point remote method invocation on a distributed object. As a consequence of this design, getting a new SPMD programming style but fully object-oriented is only a small step away.

ProActive[15] is a 100% Java library for parallel, distributed, concurrent computing with security and mobility. RMI is by default used as the transport layer. Besides remote method invocation services, *ProActive* features transparent remote Active Objects, asynchronous two-way communications with transparent *futures*, high-level synchronization mechanisms, migration of Active Objects with pending calls and an automatic localization mechanism to maintain connectivity for both "requests" and "replies". This work reports on an extension of standard point-to-point communications, as previously implemented in *ProActive*, to a model of groups of Active Objects with group communication mechanisms.

At the programming level, groups can ease software development since they simplify the implementation of some high-level computing models, such as master-slave, pipeline, and work-stealing. At the communication level groups can reduce the communication overhead for several reasons. First, the delivery of the same content to a collection of receivers can benefit from the group abstraction since specific optimizations can be applied even if the underlying transport layer is based on unicast communication. A first example of optimization is the serialization of parameter objects. Indeed, the network transfer of objects requires serialization before sending them. Since serialization may take significant processing time, sending the same object to the members of the group is significantly

improved if the same serialized copy of the object is used for a unicast transfer to each member. Using a thread pool to perform parallel method invocations constitutes a second optimization. Finally, group communication can be implemented on top of a multicast transport layer for better performance. This means the default communication layer (RMI) can be changed. Indeed,[16] proves that it is possible to use a multicast communication layer. In particular, it presents the performance of the *ProActive* group communication mechanism, the subject of the present article, using the Tree-based Reliable Multicast Protocol (TRAM) included in the Java Reliable Multicast Service (JRMS v1.1). Implementation specifics are not the focus of this work, therefore optimizations will not be discussed further. This article focuses on presenting the principles underlying a "typed group" mechanism, and its suitability in the programming of a wide range of parallel and distributed applications and tools. In particular, we present in detail one application of the typed group mechanism to define a new SPMD programming style we name Object-Oriented SPMD.

The article is structured as follows: Section 2 reviews the main parallel programming features provided by the ProActive platform. This is the basis for the typed group communication mechanism defined in Section 3. Section 4 then describes how a fulfilled object-oriented Single Program Multiple Data (SPMD) parallel programming model is built, grounded on the typed group mechanism. Section 5 describes a few cases where this mechanism shows significant benefits, thus justifying its relevance. Section 6 summarizes the work and presents conclusions.

2. THE *ProActive* MIDDLEWARE

ProActive is an LGPL Java library for parallel, distributed, and concurrent computing also featuring mobility and security in a uniform framework. With a small core set of simple primitives, *ProActive* provides a comprehensive API which simplifies the programming of applications that are distributed on Local Area Networks (LANs), clusters, or grids.

As *ProActive* is built on top of the standard Java API,[3] it does not require any modification to the Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine.

2.1. Base Model

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each

[3]Mainly Java RMI and the Reflection API.

active object has its own thread of control and is granted the ability to decide in which order to respond to the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity*.[17] When a method is called on an active object, it returns immediately (as a thread cannot execute methods on another active object). A future object, which is a placeholder for the result of the methods invocation, is returned. From the point of view of the caller, there is no difference between the future object and the object that would have been returned if the same call had been issued to a local, non-active object. The calling thread can then continue executing its code as if the call had been performed synchronously. The role of the future object is to block this thread if it invokes a method on the future object and the result has not yet been set (i.e. the active object on which the call was received has not yet performed the call and placed the result into the future object): the wait-by-necessity mechanism is this inter-object synchronization policy. Contrary to classical RMI, all method call parameters sent to an active object are passed by deep-copy. A deep-copy consists of cloning an object along with the objects to which it refers. If either of these referred objects themselves contain objects then a deep copy copies those objects as well, and so on until the entire graph is traversed and copied. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee.

Mobility of active objects is the ability to relocate, dynamically at runtime, the components of a distributed application. The *ProActive* library provides a way to migrate any active object from any JVM to any other through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call. Migration is a useful to provide load balancing or to build agent-oriented applications in which autonomous agents navigate through the network to collect data and finally return to present them.

2.2. Mapping Active Objects to JVMS: Nodes

Another extra service provided by *ProActive* (compared to RMI, for instance) is the capability to *remotely create remotely accessible objects*. For that reason, there is a need to identify JVMs, and to add a few services. Nodes provide those extra capabilities: a node is an object defined in *ProActive* whose aim is to gather several active objects into a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or more nodes. The traditional way

to name and handle nodes is to associate them with a URL giving their location, for instance `rmi://lo.inria.fr/node`. Let us consider some Java class named `MyClass`. The instructions:

```
MyClass a1 = (MyClass) ProActive.newActive ("MyClass",
params, myNode);
MyClass a2 = new MyClass(params)
```

create a new active object `a1` of type `MyClass` on the JVM identified by "`rmi://lo.inria.fr/node`", and a regular Java object `a2` of type `MyClass`. The object `a1` can be compared to `a2` which has been locally instantiated using traditional Java syntax. To any down stream processing code expecting an object of class `MyClass`, `a1` and `a2` will appear to be identical. Now all calls to that remote active object will be asynchronous, and subject to *wait-by-necessity*:

```
a1.foo (...); // Asynchronous call
v = a1.bar (...); // Asynchronous call
...
v.f (...); // Wait-by-necessity: wait until v gets its
value
```

Compared to traditional futures, *wait-by-necessity* offers two important features: (1) futures are created implicitly and systematically, (2) futures can be passed to other remote processes.

Note that an active object can also be bound dynamically to a node as the result of a migration. In order to help in the deployment phase of *ProActive* components, and make the code more independent from the target hardware configuration, an external XML file can instead be used to make the association between logical node names used in the code and their real location: this is the concept of virtual nodes as entities for mapping active objects, as introduced in Ref. 18. Those virtual nodes are described externally through XML-based deployment descriptors which are then read by the runtime system when the ProActive environment is started. 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 those descriptors. As such, deployment descriptors provide a means 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 (e.g. JVM) that must be launched and the way to do it.

3. GROUP COMMUNICATION

The RMI model only provides synchronous point-to-point communication. The communication model of *ProActive* enhances RMI by adding

asynchronism, *futures*, automatic synchronization and *wait-by-necessity*. Despite these improvements, parallel and distributed applications often need more advanced communication strategies, such as group communication. These applications need the ability to combine many objects (remote or local) to communicate with them in one go. Such operations must be more efficient than replication of simple point-to-point communication, while still preserving similar behavior. In addition we want to maintain transparency regarding group communication, not only at the sending of method invocations but also while the gathering of replies.

3.1. The Typed Group Model

Alternate approaches for parallel and distributed computing in Java include the use of more dedicated and explicit parallel programming frameworks, such as parallel and distributed collections[16] which hide the presence of parallel processes, or implementations of MPI-like libraries in an SPMD programming style.[20] As defined in Ref. 21, the group mechanism we propose is more general, as it enables the construction of parallel programming models, while providing group communication to distributed applications originally not designed to be parallel.

The group communication mechanism is built on the *ProActive's* simple mechanism for asynchronous remote method invocation with automatic *futures* for collecting replies. As this last mechanism is implemented using standard Java, the group mechanism is itself platform independent: it requires no changes to the JVM, no preprocessing or compiler modification, as with the rest of *ProActive* the library. A group communication operation must be thought of as a replication of multiple *ProActive* remote method invocations towards all the active objects within the group. Of course, the aim is to incorporate optimizations into the group mechanism implementation in such a way as to achieve better performance than a sequential execution of N individual remote method calls. In this way, our mechanism is a generalization of the remote method call mechanism already used in *ProActive*, built upon RMI, however nothing prevents an implementation of this group communication strategy from using other transport strategies.

The availability of such a group communication mechanism simplifies the programming of applications with similar activities running in parallel. It is natural to group together similar activities because they are likely to receive the same data or the same instructions through method invocations. In an object-oriented framework, this idea of similar activities is represented by all members of a group implementing a common interface, or extending a common superclass. Indeed, from the programming point of view, using a

group of active objects of the same type, subsequently called a *typed group*, takes exactly the same form as using only one active object of this type. The multi-communication to each member of a group is abstracted from the code and is implicit for a grouped active object — only the functional (i.e. common interface) aspects are then visible in the code.

The construction of such groups is possible due to the *ProActive* library's use of reification techniques: the class of an object that we want to make "active", and thus remotely accessible, is reified at the meta-level, at runtime. Method calls to an active object are executed transparently through a stub which is type compatible with the original object. The stub's role is to consider and manage the call as a first class entity and apply to it the required semantics: if it is a call to a single remote active object, then the standard asynchronous remote method invocation of *ProActive* is applied; if the call is to a group of objects, then the semantics of group communication are applied.

3.2. Summary of the Group Communication API

Given a Java class, one can initiate group communications using the standard public methods of the class together with the classic "dot" (i.e. `object.method()`) notation; in this way group communications remain *typed*. Furthermore, groups are automatically constructed to handle the result of collective operations, providing an elegant and effective way to program gather operations.

Here is an example of a typical group creation using the Java class `MyClass` presented earlier:

```
// A group of type "MyClass" and its 3 members are created at once on the
// specified nodes, parameters are specified in params,
Object[][] params = {{...},{...},{...}};
MyClass agroup = (MyClass) ProActiveGroup.newGroup("MyClass", params, {node1,node2,node3});
```

Elements can be included in a typed group only if their class matches or extends the class specified during group creation. Note that we allow *polymorphic* groups. For example, an object of class `NewClass` (`NewClass` extending `MyClass`) can be included in a group of type `MyClass`. However based on Java typing, only the methods defined in `MyClass` can be invoked on the group. Groups can also be dynamically modified, adding or removing members, or extracting a sub-group from a typed group.

Method invocation on a group has an identical syntax to standard method invocation:

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

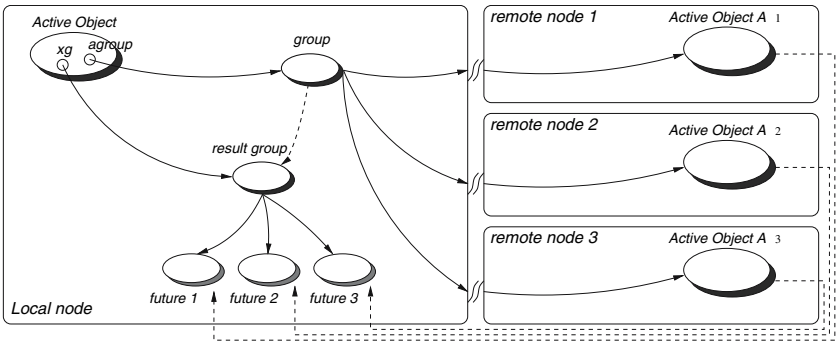

Fig. 1. Method call on group.

Such a call is propagated to all members of the group using multithreading: a method call on a group transforms into a method call on each of the group members. If a member is a *ProActive* active object, the method call will be asynchronous, and if the member is a standard Java object, the method call will be a standard Java method call (within the same JVM). By default, the parameters of the invoked method are broadcast to all the members of the group.

An important observation of the group mechanism is that the *result* of a typed group communication *can also be a group*. The resulting group is transparently built at invocation time, with a future for each member of the caller group. It will be dynamically updated with the incoming results, thus gathering results as shown in Fig. 1 (see *"result group"*). The *wait-by-necessity* mechanism is also valid on groups: if all replies are waiting then the caller blocks, however as soon as one reply arrives in the result group the method call on this result is executed, such as in the following example:

```
AnotherClass xg = agroup.foo(); // xg is a typed group of
"AnotherClass"
xg.bar();     // This is also a collective operation
```

A new `bar()` method call is automatically triggered on a reply from the call `agroup.foo()` as soon as this reply comes back in the dynamically formed group `xg`. The instruction `xg.bar()` completes when `bar()` has been called on all members: this constitutes a *local* synchronization point from the point of view of the initiator of the group method call, i.e., certifying that all peers in the group agroup have executed the method `foo()`. Another remark is that collected results which have been *gathered* through the `xg` group can subsequently be merged. This is similar to achieving a global reduction. The reduction operator can be any

user defined method (such as `bar()` in the above example). Moreover, the operator can be applied as soon as each result comes back. Even if the reduction operation is not executed in parallel, its runtime cost can be overlapped by the transmission delay of the not-yet-arrived results. Besides, the mechanism provides some primitives to enable a finer treatment of synchronizations. For instance: `waitOne, waitN, waitAll`, etc. Here is an example:

```
AnotherClass xg = agroup.foo(); // To wait and capture the
first result:
AnotherClass x = (AnotherClass) ProActiveGroup.waitAndG-
etOne(xg);
```

Regarding the parameters of a method call to a group of objects, the default behavior is to broadcast them to all members. Sometimes only a specific subset of the parameters, often depending on the rank of the member in the group, may actually be necessary for the method execution, making the remainder of the parameters unnecessary. It therefore is necessary to consider a standard mechanism to easily allocate different parts of the parameter set to different members of the group.

The typed group communication provides a way to scatter the parameters of a method call between the members of the group. A common way to achieve the scattering of global parameters is to use the member's rank within the group to select the appropriate parameter subset for the member's method execution. The natural extension of this idea to typed groups is that an object group is created where the type of the group is a "parameter" object and the group of parameter objects is then interpolated with a method call to a group of active objects, such that the *n*th active object method call in the group is passed the *n*th parameter object from the parameter group.

Like any other object, a group of parameters of type `Param` can be passed instead of a single parameter of type `Param` specified for a given method call. The default behavior regarding parameter passing for a method call on a group is to pass a deep copy of the group of type `Param` to all members.[4] Thus, in order to scatter this group of elements of type Param instead the programmer must apply the static method `setScatterGroup` of the `ProActiveGroup` class to the group. In order to switch back to the default behavior, the static method `unsetScatterGroup` is available.

[4]If the members of the group of type `Param` are in fact active objects or groups, then only copies of the references are made. The group collecting such members does not contain a copy of those active objects or groups but only references to them.

The control of diffusion (broadcast) and distribution (scatter) is very fine. It can be specified parameter by parameter. Non-group object are always broadcasted. As presented in the code below, distribution and diffusion of data can be performed in the same group communication.

```
    // Broadcast the object a and the groups bg and cg to
all the
    // members of the group agroup:
agroup.foo(a, bg, cg);
    // Change the distribution mode of the parameter group
cg:
ProActiveGroup.setScatterGroup(cg);
    // Broadcast the object a and the group bg but
    // scatter the members of cg onto the members of ag:
agroup.foo(a, bg, cg);
```

If the parameter group is bigger than the target group the excess members of the parameter group will be ignored. Conversely, should the target group be larger than the size of the parameter group, then the members of the parameter group will be reused in a round-robin (cyclic) fashion. Note that this parameter dispatching mechanism is in many ways a very flexible one. It provides:

- *automatic sending of a group to all members of a group* (default),
- *the possibility to scatter parameters to groups in a cyclic manner*(`setScatterGroup`),
- *the possibility to mix non-group, group, cyclic-scatter group as arguments in a given call.*

All of this is achieved without any modification to the method signature.

3.3. Exceptions and Failures

The failure model provides a mechanism to handle the failure of a method execution or of a method invocation. In the Java framework failures and errors are expressed with `Exceptions`. We can distinguish two kinds of exceptions: the exceptions raised during the method execution and the exceptions raised by the system or the middleware. The first class of exceptions are to be expected while the second should not, as they represent errors in the system or middleware which (in general) is assumed to be stable and error free. In case of a failure of any call the standard approach taken by the JVM is to stop the call and propagate at most one exception. However, since groups are transparent for the functional aspect (method invocation) and rely on communicating with all the members,

the failure of one call should not prevent other calls from terminating. Furthermore, multiple failures could arise during a single group communication operation. Consequently, we define a more suitable mechanism of managing exceptions of any kind that may occur during group communications:

- exceptions are not directly propagated, but are transparently caught and stored for later handling. A structure is introduced in order to store raised exceptions, and inspect them at any time (before the call completes, or after). If a member of a group communication raises an exception, this exception is stored in the result group at the exact place where the result is expected. `Exceptions` are stored in an object named `ExceptionInGroup` that contains a reference to the object on which we tried to invoke the method which triggered the exception. This allows identification of the object that (possibly) experienced a failure. `ExceptionInGroups` are collected into an `ExceptionGroupList` that extends `RuntimeException`, which can be iterated over. For instance:

```
// xg may contain exceptions
AnotherClass xg = agroup.foo();
// Gets the Group interface
Group group_xg = ProActiveGroup.getGroup(xg);
ExceptionGroupList el = group_xg.getException-
GroupList();
... // Iterates on the exceptions
```

- The group communication system does not remove a failed member from a group. It is the programmer's responsibility to do that. A method invocation on a group, therefore, ignores the members that are `Exceptions`. The call is only propagated to valid members. In order to maintain the ordering property, a null reference is placed in the result group at the same index as the exception member.

```
// The call to bar() is not relayed on the exceptions
contained in xg
ThirdClass yg = xg.bar(); // yg may contain null mem-
bers
```

The group communication system assumes that it is not able to resolve the exceptions itself (i.e. determine if the member is lost, momentarily unavailable, or if the exception is an expected behavior). It only avoids the propagation of errors; i.e. invalid method invocations on a

failed group member. Besides, group method calls never critically fail: if the active object is disconnected from the network its call will result in an `ExceptionGroupList` object containing all communication errors; if the caller object disappears (crash or disconnection) once the call has been performed, the method call will be executed on the remote objects even so the caller is absent to collect results. It is the responsibility of the application to program its behavior in case of faults. For instance, assume that the application must stop to wait for the result of a group method call, in case any one call fails. It is possible to program this based upon the iterative use of one of the synchronization primitives mentioned before, as `waitOne`., so as to be aware of the occurrence of any exception as soon as possible.

3.4. Benchmarks

Figure 2 plots the average time (in milliseconds) to perform one asynchronous method invocation on a group of objects, with different sizes for the group. The group members are distributed on 16 machines (a cluster of 933 MHz Intel Pentium III processors interconnected with a 100 Mb/s Ethernet network). The curves represent the performance depending on the number of threads used to make the calls. The more threads we use, the smaller the delay to the group communication operation. The four upper curves are associated with a fixed number of threads. The lowest is

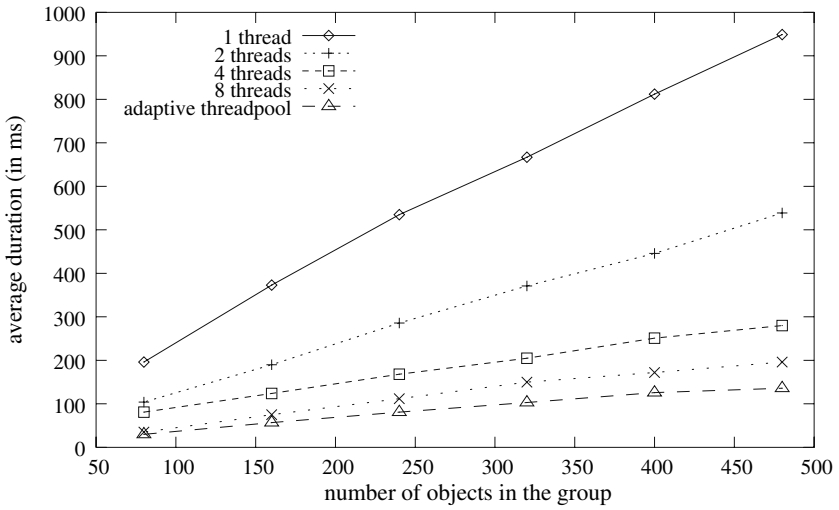

Fig. 2. Adaptative thread pool.

associated with a dynamic number of threads. It shows better performance because the number of threads is at any moment automatically and transparently adjusted to the number required.

By default we associate to a group an adaptive pool of threads in which the member/thread ratio may be adapted by the programmer depending of the requirement of its application. The best ratio for efficient group communications may depend on the group size, the size of exchanged data, the frequency of communication, etc.

An additional number of threads is also maintained. There are two interests of this. First, one may want to maintain a fixed number of threads for a particular purpose: for instance maintaining only one thread in order to emulate a mono-threaded sequential service. Second, additional threads allow to early benefit of multithreading in small sized groups. So the size of the thread pool varies as follows:

$$\text{if ratio} \neq 0: \quad \text{nbThreads} = (\text{nbGroupMembers} \,/\, \text{ratio}) + \text{additionalThreads}$$

0 is a special value for the ratio. It means that the thread pool size is no longer depending on the group size: it is no more dynamic, so:

$$\text{if ratio} = 0: \quad \text{nbThreads} = \text{additionalThreads}$$

We also leave the possibility to the programmer to define his own thread pool's size adaptation by redefining the adaptation method.

Figure 3 presents the average time (in milliseconds) to perform one asynchronous method invocation depending on the amount of data to send (with objects used as parameters). The group contains 80 objects distributed on the same 16 machines. The upper curve shows the performance without any operation optimization. The curve in the middle plots the performance obtained by optimizing the reification operations. The last curve represents the performance obtained by optimizing the reification operations and the serialization. The gap between the two upper curves represents the time spent performing multiple reifications of the same method invocation. As the number of group members remains the same during the whole experiment, the benefit of the optimized reification is constant (one operation instead of 80), thus explaining the fixed gap size between these two curves. In contrast, the optimized serialization becomes more effective depending on the parameters size. Joint optimization provides even better performance, with up to a 3.9 speed up factor in the example presented in Fig. 3.

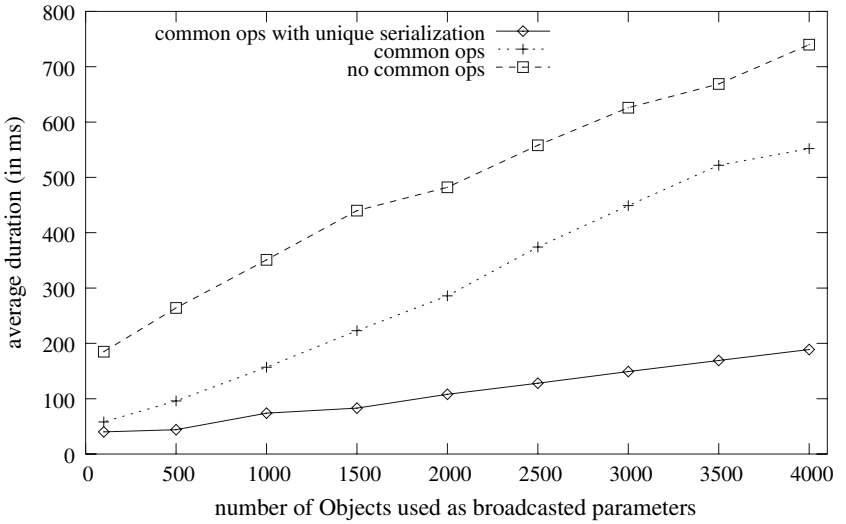

Fig. 3. Factorization of common operations.

To build large applications, *ProActive* provides hierarchical groups: a group of objects that is built as a group of groups. This mechanism helps in the data structuring of the application and makes it much more scalable. A hierarchical group is easily built by adding group references to a group. This operation is simple because groups appear as single typed objects, and thus can be added into another typed group. Type compatibility is the only condition for one group to be added into another group. Here is an example showing the creation of a hierarchical group:

```
    // Two groups
MyClass ag1 = (MyClass) ProActiveGroup.newGroup("MyClass",
...);
MyClass ag2 = (MyClass) ProActiveGroup.newGroup("MyClass",
...);
    // Get the group representation
Group gA = ProActiveGroup.getGroup(ag1);
    // Then, add the group ag2 into ag1
gA.add(ag2);
    // ag2 is now a member of ag1
```

Note that one can merge two groups, rather than add them in a hierarchical way. This is provided through the `addMerge()` method of the Group interface.

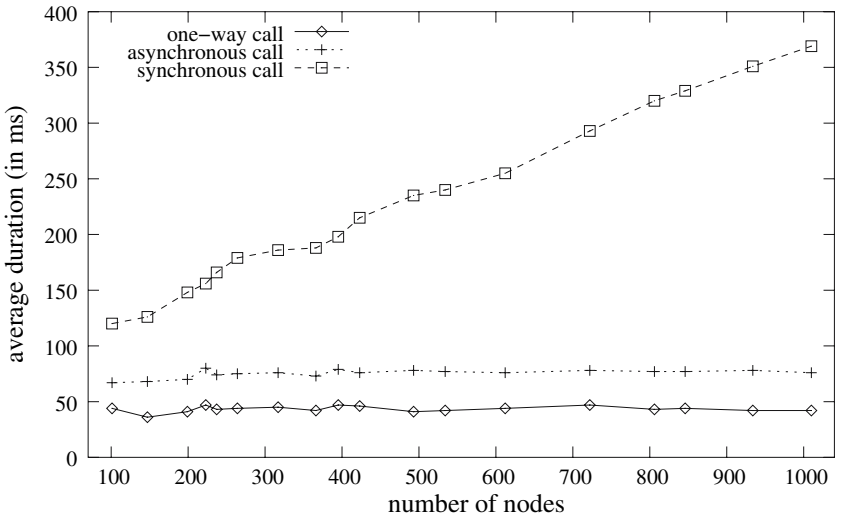

Fig. 4. Hierarchical group on Grid'5000.

By default, groups are local, but it is possible to make them remotely accessible. A remotely accessible group is similar to a service: a message is first addressed to the service, and then forwarded to the group members. *ProActive* provides an easy way to transform any object into a remotely accessible object thanks to the `turnActive()` method. As such, a typed group may be turned into an active object, called an "active group". In this way a group earns all the features of an active object. It becomes remotely accessible, is served by a FIFO message queuing policy, and can be migrated.

The Grid'5000 project aims at building an experimental grid platform gathering nine geographically distributed sites in France, heterogeneously combining up to 5000 processors. Figure 4 presents the performance of communication in a hierarchical group deployed on the Grid'5000 platform. During these tests, each of the nine sites hosted one subgroup. The nine subgroups were then added into a hierarchical group created on one site.

The "one-way call" curve plots the time for a method call without result to reach the subgroups. This time is constant whatever the number of nodes, just because the caller has only to be blocked until the call has reached all subgroups whatever the total number of sites in the Grid (but the number of subgroups remain the same in the entire experiment). The "asynchronous call" curve plots the time for all the subgroup members to

receive the method call, and includes the operation of building the result group and all the futures since the method used returned a result. It can be seen that the time remains constant despite the increasing number of member nodes. The reason for this is that there are 10 – 60 nodes per subgroup on each site. At this scale the time for a group communication on a local LAN is negligible in contrast to site-to-site communication latencies. Finally the "synchronous call" curve plots the time for the communication to fully complete; it means the time for all the members to return a result and for all the subgroups to notify the hierarchical group. The results are more sensitive to the number of nodes because they are directly linked to the performance of the slowest computer in the benchmark.

Overall, this benchmark demonstrates that a typed group communication can be made scalable from the caller point-of-view perceived performances.

4. SPMD PROGRAMMING

The SPMD[5] parallel programming paradigm is a common way to organize a parallel program to be run on clusters of workstations, parallel machines, and, more recently, on grids. A single program is written and loaded onto each node of a parallel computer. Each copy of the program runs independently, other than for the coordination events. Each copy of the program (process) owns a rank number which acts as a unique ID. The specific path through the code is in part selected by this unique ID. Traditional SPMD languages typically do not provide implicit data transmission semantics, but rather communication patterns like *explicit message-passing* implemented as library primitives. This simplifies the task of the compiler, and encourages programmers to use algorithms that exploit locality. Data on remote processors are accessed exclusively through explicit library calls. The most famous environments implementing an SPMD model are PVM (Parallel Virtual Machine) and MPI (Message Passing Interface), both relying on explicit message-passing.

4.1. Adding Object Orientation to the SPMD model: Context and Related Works

4.1.1. Explicit Message-passing Based Object-oriented Approaches

In the 1990's, due to the increasing success of object-oriented programming, many research groups have experimented with the idea of com-

[5]SPMD stands for Single Program Multiple Data.

bining the usage of an object-oriented programming language (such as C++ or Java) and MPI or PVM for writing and running parallel and distributed applications. The MPI-2 specification collects the notions of the MPI standard as suitable class hierarchies in C++, defining most of the MPI library functions as class member functions. This specification has been further extended in Object-Oriented MPI (OOMPI)[22] in order to be able to deal with the transmission of objects. More precisely, OOMPI is a class library specification that encapsulates the functionality of MPI into a functional class hierarchy to provide a simple, flexible, and intuitive interface. OOMPI provides the capability for sending the data that is contained within objects. Since the data contained within an object is essentially a user-defined structure, OOMPI provides a means to build MPI user-defined data types corresponding to object data and to communicate that data in the same manner as communicating primitive data types. These approaches have been further developed with the success of Java and have now led to two main classes of message-passing SPMD within Java:

- a *wrapping of the native MPI implementation library* within the object-oriented language (e.g. mpiJava[23], or JavaMPI[24] where wrappers are automatically generated)
- an *MPI-like implementation of a message-passing specification, written using the object-oriented language* itself, and available as a library such as MPIJ[25], or MPJ[26] in which notions such as communicators, datatype, etc., are modeled as classes.

By 2000–2002, these approaches were considered as a first phase in a broader venture to define a more Java-centric high performance message-passing environment. The main aim was to succeed in reconciling both performance and portability, while not departing from the core goal of offering MPI-like services to Java programs.

4.1.2. Remote Method Invocation Based Approaches

To reproduce SPMD parallelism model in a pure object-oriented way we rely on the following association: by relating a *ProActive*'s group of active objects to an SPMD model's group of parallel 'machines', we associate an active object 'thread of execution' with each 'machine' in the parallel computation. As noted in Ref. 12, this SPMD style can even be combined with a client/server model so as to yield a mixed style. We think this is very appropriate to one application of grid computing: the coupling of a parallel object-oriented SPMD computation to an external and remote application that is in charge of (for instance) steering

and visualization. All mechanisms based on remote method invocation for communication among activities take for granted that this enables the exchange of any typed data, by automatic marshaling-unmarshaling. This is clearly better suited to an object-oriented paradigm in comparison to explicit message-passing in which send and receive operations must be explicitly programmed in matched pairs.

One such approach based on Java remote method invocation, but generalized to support communication between more than two parties, is the object-based Collective Communication in Java.[20] CCJ specifically aims at adding collective operations to Java's object model, implemented on top of RMI. Parallel activities are expressed as thread groups and not as object groups (in fact, activities in Java are expressed by threads which are orthogonal to objects; so, in CCJ the two concepts remain separated, whereas our approach unifies the concept of a thread and an object through the concept of active object[6]). As threads may belong to several groups, this implies that any method of the CCJ API (e.g. `barrier, broadcast, reduce`) aiming at executing an MPI-like collective operation must have the reference of the thread group as a parameter. This is similar to passing the communicator as a parameter in any MPI communication. In CCJ all threads in a group belong to the same program and, in particular, any collective operation must be called on all threads in the implied group. Despite its effort, CCJ fails to fully embrace a full object-oriented model. CCJ actually has to be considered more as a translation of MPI into the Java framework than an innovative solution to create a new programming style inheriting from object-oriented programming and message-passing programming. The set of methods provided by the CCJ API are static methods which eliminate the possibility of directly using the interface of the object. Furthermore, CCJ requires the objects in a thread group to implement specific interfaces and extend a specific class of the API. This requirement is a significant limitation as it impacts on the application conception and eliminates code reuse from existing applications.

Another approach is found in GMI[27], where the underlying computing model is Java RMI, which is extended towards groups. In GMI, RMI concurrency-related problems must be explicitly taken into account by the programmer. GMI is able to perform group method invocation using the interface of the objects however we may regret the same constraint than in CCJ remains: there is an obligation to inherit from specific classes of the API. Furthermore GMI uses an object-oriented approach only for the method invocation on a group; the collecting of results is an

[6]The active object model we promote represents a consensus with respect to such orthogonality: each object can be assigned a thread for serving method calls.

explicit process that dismisses transparency. Finally GMI lacks of dynamicity. First because groups are static, and second because communication schemes (parameter broadcast or scatter) are defined by the programmer while writing the program and can not be modified at runtime.

4.2. Object-Oriented SPMD

The proposed active objects group mechanism presented in Section 3 is already a usable and efficient basis for programming communicating parallel applications using a pure object-oriented paradigm (an example can be found in Refs. 28,29 for a computational electromagnetism application). According to the features this typed group mechanism already provides only a few SPMD-specific aspects are missing. Their addition to the basic model is presented in this section. We name the resulting approach *Object-Oriented SPMD* (OOSPMD for short) because it produces an SPMD programming style totally based on object-oriented mechanisms: the expression of parallelism, message distribution, data exchange, and concurrency are available implicitly through the active object model.

4.2.1. Design and Principles

Our approach for parallelism and collective communications is to group Java active objects into *groups*, which differs from the approach followed in CCJ but is close to GMI. As seen in Ref. 30, we propose a pure object-oriented SPMD programming model as an extension of the basic typed group communication mechanism presented in Section 3. It is extended in two specific ways: (i) the object groups supporting the distributed computation are further organized into a specific topology by way of adding an ID for each member in the SPMD group and "neighbor references"; and, (ii) collective operations have barrier synchronization providing a complete SPMD model in line with an object-oriented approach.

Concurrency management of multiple remote method invocations is automatic and transparent when the group consists of active objects: only one request is served at a time, and the default service policy of method invocation requests is FIFO (although this can be customized if needed). This allows programmers to concentrate only on their functional code. Furthermore, based on the active object paradigm, the 'main' method cannot be used to express and run the core of the parallel task. Instead, the 'main' thread is devoted to support only the sequential servicing of requests. This requires a different approach to expressing the core of the SPMD task, specifically how the control flow dedicated to the parallel algorithm is implemented.

The SPMD features lacking in our basic typed group communication mechanism fall into three categories:

- *Identification of each member taking part in the parallel computation*, and concept of member position relative to the others; for instance a neighboring relation among members. It can be expressed with a basic ranking order or with more complex organizations.
- *Expression of the entire program run by each member taking part in the parallel computation*. Among Multiple Programs Multiple Data (MPMD) paradigms based on object groups, there are some, like GridRPC on Network Enabled Servers such as NetSolve[31] or Ninf,[32], where members act in a sense as passive servers only activated by method calls triggered by clients. On the contrary, in SPMD computing, all members must be active by their own even if, for simplicity, they execute the same program. In *ProActive*, each active object is a proper activity that enacts the sequential servicing of requests. In our approach, the SPMD program will not be expressed in a traditional "big loop" but instead as the implicit result of a succession of service request executed in FIFO order. As will be emphasized below, this way of expressing the core of an SPMD program enables reactive, adaptable, and dynamic behaviour not usually found in the traditional SPMD model.
- *Full range of collective operations (communication and global synchronization) among the members*. Considering the presentation of the typed group communications in Section 3, only the expression of global synchronization barriers is lacking and so will be addressed in this section.

4.2.2. Summary of the OOSPMD API

An OOSPMD group is defined as follows: it is a group of exclusively active objects where each member has a group proxy reference to the group itself (see Fig. 5). Each active object in the SPMD group is also provided with a specific rank in the group. SPMD groups are not immutable. It is the programmer's responsibility to ensure that possible modifications of an SPMD group (add new member, remove member, etc.) maintain the coherence of the group.

```
    // A group of type "MyClass" and its members are cre-
ated at once by
    // an external active object
Object[][] params = {{...},{...}};
Node[] nodes = {...,...,...};
```

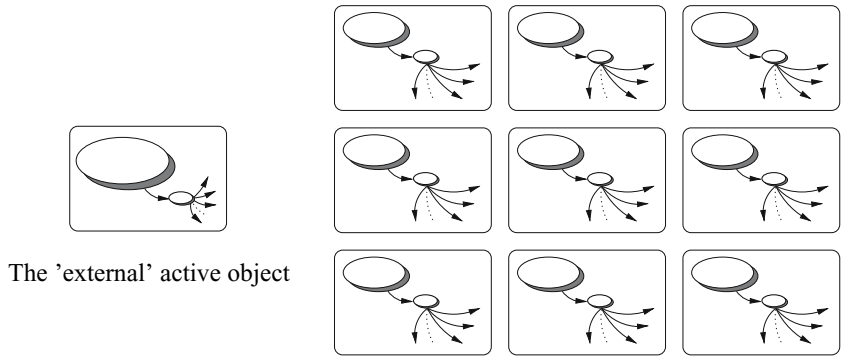

Fig. 5. An SMPD group.

```
MyClass ag = (MyClass) ProSPMD.newSPMDGroup("MyClass",
params, nodes);
    // The computation on each member may now be started,
i.e.
    // invoking a method called e.g compute() defined in
class MyClass ag.compute();
```

When each group member is created, one of the first actions is commonly to get the reference of the group it belongs to, its rank, and so on. One must be careful to clearly distinguish a classical Java reference to the object (`this`), and a *ProActive* asynchronous reference to it, as an active object. The latter enables the active object to implement the parallel task. In OOSPMD, the parallel task on any member of the SPMD group is run by repetitively invoking asynchronous methods to itself, hence the need to have a special asynchronous reference.

In our API there is no specific need to manage data exchange and synchronization explicitly with send- or receive-like operations. This approach is much simpler, as a member triggers data reception and handling through the asynchronous servicing of methods. In other words we do not need additional primitives to explicitly manage data exchange and synchronization. The ordering of the receptions is based on the FIFO ordered strategy (by default) for serving methods. Consequently, any method call triggered by other members or even by an active object not belonging to the group can be served between services of the method calls sent by a member to itself.

Concretely, the parallel task is implemented as iterative asynchronous calls of a method (e.g. named `loop` as in the code below) by the member to itself, so as to maintain an activity. This implies that the reception of data from other active objects in the system (whether or not they belong to the SPMD group) is possible only between atomic method services (e.g. servicing of `loop()`): indeed, receiving such data is effected by serving the corresponding request that is next in the request queue. This implies that any delay in the servicing of `loop()` is prohibited if the member wants to receive data from other members. Triggering the next loop pertaining to the activity must be done through a method call using the asynchronous reference of the member.

```
    // A reference to the typed group I belong to
MyClass a = (MyClass) ProSPMD.getSPMDGroup();
    // An asynchronous reference to myself
MyClass me = (MyClass) ProActive.getStubOnThis();
int rank = ProSPMD.getMyRank(); // My rank in the group
    // Start the 'iterative' loop by sending myself
    // an asynchronous method call
me.loop();
    // To iterate, loop() again calls me.loop()
```

Moreover, as in a traditional SPMD program, execution control is exclusively based on `if` statements and process ID or rank numbers. In our approach, switching execution control can also be based on dynamically created groups at any moment at runtime. Such groups can be derived from existing ones (sub-groups, or group combination for instance) or according to any kind of properties (rank, fields of the object, etc.).

4.2.3. Topologies

To simplify the access to other activities in the group with which a given member must communicate according to the parallel algorithm, it is useful if the SPMD group is further organized according to Cartesian topologies. We have defined the following topologies: line, plan, ring, cube, hypercube, torus, torusCube (torus in 3 dimensions) and tetrahedron. Furthermore, in contrast to statically designed topologies, the addition of new topologies is open. Figure 6 presents possible logical organization given to a group through such topologies.

Topologies are groups: any existing group may be understood as a topology. Creating a topology from a group allows access to a specific set of methods to make it possible to manipulate the neighborhood relationship between group members. Figure 7 presents the class hierarchy

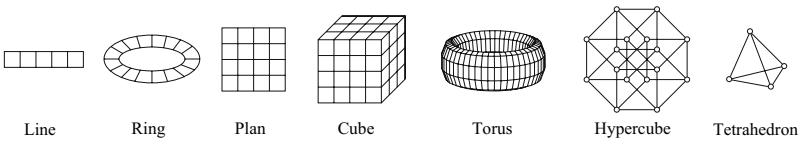

Fig. 6. Topologies.

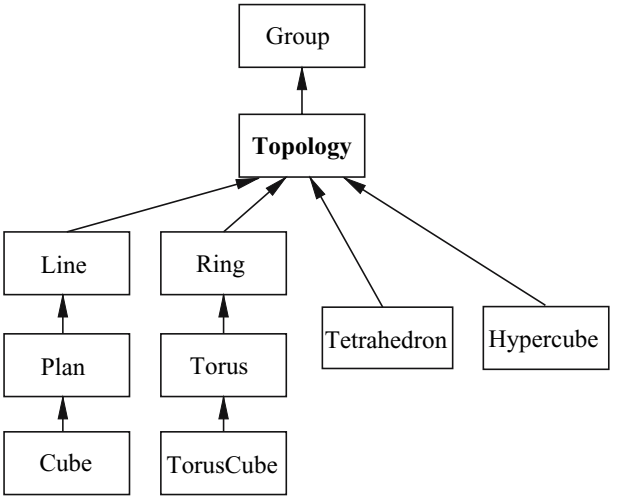

Fig. 7. Topologies classes.

of already existing topologies. The `Topology` abstract class inherits from the interface Group.[7] `Topology` can be extended to create new topologies or to redefine the access method to the neighbors. Three dimensional structures (`Cube` and `TorusCube`) extend two dimensional ones (e.g. `Plan` and `Torus`) that themselves extends One dimensional structures (e.g. `Line` and `Ring`): width, height and depth are successively added to go from a 1D to 3D logical representation of activities, organization and interaction.

A topology is built by copying a group: it is quite similar to a copy-constructor. References to the group members (and not the members themselves) contained in the group are copied to the newly created topology. The group and the topology become two distinct objects, so the modifications performed on one object is not reflected on the other. Here is a topology creation example using the previously obtained SPMD group a

[7]More precisely, the `Topology` class extends the `ProxyForGroup` class that implements the Group interface.

(refer to code above):

```
    // Organize my group as a 2D plan
Plan topology = new Plan(a, aWidth, anHeight);
```

The topologies provide methods to easily access the neighbors of a specified activity, for example to access to the activities which are *a priori* known to be most intensively interacting with the given activity. The set of methods depends on the topology. For instance, a `Line` topology provides `left` and `right` methods while a `Cube` topology provides `left`, `right`, `up`, `down`, `ahead`, `behind`, `line`, and `plan`. Those methods ease the conception and the organization of distributed applications by avoiding long, difficult and error-prone methods which manipulate the group index in order to emulate a complex structure. Additionally, all topologies provide a `neighbors()` method which returns a group composed of the close objects of a given member. For instance, the `neighbors()` method of `Line` returns a group composed of the left and right neighbors; the neighbors method of `Plan` returns a group composed of the left, right, up and down neighbors. The notion of "neighborhood" is strongly attached to the topology. By extending a topology, the programmer may reformulate the neighborhood definition to best fit the needs of the application. Here is a basic example illustrating access and communication with neighbors:

```
    // Get a reference to my neighbors in the plan
MyClass left = (MyClass) topology.left(me);
MyClass down = (MyClass) topology.down(me);
    // One-way communication with neighbors in an asyn-
chronous fashion
left.foo(params);
down.foo(params);
```

There are two ways to obtain `Topology` objects. The first one is to explicitly invoke a constructor with the `new` operation as previously illustrated. The second one is to extract a new topology from an existing one. It is obvious that a plan can be considered as a set of horizontal or vertical lines, for example. Many topologies provide methods returning topologies. By extracting sets that possibly share a common behavior, such methods ease the construction of distributed applications. Here is an example:

```
    // Get a reference to the topology formed by the first
line of the plan
Line line = topology.line(0);
```

```
    // Get a reference to the topology formed by the first
column
Line line = topology.column(0);
```

As topologies are groups, and any group is also a typed group, all topologies can be viewed as a typed group: in this case we can also call it a *typed topology*. `The get By Type` method converts the topologies into typed groups as presented in Section 3. Like a typed group, a typed topology exposes a common interface of its members. Method invocations achieve communication with all group members. The group communication semantics which have already been presented are applied to perform such method calls. The following example presents communications addressed to topologies:

```
    // Convert the topologies into typed topologies
MyClass gplan = topology.getByType();
MyClass gline = line.getByType();
    // communicate with the topologies' members
gplan.foo();
gline.foo();
```

4.2.4. Synchronization Barriers

The only collective behavior related methods of our OOSPMD API pertain to global barriers. The standard definition of a global barrier is that all members in the group (or those enrolled in the barrier, see below) must not proceed further in their computation until all the members have reached the barrier. Given the active object model, we propose a slightly different but more appropriate semantic: from the viewpoint of a member reaching a barrier, it is effective (i.e. it blocks the member) not immediately, but in the future: more precisely at the exact moment when the current service has terminated. In practical terms, all instructions lying after the barrier in the current method being served will be executed. Nevertheless, the usual meaning of a global synchronization barrier is respected but it pertains to the servicing of the next request instead of pertaining to the next instruction. When encountering a barrier, the service of the first incoming request from an SPMD group member waiting in the request queue will be able to proceed on any enrolled member only when all have reached the barrier. As mentioned earlier, requests originated from "external" objects may be served before this.

A suspended activity remains able to receive requests and put them in queue. In that way, all objects are allowed to communicate with the suspended object. Even if those methods will not be served instantaneously,

objects within the group can keep invoking group methods, even if some objects are still waiting at the barrier. A barrier is limited to its participants. It means that requests sent by an object not belonging to the SPMD group involved in the barrier can be served, even if the activity is blocked by a barrier. Of course, it is the programmer's responsibility to take care of this flexibility of being capable of servicing an external request, while being currently blocked at a synchronization barrier. It is important that objects manage inbound method servicing while they are held at a barrier such that their state is not modified in a way that would affect their behavior when they are released. In other words, an external access to an object while it is held at a barrier must not affect the behavior of that object or the object group past the barrier. Such flexibility in barriers, even if it is an avenue for possible errors if misused can be helpful to couple different applications together. For instance, consider a visualization tool to be connected to the SPMD code that computes the values: the visualization tool could monitor or steer the parallel code in an asynchronous manner, to get the current status or the most recent computed values, even if this parallel computation is currently blocked due to the barrier.

We propose three kinds of barriers, two global and one local:

- First, a *total barrier*, within which a string parameter represents a unique identity name for the barrier. It is assumed that this blocks all the members in the SPMD group.
  ```
  ProSPMD.barrier("MyBarrier");
  ```
- A *neighbor barrier*, involving not all the members of an SPMD group, but only the active objects specified in a given group. Those objects, which contribute to the end of the barrier state, are called neighbors as they are usually local to a given topology. An active object that invokes the neighbor barrier must be in the group given as a parameter.
  ```
  ProSPMD.barrier("bar", neighborsGroup);
  ```
  It is interesting to notice that the following instruction:
  ```
  ProSPMD.barrier("bar", ProSPMD.getSPMDGroup());
  ```
  is identical to a total barrier call. If the neighborhood involved in a neighbor barrier is the whole SPMD group, then the neighbor barrier becomes a total barrier.
- A *method barrier* stops the active object that calls it, waiting for a request on all the specified methods to be served. The order of the methods does not matter, nor the active objects they come from. As such, this barrier is purely local, and does not trigger extra messages to be exchanged as do the two others. For example:

```
ProSPMD.barrier({"foo","bar","gee"});
```

One may want a sequential treatment of the methods, meaning a block on the current activity until first `foo()` has arrived, then `bar()`, then `gee()`. This is achieved by invoking the method barrier in the desired order, as follow:

```
ProSPMD.barrier({"foo"});
ProSPMD.barrier({"bar"});
ProSPMD.barrier({"gee"});
```

A method barrier does not need the involvement of the SPMD group or of a neighbor group. In contrast to the previous barriers (total and neighbor), the method barrier blocks the servicing of all requests regardless of their origin (i.e. from a member of the SPMD group, neighbor group, or otherwise).

Of course, none of these barriers are implemented with an active wait. Resources are not consumed while waiting. The activity waits for the condition to be satisfied to resume.

4.2.5. Example and Benchmarks

We illustrate OO-SPMD with a concrete example. We choose *Jacobi iterations* because it is a simple application and easy to distribute in a traditional SPMD manner. The algorithm performs local computation and communication to exchange data. The Jacobi method is a method of solving a linear matrix equation. Each element is solved by computing the mean value of the adjacent values. The process is then iterated until it converges to a given threshold. The following code shows the main loop (an iteration based loop) of a solver. During an iteration, the value at a point is replaced by the average of the up, down, left, and right neighbor values. External boundary values are fixed statically at the beginning of the application and do not change at runtime.

```
while (!converged) {
   for (y=1 ; y < MATRIX_HEIGHT-1 ; y++) {
     for (x=1 ; x < MATRIX_WIDTH-1 ; x++) {
        new(x,y) = ( old(x,y-1) + old(x,y+1)
         + old(x-1,y) + old(x+1,y) )/4;
        if (abs(new(x,y) - old(x,y)) < THRESHOLD) {
            converged = true;
        }
        exchange(new,old);
} } }
```

The structure of this code is quite simple, so we use a coarse-grained data-parallel approach to transform it into a similar parallel code. The arrays `old` and `new` are distributed over nodes taking the form of active objects. Each active object, named `SubMatrix`, is responsible for receiving the boundary values from adjacent sub-matrixes and computing its own part of the data. As shown in Fig. 8, communications occur at block boundaries. Each sub-matrix communicates with two, three, or four neighbors, depending on their position (either at a corner, a border, or in the center of the whole matrix). Communications appear at sub-matrix boundaries to send boundary values to neighbors and receive values of neighbors.

4.2.5.1. MPI Jacobi. Using a message passing approach based on asynchronous sends and receives with the MPI library the resulting parallel code is necessary:

```
while (!converged) {
   internal_compute(&converged);
   MPI_Send(north_border, SUBMATRIX_WIDTH, MPI_DOUBLE,
   north, 1,
                              MPI_COMM_WORLD, &status);
   MPI_Recv(border_received_from_north, SUBMATRIX_
   WIDTH, MPI_DOUBLE,
               north, 1, MPI_COMM_WORLD, &status);
   // Repeat the same operations (send and receive)
   for south, east, west
   ...
   boundaries_compute(&converged);
   exchange(new,old);
} } }
```

The send and receive operations are repeated for each communication with a neighbor (up to 4), even if the operations are very similar.

4.2.5.2. OO-SPMD Jacobi. Using our OO-SPMD approach, the code becomes much more concise. The whole matrix is distributed and understood as a two-dimensional topology using the `Plan` topology. The neighborhood of any `SubMatrix`, named `neighbors` in the example, is automatically obtained through methods of the `Plan` topology.

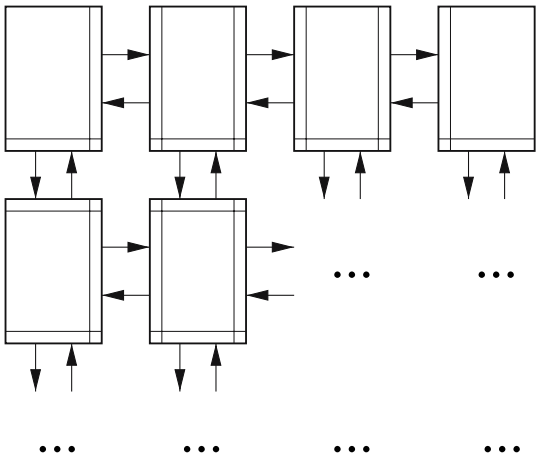

Fig. 8. Jacobi distributed algorithm.

```
me = ProActive.getStubOnThis();
public void jacobiIteration() {
    internal_compute(); //updates converged
    neighbors.send(boundariesGroup);
    ProSPMD.barrier({"send", ... ,"send"});
    me.boundaries_compute(); //updates converged
    me.exchange();
    if (!converged) me.jacobiIteration();
}
```

Synchronization is done by data flow, and the barrier ensures that the sub-matrix and its neighbors have exchanged their own boundaries values before computing the whole boundaries. The method calls performed after the barrier must be asynchronous (put in the queue of the active object), hence the use of the `me` active object reference "to self", otherwise they would be served immediately, i.e. before the execution of the barrier. Overall, according to the semantics of the `barrier()` method, the data (i.e. parameter of send) will have been exchanged before the barrier will be released, guarantying that any group member gets the data in order to compute the boundaries values (`boundaries_compute()`). Data communications to all neighbors is performed using a scatter group (the group of boundaries `boundariesGroup`): as the real parameter of the send method is a scatter group, it is transparently scattered to each member of the neighbors group. As for the MPI version, the construction

of the structures containing boundaries values was not shown in the example code. It only consists of building a group containing the boundaries.

An interesting property of our model is that it remains reactive. This means that any part or any member of an OOSPMD application may also serve incoming method call requests incoming from another application. This is allowed by the fact that the parallel task is expressed as asynchronous calls to a method (`jacobiIteration()` for instance): an external request is thus able to come in between requests addressed to the active object. We think this flexibility is very appropriate to one of the many possible applications of grid computing: the coupling of a parallel object-oriented SPMD computation and an external and remote application that is in charge of, for instance, steering or visualization.

The benchmark presented in Fig. 9 used a cluster of 16 933 MHz dual-Pentium III 512MB (SDRAM) — 256 Kb L2 cache CPUs with Linux RedHat kernel 2.4.20, interconnected with a 100 Mb/s Ethernet. For the C/MPI version we used gcc 3.3.2 and MPICH 1.2.5.2. For the Java version, we used the Sun Java Virtual Machine 1.5.0.

The graphic presents the average duration in milliseconds of one Jacobi iteration depending of the amount of data contained on each node, in millions of double. In Java one double is encoded on 8 bytes. Of course, the C language with MPI remains more efficient than Java with RMI. But the performance ratio of 3.3 (average) is maintained despite the growth of data (see the curve labeled "ratio"). It is interesting to notice that 3.3

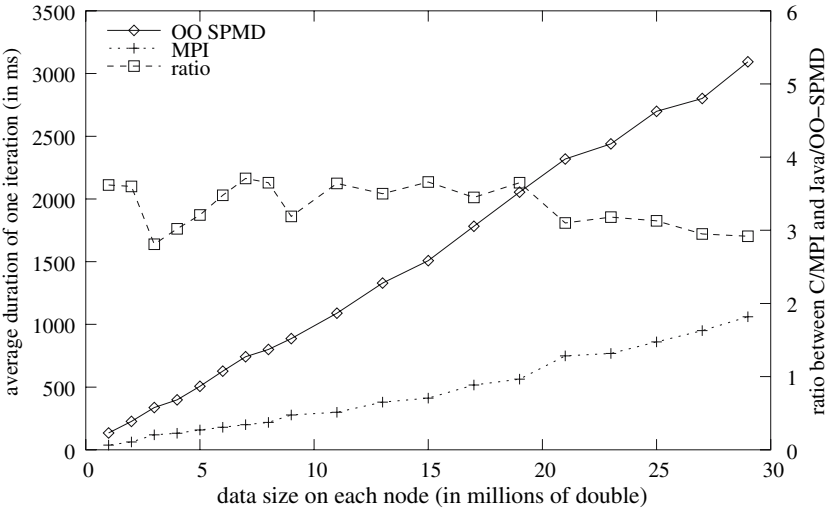

Fig. 9. Performances of C/MPI and Java/OOSPMD versions.

is also the ratio of performance between Java and C for the sequential versions. Our approach thus allows an efficient distribution and is scalable regarding data. For 29M of doubles (i.e. 221 MB sub-matrix), speedup of the C/MPI version is 15.41 and duration of one iteration is 1061 ms; speedup of the Java/OO-SPMD is 15.23 and duration of one iteration is 3094 ms.

In conclusion OOSPMD is a new programming style that inherits from both object-oriented and SPMD programming styles. In addition to the typed group mechanism, OOSPMD introduces the notion of a SPMD typed group offering particular features: neighborhoods, topologies, and dedicated barriers. This new style implies that applications that are to be run on clusters or grids would need to be rewritten or at least deeply modified which would require software development effort. Given this constraint, this new programming style is most applicable to new applications. The *ProActive* platform[15], however, provides an alternative way to incorporate a legacy SPMD (MPI-based) code into an active object based application. It relies onto a classical approach by having an active object playing the role of a wrapper of the embedded MPI code, and an API to be used within the MPI code to trigger data send and receive to and from the wrapper (resulting in only minor modifications in the existing MPI code). Doing this, the wrapped legacy MPI code can potentially be incorporated into a larger set of active objects.

5. EXAMPLES

The objective of this section is to illustrate the applicability of the typed group mechanism on some non-trivial and realistic applications. We do not detail them, but give the key idea at which point the use of this mechanism proved to be relevant. The first two examples are applications in the usual sense, that are scientific, computation intensive, so parallel. The third one illustrates its use within the definition of a framework (presently, the definition of a grid software component model). In this case, the typed group mechanism is hidden to programmers, but is still relevant.

Let us notice that the Jacobi OOSPMD version described in Section 4.2.5 was presented to specifically illustrate the new SPMD programming style. Besides, other applications were programmed without requiring the usage of an OOSPMD typed group, but only simple typed group. We concentrate on one of them pertaining to scientific computing, named *Jem3D* in the sequel, and on another one pertaining to financial risk computing, named *PicsouGrid* in the sequel.

5.1. Jem3D: Simulation of Electromagnetism Wave Propagation in 3D

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 Ref. 33 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 numerical kernels and the parallelization aspects) consists in the simulation of the propagation of an eigenmode value in a cubic metallic cavity. For this test case, the underlying tetrahedral mesh is built by defining a Cartesian grid discretization of the cube and then dividing each element of this grid in six tetrahedra, where the local calculation of the electromagnetic fields is performed. More precisely at each time step, the resulting flux (i.e. the contribution of each tetrahedron to the electromagnetic fields) is evaluated: on each tetrahedron, it's the result of the combination of the elementary fluxes computed through all its four facets.

In the sequential version, after each step, the local values calculated in a tetrahedron are passed to its neighbors, and a new local calculation starts. In the parallel and distributed version, the cube is divided into sub-domains which can be placed over different machines in order to have parallelism. Inside a sub-domain, the calculation behaves like in a domain except that tetrahedra located on the boundary of a sub-domain have to communicate their results to ones located in a different address space through their border faces, using remote calls.

Jem3D is written completely in Java on top of *ProActive*.[28,29] Sub-domains are active objects and communication between them is done through the group communication mechanism exclusively. Each communication between sub-domains aims at sending a linked list whose elements, one for each border face shared between a given pair of sub-domains, contain one array of three doubles. Consequently, each sub-domain has a reference to a typed group of sub-domains with which it share tetrahedron faces (the border faces). To send all elements to all sub-domains in a single group communication, the linked lists are collected as a group, of type scatter.

Experiments have been conducted using different mesh sizes and number of nodes, and compared with an equivalent MPI/Fortran version. Regarding the *ProActive* version, we experimented the one where the standard RMI protocol is used as a transport protocol, and an other one, where an optimized implementation of RMI (Ibis) is used. The performances below were obtained on the Distributed ASCI Supercomputer 2

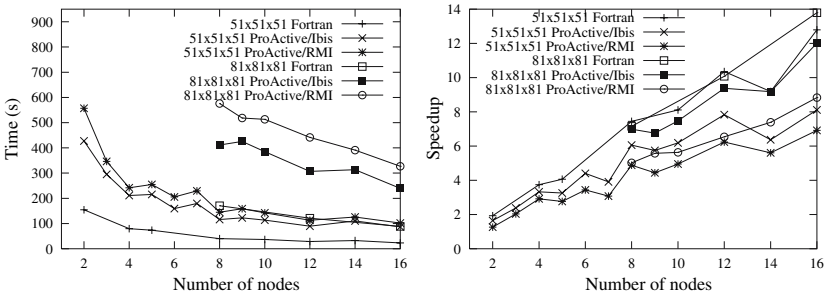

Fig. 10. Execution time and speedup for the Java and FORTRAN.

(DAS2-2).[8] The nodes are composed of Dual Pentium III CPU running a 1 GHz with 1 GB or more memory, running Red Hat Linux and linked with fast-Ethernet. Two remote sites of the DAS-2 are connected through 10 Gbits/s connections. We used the Sun JDK 1.4.2. for all our experiments.

First the ratio between the sequential execution time of the Java and FORTRAN versions is around 2.3 which we believe is good. Considering the distributed version, whose results can be seen in Fig. 10, the Java ProActive version is, to no surprise, still slower than the FORTRAN one. However, whereas the ratio of execution time ProActive RMI vs. FORTRAN was almost all the time around 3.5, we achieved a much better performance with ProActive Ibis, lowering this ratio to around 2.5, very close to the sequential one.

5.2. PicsouGrid: Option Pricing in Finance

PicsouGrid is a software architecture, built using *ProActive*, with the aim to offer a grid aware architecture for financial risk analysis, focusing on options, one of the main instruments of financial risk management.[34] The grid solution we introduced[35] has as main goal to create a grid based set of distributed computing services, started at boot-time and remaining up, ready to quickly process any client request, and fault-tolerant, in order to fulfill requests even in non-reliable environments.

Consequently, a hierarchical structure was adopted: a server (acting as an entry point to the PicsouGrid) controls a set of sub-servers, which in turn control a large number of workers. Sub-servers are collected as a typed group. Each sub-server controls a set of workers, also managed as a typed group. Many risk analysis are based on Monte Carlo simulations,

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

and need to run at least a fixed number of simulations to achieve the required accuracy. To achieve these stochastic computations as fast as possible, and in a reliable manner, the adopted solution consists in dividing the number of tasks given the number of sub-servers; then each sub-server sends tasks to workers until it has collected enough results. This translates into a group communication, first from the server to the sub-servers, second to each sub-server to its workers, with group of futures to collect the results. In case of failure of any member, it is immediately replaced by a new one that replaces it in the group. The replacing member is immediately given a piece of work through a point-to-point *ProActive* standard communication, and the corresponding group of futures is updated accordingly.

We have evaluated PicsouGrid with a simple European option pricing, on Grid'5000. The number of sub-servers needs to be adapted to the size of the grid. On one site, one sub-server is enough to manage up to 40 workers, then 2 sub-servers are desirable to manage around 70 processors, and then 4 sub-servers are desirable to manage more processors. Identifying the best configuration need to be investigated in the future, but at least, the group based architecture allows to set-up such numbers without impact on the functional part of the application.

5.3. GCM: A Software Component Model Offering Collective Interfaces

The software component-oriented approach has gained more and more interest as an adequate support for grid applications programming, deployment, and dynamic reconfiguration. The most popular and already used component model is Common Component Architecture (CCA), for which several implementations already exist: DCA, XCAT, LegionCCA, etc.[36] However, we recently promoted, within the context of the CoreGRID European Network of Excellence, a new one, named GCM in the sequel. GCM builds upon the hierarchical component model named *Fractal*.[37,38]

A Fractal component is formed out of three parts:

- *A content* that can be recursive. As a component can contain other components, the model is hierarchical, and this property, unique to Fractal, dramatically enhances code reuse and structuration.
- *A set of controllers* that provide introspection capabilities for monitoring and exercising control over the execution of components. Dynamic (re-)configuration is the key add-on of components compared to say objects. Fractal controllers provide a proper support to this.

- *A set of interfaces* with which the component interacts with other components, in a RMI like style. These interfaces can be either client or server, and are interconnected using bindings.

As *ProActive* offers many features, such as distribution, asynchronism, and mobility that would be interesting for Fractal components, an implementation of the Fractal API was developed on *ProActive*.[39]

However, parallel computation patterns such as those offered by skeletons (master-slave, divide-and-conquer, etc) or SPMD models (including SPMD codes coupling[40]), exhibit collective behaviors. Such patterns are needed also in the context of component-based grid programming.[41] Consequently, creating such patterns as compositions of components requires the capability to expose the induced collective behaviors at the level of components. Consequently, the GCM introduces the notion of collective interfaces:[42]

- Multicast: a multicast interface transforms a single invocation into a list of invocations. Its customization is done along two directions: selection of the bound components that will receive an invocation, and how invocation parameters are distributed (they are either broadcasted without modification, or scattered). By default, the client using such an interface has to expect a list of results. A third customization direction enables to specify a transformation function in order to change this standard behaviour, possibly also changing the type of the result (Fig. 11).
- Gathercast: a gathercast interface transforms a set of invocations into a single invocation. A gathercast interface coordinates incoming invocations before continuing the communication flow. Consequently, it acts as a synchronization barrier besides gathering

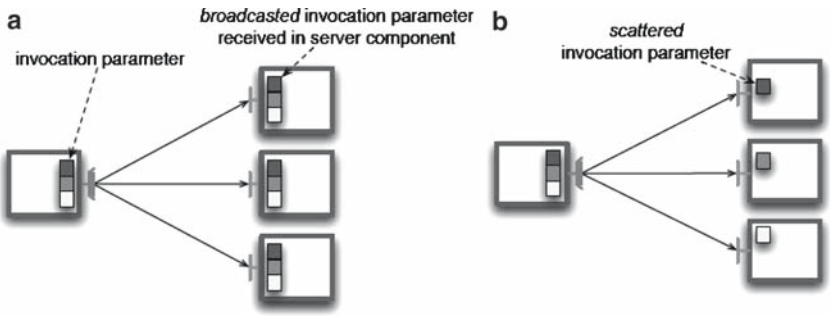

Fig. 11. A Multicast interface, showing two possible parameter distributions. (a) Invocation Parameter and (b) scattered invocation Parameter.

the invocation parameters. Its customization is done also along two directions: selection of the bound components from which it will receive an invocation, and how the result is propagated back to the callers (either broadcasted or scattered). Like for the multicast case, the server bound on such an interface has to expect a list for each of the invocation parameters. So, a third direction of customization enables to specify a transformation in order to change the result, and possibly also the type, of parameters aggregation (e.g. through a reduction function to yield one single value out of a collected set) (Fig. 12).

- MxN: The MxN problem[43] pertains to the coupling of two parallel components, one having an M cardinality, the other the N cardinality, in such a way that their data is efficiently exchanged, possibly subject to redistribution. In the GCM, this problem is naturally expressed by connecting the two hierarchical components: on the client side, a component with M inner components, offering a gathercast interface, and on the server side, a component with N inner components, offering a multicast interface. Redistribution of data is configured through the respective collective interfaces controllers. We are working on an optimization of the binding between the gathercast and multicast interfaces to avoid the possible bottleneck of the collective interface at the level of the two composite components, and consequently improve the efficiency of data redistribution (see Fig. 13).

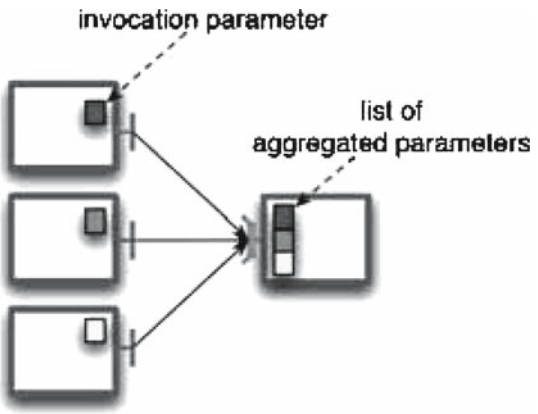

Fig. 12. A Gathercast interface, showing one parameter aggregation.

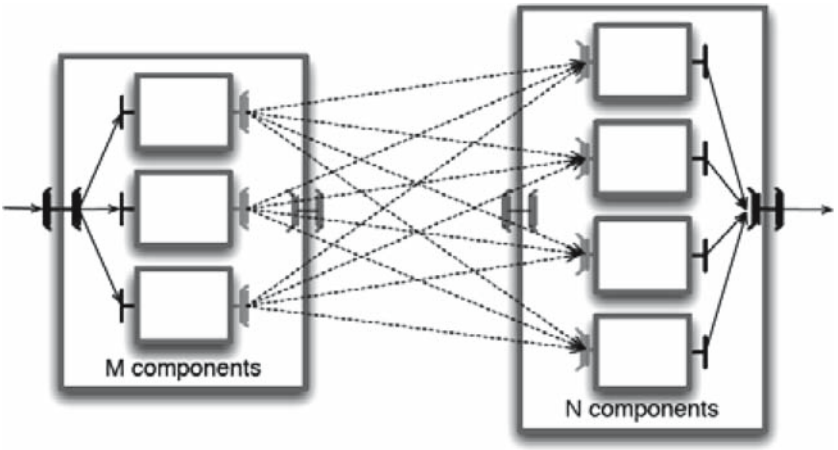

Fig. 13. An optimized MxN interface, replacing the gathercast-multicast pair by M multicast interfaces directly bound to N gathercast interfaces to express data redistribution.

It is obvious that the implementation of collective interfaces, specially the multicast, can reuse the existing mechanism for typed group communications: the capabilities to have parameters be broadcast or scattered, and the transfer of invocations in parallel to a dynamically formed set of bound components.[44]

In this framework, components are being given their 'grid-oriented' nature by relying on *ProActive* active objects, and typed groups. This gives interesting features:

- *ease of use*, thanks to transparency, and keeping configurability of the collective behavior of interfaces, without relying on additional components (as opposed to Ref. 45),
- *performance*, thanks to asynchronism and group communication optimizations and the absence of intermediate components,
- while enabling *deployment of those parallel and distributed components on any grid middleware and platforms*, and if needed, on any heterogeneous combination of them.

6. CONCLUSION

We have presented an original approach to the handling of object groups, and, more specifically, an elegant and efficient extension to the classical typed method invocation mechanism applied to groups. This was done with the general aim to ease parallel and distributed programming

which is becoming increasingly complex and demanding with the emergence of heterogeneous, large-scale distributed systems. This elegant, natural and somehow transparent extension has been made possible mainly due to the powerful mechanism that the *ProActive* library is based upon: a meta-object protocol which reifies method call towards distributed object. Moreover, the benchmarks we presented prove the scalability and efficiency of the mechanism. Consequently, the research presented in this article gives strong arguments in favor of meta-level based and object-oriented programming models, even for high-performance parallel and distributed computing.

The group mechanism can directly be applied within standard object-oriented parallel and distributed programs, such as those designed with the active object pattern[28], some of them having been briefly presented in Section 5. We have also shown that the mechanism can be very useful for providing more elaborated and structured frameworks i.e. to build parallel and distributed applications by software composition.

Nevertheless, we focused on the approach that is the most popular in high-performance computing: the SPMD model. The original object-oriented SPMD programming solution we define is a smooth and perfectly integrated extension of the active object principle. We hope to have demonstrated to the programmer that by using it programs can be grounded on a single concept, the *active object*. Using this paradigm, the whole spectrum of applications can be seamlessly targeted. This ranges from sequential mono-threaded, concurrent and multi-threaded, distributed, up to parallel and distributed applications.

One of our current works pertains on wrapping a legacy MPI parallel program within *ProActive* active objects, and so to achieve the coupling of MPI codes on the grid thanks to ProActive. A natural perspective is to also apply this wrapping of a MPI code to expose it as a set of GCM components: the active object associated to each component would be in charge of translating interactions from the inside or the outside of the MPI code into component interface invocations.

One of our broader research perspectives pertains to the a priori not natural combination of SPMD and component-orientation for programming a single application targeted to be run on a grid: our claim is that programming for the grid ends up in defining a hierarchical but parallel organization of quasisynchronized pieces of SPMD (usually legacy) codes. The structure and depth of the software hierarchy should be able to reflect and adapt to the physical structure and depth (e.g. multi-core nodes, regrouped on multi-processors PCs, organized within clusters, then, interconnected through wide area networks). We expect the hierarchical, typed group of objects software mechanism we defined in this article, and its smooth extension for supporting OO-SPMD groups to be very useful

for this purpose. Indeed, we aim to organize the application into hierarchical levels of composite, parallel components; each component interacting and synchronizing along the OO-SPMD group associated to the level it belongs to (and the same at each level of recursion). Moreover, intra or inter-level effective communications among components can be adapted to the properties of the underlying grid platform (e.g. if they are firewalls to protect access to clusters), or to the nature of the communication link. Indeed, the way the group mechanism presented here has been implemented is sufficiently flexible to allow it to be ported on any sort of communication low-level protocol, including multi-point variants as briefly mentioned in the introduction and detailed in Ref. 16.

REFERENCES

1. The Globus Project, http://www.globus.org.
2. A. Natrajan, A. Nguyen-Tuong, M. A. Humphrey, and A. S. Grimshaw, The Legion Grid Portal, *Concurrency and Computation: Practice and Experience* **14**(13–15):1365–1394 (2002).
3. Unicore, http://unicore.sourceforge.net.
4. J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, Condor-G: A Computation Management Agent for Multi-Institutional Grids, in *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10'01)*. San Francisco, California, USA: IEEE Computer Society, pp. 55–63 (Aug. 2001).
5. M. D. Santo, N. Ranaldo, and E. Zimeo, A Broker Architecture for Object-Oriented Master/Slave Computing in a Hierarchical Grid System, in *Proceedings of Parallel Computing*, Dresden, Germany (Sept. 2003).
6. S. Gorlatch, Send-receive Considered Harmful: Myths and Realities of message passing, *ACM Transaction on Programming Languages and Systems,* **26**(1):47–56 (2004).
7. K. Jeacle, and J. Crowcroft, Reliable High-Speed Grid Data Delivery Using IP Multicast, in *Proceedings of the UK e-Science All-Hands Meeting*, Nottingham, United Kingdom (Sept. 2003).
8. M. Maimour, and C. Pham, An Active Reliable Multicast Framework for the Grids, in *Proceedings of the International Conference on Computational Science*, Amsterdam, The Netherlands, pp. 588–597 (Apr. 2002).
9. *Java Remote Method Invocation Specication*, Sun Microsystems (Oct. 1998) ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.
10. A. D. Birell, and B. J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, **2**(1):39–59 (1984).
11. S. Maffeis, The Object Group Design Pattern, in *Proceedings of the Second USENIX Conference on Object-Oriented Technologies*, Toronto, Canada (June 1996).
12. J. Maassen, Method Invocation Based Communication Models for Parallel Programming in Java, Ph.D. dissertation, Vrije Universiteit, Amsterdam, The Netherlands (June 2003).
13. A. S. Grimshaw, W. T. Strayer, and P. Narayan, Dynamic Object-Oriented Parallel Processing, IEEE Parallel & Distributed Technology: Systems & Applications, **1**(2):33–47 (1993).
14. C. Pérez, T. Priol, and A. Ribes, PaCO++: A Parallel Object Model for High Performance Distributed Systems, In *Distributed Object and Component-based Software Sys-*

tems Minitrack in the the 37th Hawaii International Conference on System Sciences (HICSS-37), Hawaii, USA 2004, IEEE Computer Society Press, (2004).
15. ProActive, http://www-sop.inria.fr/oasis/ProActive.
16. L. Baduel, D. Caromel, N. Ranaldo, and E. Zimeo Effective and Efficient Communication in Grid Computing with an Extension of ProActive Groups, in *Proceedings of the Java for Parallel and Distributed Computing Workshop at IPDPS* (2005).
17. D. Caromel, Towards a Method of Object-Oriented Concurrent Programming, *Communications of the ACM*, **36**(19):90–102 (1993).
18. F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssiere, Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications, in *11th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Edinburgh, Scotland: IEEE Computer Society pp. 93–102 (July 2002).
19. V. Felea and B. Toursel, Methodology for Java Distributed and Parallel Programming Using Distributed Collections, in *Proceedings of the Workshop on Java for Parallel and Distributed Computing at IPDPS*, Fort Lauderdale, Florida, USA (Apr. 2002).
20. A. Nelisse, T. Kielmann, H. E. Bal, and J. Maassen, Object-based Collective Communication in Java, in *Joint ACM Java Grande - ISCOPE Conference*. Palo Alto, California, USA: ACM Press, pp. 11–20, (June 2001) iSBN 1–58113-359-6.
21. L. Baduel, F. Baude and D. Caromel, Efficient, Flexible, and Typed Group Communications in Java, in *Joint ACM Java Grande - ISCOPE Conference*, Seattle, Waghington, USA: ACM Press, pp. 28–36 (Nov. 2002).
22. J. M. Squyres, B. C. McCandless, and A. Lumsdaine, Object Oriented MPI: A Class Library for the Message Passing Interface, in *Proceedings of the POOMA conference*, Santa Fe, New Mexico, USA (Feb. 1996).
23. M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, mpiJava: An Object-Oriented Java interface toMPI, in International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP, San Juan, Puerto Rico (Apr. 1999).
24. S. Mintchev and V. Getov, Towards Portable Message Passing in Java: Binding MPI, in *Recent Advances in PVM and MPI*, Ser. LNCS, no. 1332, Springer Verlag (1997).
25. G. Judd, M. Clement, and Q. Snell, DOGMA: Distributed Object Group Metacomputing Architecture, *Concurrency: Practice and Experience*, **10**(11–13):977–983 (1998).
26. B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, MPJ:MPI-like Message Passing for Java, *Concurrency: Practice and Experience*, **12**(11):1019–1038 (2000).
27. J. Maassen, T. Kielmann, and H. E. Bal, GMI: Flexible and Efficient Group Method Invocation for Parallel Programming, in *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'02)*, Washington D.C., USA (Mar. 2002).
28. L. Baduel, F. Baude, D. Caromel, C. Delbé, S. ElKasmi, 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, NewMexico, USA: IEEE Computer Society (Apr. 2004).
29. F. Huet, D. Caromel, and H. E. Bal, A High Performance JavaMiddleware with a Real Application, in *Proceedings of the Supercomputing conference*, Pittsburgh, Pensylvania, USA (Nov. 2004).
30. L. Baduel, F. Baude, and D. Caromel, Object-Oriented SPMD, in *CCGrid 2005: IEEE International Symposium on Cluster Computing and the Grid*, Cardiff, United Kingdom (May 2005).
31. NetSolve, http://icl.cs.utk.edu/netsolve.
32. Ninf-G, http://ninf.apgrid.org.

33. 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).
34. J. Hull. Options, Futures and Other Derivatives. Prentice Hall (2005).
35. S. Bezzine, V. Galtier, S. Vialle, F. Baude, M. Bossy, V-D. Doan, and L. Henrio. "A Fault-Tolerant and Multi-Paradigm Grid Architecture for Time Constrained Problems. Application to Option Pricing in Finance in 2nd *IEEE Int. Conference on e-Science and Grid Computing* (December 2006).
36. R. Schmidt, M. Head, M. Govindaraju, M. Lewis, and S. Benkner, Design and Implementation Choices for Implementing Distributed CCA Frameworks in *HPC–GECo/Compframe Joint Workshop at the 15th High Performance Distributed Computing conference (HPDC-15'06)*, Paris, France: IEEE Computer Society (June 2006).
37. The Fractal Project, http://fractal.objectweb.org.
38. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java, in *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, Vol. 36, pp. 11–12 (2006).
39. F. Baude, D. Caromel, and M. Morel, From Distributed Objects to Hierarchical Grid Components, in *International Symposium on Distributed Objects and Applications (DOA)*, ser. LNCS, no. 2888. Springer Verlag, pp. 1226–1242 (2003).
40. C. Pérez, T. Priol, and A. Ribes, A Parallel CORBA Component Model for Numerical Code Coupling, *International Journal of High Performance Computing Applications*, **17**(4):417–429 (2003).
41. M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo, Components for High Performance Grid programming in the Grid.it Project, in *Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, eds, Springer, 2005, revised versions of the communications made at the Workshop on Component Models and Systems for Grid Applications at ACM International Conference on Supercomputing (June 2004).
42. Proposals for a Grid Component Model, CoreGRID, Programming Model Institute, Tech. Rep. D.PM.02, Nov 2005, www.ercim.org/bscw/bscw.cgi/d98179/D.PM.02.pdf.
43. F. Bertrand, D. Bernholdt, R. Bramley, K. Damevski, J. Kohl, J. Larson, and A. Sussman, Data Redistribution and Remote Method Invocation in Parallel Component Architecture, in *19th International Parallel and Distributed Processing Symposium (IPDPS'05)* (Apr. 2005).
44. Matthieu Morel Components for Grid computing. PhD Thesis. Univ. of Nice Sophia-Antipolis (Nov. 2006).
45. A. Mayer, S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington, 'Meaning and Behaviour in Grid Oriented Components,' in *3rd International Workshop on Grid Computing at Grid2002*, Baltimore, Maryland, USA, pp. 100–111 (Nov. 2002) volume 2536 of LNCS.