Object-Oriented SPMD

Laurent Baduel, Françoise Baude, Denis Caromel
INRIA - CNRS - University of Nice Sophia-Antipolis
2004, Route des Lucioles - BP93 - 06902 Sophia Antipolis Cedex France
Email: First.Last@sophia.inria.fr Tel: +33 4 92 38 75 56 Fax: +33 4 92 38 76 44

***Abstract*— This article presents an evolution of classical SPMD programming for clusters and grids.**

It is named "Object-Oriented SPMD" as it is based on remote method invocation. More precisely, it is based on an active object pattern, extended as a typed group of active objects, to which SPMD's specificities are added. The proposed programming model is more flexible: techniques to postpone barrier and to remove any explicit loop make it possible to privilege reactivity and reuse.

The resulting OO-SPMD API has been implemented in *ProActive*. Good scalability and quite competitive performances, compared to what is obtained using C-MPI, are demonstrated.

Keywords: Grid Computing, SPMD Programming, Java, *ProActive*.

I. INTRODUCTION: CONTEXT AND RELATED WORKS

A. SPMD programming

SPMD stands for Single Program Multiple Data. SPMD programming is a common way to organize a parallel program, on both clusters of workstations and parallel machines, and more recently also on grids [1]. A single program is written and loaded onto each node of a parallel computer. Each copy of the program runs independently, coordination events apart. So the instruction streams executed on each node can be completely different, alas for the most common pattern, i.e., master-slave, only two different streams are needed. Each copy of program (process) owns a rank number: a unique ID. The specific path through the code is in part selected by this ID.

Traditionally, in the SPMD model, the language itself does not provide implicit data transmission semantics. In general, the communication patterns are **explicit message-passing** implemented as library primitives. This simplifies the task of the compiler, and encourage programmers to use algorithms that exploit locality. Data on remote processors are accessed exclusively through explicit library calls.

SPMD model maps easily and efficiently to distributed and to parallel applications and distributed memory computing. The most famous environments implementing a message-passing SPMD model are PVM (Parallel Virtual Machine) and MPI (Message Passing Interface).

B. SPMD programming with an Object-Oriented flavour

1) Message-Passing SPMD: In the 1990's, due to the increasing success of object-oriented programming, many research groups have experimented the idea to both combine 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. One of the precursors has been the MPI-2 specification itself, collecting the notions of the MPI standard as suitable class hierarchies in C++, and defining most of the library functions as class member functions. This specification has been further extended in Object-Oriented MPI [2] in order to be able to deal with the transmission of objects. Essentially, OOMPI provides mechanisms to build user-defined data types according to the MPI spec, in order to represent those objects, and further communicating them. Those approaches have been even further developed with the success of Java and have eventually lead to two main categories of propositions for having message-passing SPMD within Java:

- a wrapping of the native MPI implementation library itself within the object oriented language (e.g. mpiJava [3], or JavaMPI [4] where wrappers are automatically generated)
- an *MPI-like* implementation of a message-passing specification as MPI, written using the object-oriented language itself, and available as a library. Notably, MPIJ [5] which seeks to be competitive with native MPI implementations. The most achieved is MPJ [6], in which notions such as Communicators, Datatype for the type of the elements in the message buffers, etc, are modeled as classes.

Overall, in the early 2000's, those works – done under the auspices of the JavaGrande Forum [7] – 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 to conciliate both performance and portability, while not departing from the consensual goal of offering MPI-like services to Java programs.

2) Remote method based SPMD: All propositions grounding up on remote method invocation for communication among activities take for granted that this enables the exchange of any typed data, by automatic marshaling-unmarshaling. Clearly, this better suits to the object oriented paradigm than explicit message-passing, in which send and receive must be explicitly programmed in matched pairs. One work grounding on Java remote method invocation, but generalizing it so it can supports communication between more than two parties is CCJ [8]. Specifically, CCJ aims at adding collective operations to Java's object model (implementing everything on top of RMI). Parallel activities are expressed as threads groups and not as objects groups (in fact, activities in Java are expressed by threads which are orthogonal to objects). 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 group of threads as parameter (in

a similar way as passing the communicator as parameter in any MPI communication). Also, in CCJ, all threads have the same program and, in particular, any collective operation must be called by all threads in the implied group. Differently to the approach followed in CCJ, another concept for collective communications is to group Java objects into *groups*, and extend the remote method invocation mechanism such that it transparently applies to a group of possibly remote objects. It fits much better in the object-oriented approach: triggering the execution of a chunk of code (described in any public method in the class) in parallel is done simply by calling the corresponding method on the group, remotely and possibly asynchronously. By doing this, remote method invocation is exploited as the *only* communication mechanism between any number of remote activities.

Having a group of objects towards which methods are invoked is usually considered to be a suitable OO abstraction for building *distributed* applications – even if it usually requires the additional usage of multicast delivery protocols such as causally or totally order delivery. The suitability of groups and associated group method invocation mechanisms are more rarely studied as a suitable support for parallel computing (notable exceptions being GMI [9] in Java, ARMI [10] in C++).

C. Contribution

In this paper, we propose a pure object-oriented SPMD programming model as an extension of a typed group communication mechanism we previously defined in [11]. For this, the objects groups supporting the distributed computation will also be further organized following a topology, i.e. adding the notion of an ID for each member in the SPMD group and the way to easily reference its neighbors. Collective operations will be revisited and extended with barrier synchronization such as providing a complete *Object Oriented SPMD* model.

The solution we propose is grounded on *ProActive*, a strongly proven programming [12] and deployment model for distributed object-based computations, on any distributed memory platforms including grids [13], [14]. *ProActive* is based on the active object paradigm, and moreover featuring a well defined semantics of the computing model [15]. In this respect, the SPMD programming solution we define is a smooth and perfectly integrated extension of the active object principle. We want to demonstrate to the programmer that using it, he can define programs grounded on a single concept, the *active object*. Using this paradigm, he can seamlessly target the whole spectrum of applications: from sequential mono-threaded, concurrent and multi-threaded, distributed, up to parallel and distributed ones.

To our knowledge, a proposition which is close to ours is GMI [9], (in the objective and in the way to achieve it). A strong difference comes from the fact that GMI generalizes Java RMI. As such, it is confronted with its constraints, specially, the need for the programmer to take care of possible concurrent executions of a same method (implying to mix functional code with the usage of regular Java monitor mechanisms). On the contrary, the active object pattern is a cleaner abstraction for distributed computing, and as such should ends up easier for programing Object-Oriented SPMD applications.

Section II presents briefly the *ProActive* library. Section III presents the typed group communication of *ProActive* and the recent optimizations we have added to it. Section IV introduces the complete Object-Oriented SPMD programming model. One strong advantage is that the corresponding API is very light: only primitives for SPMD group membership and barrier synchronizations are required. Indeed, all point-to-point and collective communications are implicit as the focus is more on which method to execute in parallel instead of how to effectively manage the parallel and distributed associated aspects. Section V presents benchmarks on large configurations, including comparisons with MPI. Section VI concludes.

II. The *ProActive* library

ProActive is an LGPL Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, *ProActive* provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Network (LAN), on clusters, or on grids.

As *ProActive* is built on top of the Java standard API[1], it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine.

a) Base model: A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [12]. Contrary to classical RMI, all kinds of method call parameters towards an active object are passed by (deep-)copy. 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. The *ProActive* library provides a way to migrate any active object from any JVM to any other one through the `migrateTo(...)` primitive which can either be called from the object itself or from another active object through a public method call.

b) 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 in 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 several nodes. The traditional

[1] mainly Java RMI and the Reflection API

way to name and handle nodes in a simple manner is to associate them with a symbolic name, that is a URL giving their location, for instance `rmi://lo.inria.fr/node`.

Let us take a standard Java class `A`. The instruction:

```
A a = (A) ProActive.newActive("A",params, N1);
```

creates a new active object of type `A` on the JVM identified with `N1`, for instance `rmi://lo.inria.fr/node`. Further, all calls to that remote object will be asynchronous, and subject to the *wait-by-necessity*:

```
a.foo (...);         // Asynchronous call
v = a.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, the concept of virtual nodes as entities for mapping active objects has been introduced [13]. Those virtual nodes are described externally through XML-based deployment descriptors which are then read by the runtime when needed. 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 mean to abstract from the source code of the application any reference to software or hardware configuration. It also provides an integrated mechanism to specify external processes (e.g. JVM) that must be launched, and the way to do it.

III. Typed group communications

The group communication mechanism of *ProActive* efficiently achieves asynchronous remote method invocation for a group of remote objects, with automatic gathering of replies.

A. Summary of group communication

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

On the standard Java class `A` used above, here is an example of a typical group creation:

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

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

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

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

Such a call is propagated to all members of the group using multithreading: a method call on a group yields 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 broadcasted to all the members of the group.

An important specificity of the group mechanism is: the *result* of a typed group communication *can also be a group*. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically updated with the incoming results, thus gathering results, as shown in Figure 1: *"result group"*. The *wait-by-necessity* mechanism is also valid on groups: if all replies are awaited then the caller blocks, but as soon as one reply arrives in the result group the method call on this result is executed. E.g. in

```
   // A method call on a group with result
V vg = ag.bar();
   // vg is a typed group of "V"
   // This is also a collective operation:
vg.f();
```

a new `f()` method call is automatically triggered as soon as a reply from the call `ag.bar()` comes back in the group `vg` (dynamically formed). The instruction `vg.f()` completes when `f()` has been called on all members: 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 `ag` have executed the method `bar()`. Another remark is that collected results, and thus *gathered* through the `vg` group can subsequently be merged. This is like achieving a global reduction. The reduction operator can be any user defined method (such as `f()` in the above example), and moreover, the operator can be applied as soon as each result comes back. So, even if the reduction operation is not executed in parallel, its cost can be hidden by the transmission of the not yet arrived results.

Other features are available regarding group communications: parameter dispatching using groups (through the definition of *scatter groups*), hierarchical groups, dynamic group manipulation (`add`, `remove` of members), explicit group synchronization (`waitOne`, `waitAll`, `waitAndGet`); see [11] for further details and implementation techniques.

B. Optimizations

The group communication mechanism is built upon the *ProActive* elementary mechanism for asynchronous remote method invocation with automatic future for collecting a reply. As this last mechanism is implemented using standard

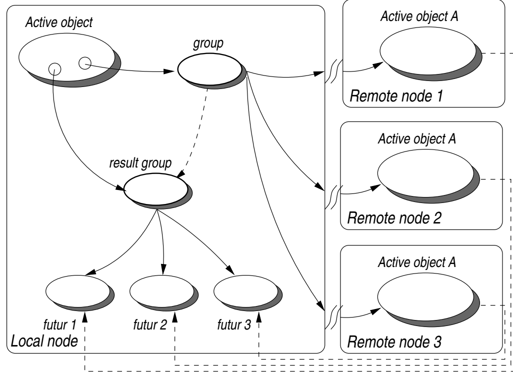

Fig. 1. Method call on group

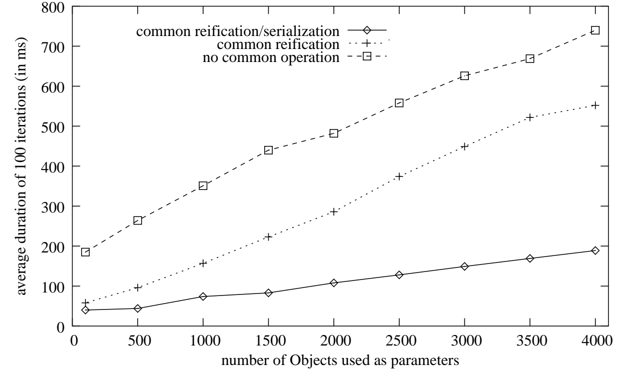

Fig. 2. Factorization of common operations

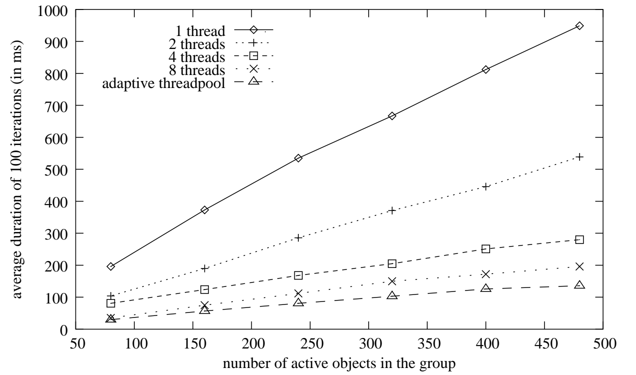

Fig. 3. Adaptative threadpool

Java, such as RMI, the group mechanism is itself platform independent. A group communication must be thought as a replication of more than one (say N) *ProActive* remote method invocations towards N objects. Of course, we incorporated optimizations into the group mechanism implementation.

a) Common operations factorization: Many operations are common while invoking a method on a group of objects. Those operations may be factorized. First is the reification operation that transforms the method invocation into a Java object using the Meta Object Protocol. This operation involves reflection techniques that are known to be expensive. The method being the same for all group members, the operation is done just once.

Second point subject to factorization is the serialization of the method parameters sent during the group communication. As the serialization process is very slow, we want to avoid the repetition of this operation. Before the RMI mechanism steps in, the parameters (and codebase informations) are converted into a byte array to be more efficiently sent several times by RMI. This does not apply in the case of scatter group in which parameters for each member differ.

Figure 2 presents the average time (in milliseconds) spent to perform one hundred method invocations depending on the amount of data to send (objects used as parameters). The group contains 80 objects distributed on 16 machines (cluster of PIII @ 933MHz interconnected with a 100Mb/s ethernet network). The upper curve exposes the performances without any operation factorized. The curve in the middle plots the performances obtained by factorizing the reification operations. The last curve represents the performances obtained by factorizing the reification operations and the serialization. This allows better performances (up to a 3.9 ratio in the Figure 2).

b) Adaptive threadpool: Using several threads allows to send messages simultaneously. Doing this way, the delays required by RMI to make the rendez-vous with each remote object are recovered and no more added. In order to maintain the *ProActive* method invocation semantic, we introduce a synchronization. We extend the notion of rendez-vous for group communication: doing this, an asynchronous group communication blocks until the method invocation has reached all group members.

Because group membership is dynamic, a fixed number of threads used to communicate with the group members is not appropriate. Likewise, a one-to-one group member / thread ratio is not suitable: too many threads will harm performances. Our solution is to associate to a group an adaptive pool of threads. The member / thread ratio may be defined by the programmer depending of the requirement of its application (default value is 8).

Figure 3 plots the average time (in milliseconds) spent to perform one hundred method invocations depending on the number of objects in a group. The group members are distributed on 16 machines (cluster of PIII @ 933MHz interconnected with a 100Mb/s ethernet network). The curves represent the performances depending on the number of threads used to make the calls. The more we used threads the smaller is the delay to make the group communication. The four upper curves are associated with a fixed number of threads. The lower one is associated with a dynamic number of threads. It shows better performance, because the number of threads is (automatically and transparently) at any moment the adequate number needed.

IV. Object-Oriented SPMD

The proposed active objects group mechanism presented in section III is already a usable and even efficient basis to program non embarrassingly parallel applications using a pure object-oriented paradigm, i.e. using only object-oriented method invocation for e.g. computational electromagnetism [14], [16]. But, some of the features specific to SPMD programing were lacking, and their addition constitutes the core

of this section. We name the resulting proposition as *Object Oriented SPMD* (OO-SPMD for short).

A. Requirements

These specificities fall into three categories:

- identification of each member taking part in the parallel computation, and concept of member position relatively to the others (i.e. neighboring relation among members)
- expression of the program run by each member taking part in the parallel computation. In pure object groups based paradigms (e.g. as GridRPC for grid computing on Network Enabled Servers like NetSolve [17] or Ninf [18]), members act in a sense as *passive* servers only activated by method calls triggered by clients. Servers do not have their own activity. On the contrary, in SPMD computing, all members are active by their own even if, for simplicity, they all execute the same program (e.g., in all flavors of MPI, in CCJ [8], in GMI [9], this program is run by the main thread on each process or participating JVM). In *ProActive*, each active object is by essence the support of a proper activity (there is no *main*, but a *runActivity* method). This activity aims at enacting the sequential service of requests (see paragraph II.a). So, in our approach, the SPMD program will not be expressed as a classical *big loop*, but as the implicit result of a succession of request services executed in FIFO order. As will be emphasized below, this way of expressing the core of any member's SPMD program enables behaviors pertaining to reactivity, evolutivity, dynamicity usually considered to be far away from 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 III, only the expression of global synchronization barriers is lacking and so needs to be considered below.

B. Main principles of OO-SPMD

An OO-SPMD group is defined as follows: it is a group of active objects where each member has a reference, a group proxy, towards the group itself (see Figure 4). Each active object in the SPMD group is also provided with a specific *rank* in the group.

```
  // A group of type "A" and its members
  // are created at once by an external
  // active object
Object[][] params = {{...}, {...}};
A ag = (A) ProSPMD.newSPMDGroup("A",
                   params, {Node1,...});
  // The computation on each member may
  // now be started, i.e. invoking a method
  // called e.g compute() defined in class A
ag.compute();
```

On each group member created, one of the first actions to run is to get the reference of the group it belongs to, the rank, etc. 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. This last one enables the active object to implement the parallel task. Traditionally in SPMD, the parallel task is expressed as an iterative or recursive loop, which essentially handles message receptions and triggers the corresponding treatment, according to the message's tag (a `case` or a `if` control structure is usually programmed). In OO-SPMD, the parallel task on any member of the SPMD group is run by repeatitively invoking asynchronous methods to itself (so, the need to have an asynchronous reference). A member triggers data receptions and the corresponding treatment through the asynchronous service of methods remotely called by other members in the group. All method services are FIFO-ordered.

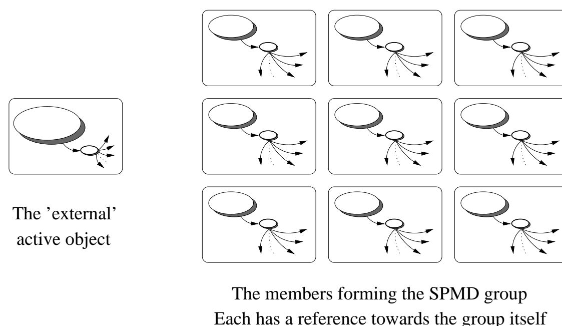

Fig. 4. An SPMD group

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

Moreover, 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 be also based on dynamically created groups at any moment at runtime. Such groups can be derived from existing ones (subgroups, or group combination for instance) or according to any kind of properties (rank, fields of the object, ...).

C. Topologies

To simplify the access to neighbors 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 (as in MPI). At this time, we offer the following: line, plan, ring, cube, hypercube, torus, torusCube (torus in 3 dimensions) but, contrary to statically designed topologies, the addition of new topologies is open. Topologies may also be obtained from an other topology. Here is an example:

```
  // Organize my group as a 2D plan
Plan topology = new Plan(a, WIDTH, HEIGHT);
  // Get a ref. to my neighbors in the plan
A left = (A) topology.left(me);
```

```
A down = (A) topology.down(me);
...
  // One-way communication with neighbors
  // in an asynchronous fashion
left.foo(params);
down.foo(params);
...
  // Get a ref. to the topology formed by
  // the first line of the plan
Line line = topology.line(0);
```

The notion of neighborhood is strongly attached to the topology. By extending a topology, the programmer may redefine the neighborhood to best fit the needs of the application.

D. Synchronization barriers

The only collective behavior related methods of our OO-SPMD API pertain to global barriers. Indeed, as already explained in section III, all collective (resp. point-to-point) communications within the group can be expressed as applicative-level method calls triggered via the group proxy (resp. via the asynchronous reference of the target member).

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 while not 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) only in the future: more precisely 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, so care must be taken (see an example in subsection V-B). Nevertheless, the meaning of what is a global synchronization barrier is as usual, but instead of pertaining to the next instruction, it pertains to the next request's service: when encountering a barrier, the service of the first request waiting in the request queue will be able to proceed on any enrolled member only when all have reached the barrier.

Technically, when an active object executes a call to a global barrier this triggers the storage in the front of its request queue of a specific token. Associated to this token is the total number of members (including the member itself) to wait for, i.e. that must reach the barrier. Each time a given global barrier is reached by a member, this triggers the decrementation of this number on each member enrolled in the barrier. Eventually, the barrier is released on each enrolled member, as soon as the number reaches zero.

Actually, we propose three kinds of barriers, two globals and one more local:

- 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, that 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 parameter.
  `ProSPMD.barrier("Bar", neighborsGroup);`
- 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 the two others.
  `ProSPMD.barrier({"foo","bar","gee"});`

V. EXAMPLE AND BENCHMARKS

We illustrate OO-SPMD with a concrete example. We choose *Jacobi iterations* because it is a simple application, 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; it means until the difference between old and new value in absolute becomes lower than a given threshold.

The following code shows the main loop (an iteration based loop) of a solver. At each 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 boundary values from adjacent sub-matrixes and computing its own part of data.

The parallel algorithm depends of the data distribution scheme. We choose a two-dimensional distribution scheme. As shown in figure 5, communications occur at block boundaries. So the amount of data exchanged is minimized by the two-dimensional distribution which has a better internal area / border ratio. With this partition, each sub-matrix may communicate with two, three, or four neighbors, depending of their position (respectively at a corner, a border, or in the center of the whole matrix). This partition is more effective when the data to processor ratio is large.

Communications appear at sub-matrix boundaries to send boundaries values to neighbors and receive values of neighbors. A copy of the boundary of each sub-matrix is present in its neighbor sub-matrix. Storage of boundary data is allocated at the producer, and at the consumer sub-matrixes. This is a static allocation because the size and the location of boundary

Fig. 5. Distributed algorithm

buffers is fixed and never evolve during Jacobi. This is induced by the Jacobi algorithm itself, but if needed, our framework could support strongly dynamic algorithms.

A. MPI Jacobi

Using a message passing approach based on asynchronous send and receive, with the MPI library, the resulting parallel code is something like:

```
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);
    // 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 the same.

B. OO-SPMD Jacobi

Using our OO-SPMD approach, the code becomes much 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.

```
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), otherwise they would be served immediately, i.e. before the execution of the barrier. Overall, according to the semantic of the *method barrier*, the data (i.e. parameter of `send`) will have been exchanged before the barrier will be released, guarantying that any 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 values `boundariesGroup`): as the real parameter of the *send* method is a group declared as of type *scatter*, 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 specified on this chunk of code. It only consists of building a group containing the boundaries.

A very interesting property of our model is that it remains *reactive*. It means that any part or any member of an OO-SPMD 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, on one side a parallel object-oriented SPMD computation, and on the other side, an external and remote application that is in charge of, for instance, steering, visualization, etc.

C. Benchmarks

1) Data scalability: The first benchmark, presented by the Figure 6, uses a cluster of 16 bi-Pentium III @ 933 Mhz 512MB (SDRAM) - 256 Kb L2 cache, Linux RedHat 2.4.20, interconnected with a 100 Mb/s Ethernet. Even if machines are bi-processor, we used only one processor per machine during our experimentations. 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 data contained on each node, in millions of double. The average time was computed after 100 iterations. Of course, the C language with MPI remains more efficient than Java with RMI. But the ratio of 3.3 (average) of performance is maintained despite the growth of data (see the bold curve). It is interesting to notice that 3.3 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, speedup of the C/MPI version is 15.41, speedup of the Java/OO-SPMD is 15.23.

2) Deployment scalability: Figure 7 presents the Jacobi application running on up to 130 machines. This experimentation was done using a Peer-to-Peer deployment scheme provided by *ProActive* within an Intranet configuration. The machines used are desktop computers, simultaneously used by their users. They are heterogeneous (slowest is a Pentium III @ 993 MHz 512MB, fastest is a bi-Pentium IV @ 3,2 GHz 2GB), they are interconnected with 100 Mb/s network, they are running under Linux (with different kernel versions). Deployed applications run with a lower priority (nice 19) in order to not disturb regular users. We used the Sun Java Virtual Machine 1.4.2.

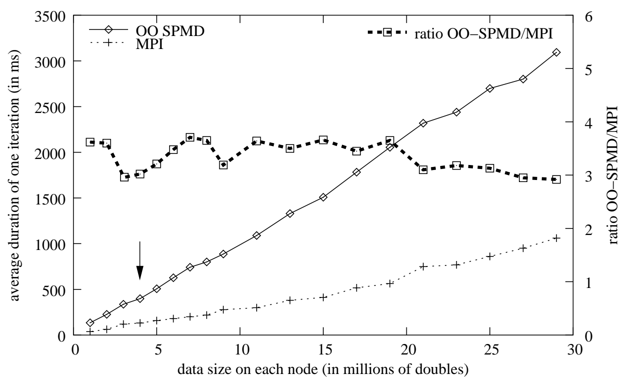

Fig. 6. Benchmark: C/MPI and Java/OO-SPMD versions

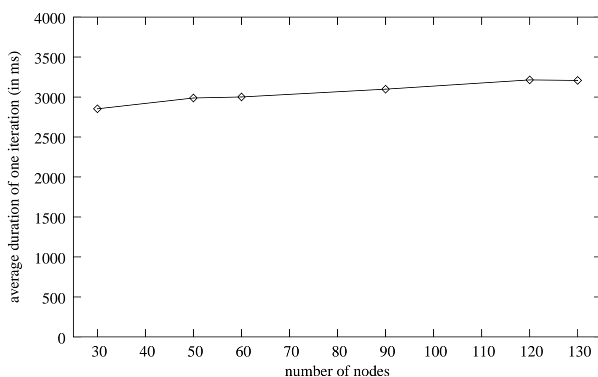

Fig. 7. OO-SPMD scalability in a peer-to-peer experiment

For all measurements, each node is responsible for the same amount of data (2000x2000 doubles). The overall size of the problem grows with the number of nodes involved in the computation. The line plots the average duration, in milliseconds, of one Jacobi iteration depending of the number of nodes involved. As previously, the average time was computed after 100 iterations. Compared to the previous benchmarks, for the same amount of data per node (see arrow in Figure 6), execution is 7 times slower. We blame the lower priority of execution and the older JVM for this loss of performance. Besides, the performance remains regular, regardless of the number of used nodes. From this, we conclude that the application is scalable.

VI. CONCLUSION

We have introduced a parallel programming model, which we name *Object-Oriented SPMD* as an alternative to the traditional Message-Passing SPMD style. Overall, it allows more flexibility, and a higher level of abstraction. First, it enforces members taking part in the computation just the required involvement in collective operations. E.g. in MPI, a call to `MPI_broadcast` must be run by all members, even if for all except the sender, this call aims only at receiving the message. On the contrary, using our solution, a method invocation towards a single active object to trigger a point-to-point interaction, or towards a SPMD group of active objects to trigger a collective interaction between all the members only differ by the target object reference. This way, we promote asychronous remote method invocation and the active object pattern as the only required communication and structuration mechanism. Secondly, our approach to SPMD programming has potential for evolution. Instead of defining the parallel task as a single 'big' loop as in traditional SPMD programming, OO-SPMD enables to receive and treat data in a more flexible order (discarding the need to program sometimes intricate `case` statements depending of received message's tag).

The resulting OO-SPMD API already forms part of the *ProActive* open-source library, freely distributed through the Object Web consortium for open-source middleware. Our ambition is to have this approach used on real size applications. We already successfully applied the typed group communication mechanism to solve simulation in electromagnetism [14], [16]. Our current work is to apply the whole OO-SPMD approach to it. Next, we plan to target other application domains, such as biogenetics (applying BLAST in parallel), for which we already have developed applications, but not yet using OO-SPMD.

REFERENCES

[1] G. Fox, M. Pierce, D. Gannon, and M. Thomas, "Overview of grid computing environments," Global Grid Forum, Tech. Rep., 2002.
[2] J. Squyres, B. McCandless, and A. Lumsdaine, "Object Oriented MPI: A Class Library for the Message Passing Interface," in *POOMA'96*, http://www.osl.iu.edu/download/research/oompi/oompi.pdf.
[3] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, "mpiJava: An Object-Oriented Java interface to MPI," in *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*.
[4] S. Mintchev and V. Getov, "Towards portable message passing in Java: Binding MPI," in *Recent Advances in PVM and MPI*, ser. LNCS, no. 1332, 1997.
[5] G. Judd, M. Clement, and Q. Snell, "DOGMA: Distributed Object Group Metacomputing Architecture," *Concurrency: Practice and Experience*, vol. 10, no. 11/13, 1998.
[6] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox, "MPJ: MPI-like message passing for Java," *Concurrency: Practice and Experience*, vol. 12, no. 11, pp. 1019–1038, 2000.
[7] "Java Grande Forum," www.javagrande.org.
[8] 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, June 2001, pp. 11–20.
[9] J. Maassen, T. Kielmann, and H. Bal, "GMI: Flexible and Efficient Group Method Invocation for Parallel Programming," in *LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, 2002.
[10] S. Saunders and L. Rauchwerger, "ARMI: An Adaptive, Platform Independent Communication Library," in *PPoPP'03*.
[11] L. Baduel, F. Baude, and D. Caromel, "Efficient, Flexible, and Typed Group Communications in Java," in *Joint ACM Java Grande - ISCOPE Conference*. Seattle: ACM Press, 2002, pp. 28–36.
[12] D. Caromel, "Towards a Method of Object-Oriented Concurrent Programming," *Communications of the ACM*, vol. 36, no. 9, pp. 90–102, September 1993.
[13] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière, "Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications," in *11th IEEE International Symposium on High Performance Distributed Computing HPDC-11*, 2002, pp. 93–102.
[14] L. Baduel, F. Baude, D. Caromel, C. Delbe, N. Gama, S. E. Kasmi, and S. Lanteri, "A parallel object-oriented application for 3d electromagnetism," in *IEEE International Symposium on Parallel and Distributed Computing, IPDPS*, april 2004.
[15] D. Caromel, L. Henrio, and B. Serpette, "Asynchronous and deterministic objects," in *31st ACM Symposium on Principles of Programming Languages*, 2004.
[16] F. Huet, D. Caromel, and H. E. Bal, "A High Performance Java Middleware with a Real Application," in *SuperComputing 2004*.
[17] "NetSolve," http://icl.cs.utk.edu/netsolve.
[18] "Ninf-G," http://ninf.apgrid.org/.