A Parallel Object-Oriented Application for 3D Electromagnetism

Laurent Baduel, Françoise Baude, Denis Caromel, Christian Delbé,
Nicolas Gama, Said El Kasmi, Stéphane Lanteri

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr

Abstract

Within the trend of object-based distributed computing, we present the design and implementation of a numerical simulation for electromagnetic waves propagation. A sequential Java design and implementation is first presented. Further, a distributed and parallel version is derived from the first, using an active object pattern. In addition, benchmarks are presented on this non embarrassingly parallel application.

A first contribution of this paper resides in the sequential object-oriented design that proved to be very modular and extensible; the classes and abstractions are designed to allow both element and volume type methods, furthermore, valid on structured, unstructured, or hybrid meshes. Compared to a Fortran version, the performance of this highly modular version proved to be in the same range.

It is also shown how smoothly the sequential version can be distributed, keeping the same structuring and object abstractions, allowing to deal with larger data size. Finally, benchmarks on up to 64 processors compare the performances with respect to sequential and parallel versions, putting that in perspective with a comparable Fortran version.

***Keywords**: object-oriented and distributed computing, active object, numerical simulation, Java*

1 Introduction

Within the trend of object-based distributed computing, we present the design and implementation of a numerical simulation for electromagnetic waves propagation.

The general objective of this collaboration between computer scientists and applied mathematicians, is to use modern programming languages and libraries such as Java and *ProActive*[1], for the design of a problem solving environment (PSE) for complex applications in the bio-electromagnetics field. Such an environment will ideally integrate software components for geometric modeling from medical images, unstructured grid generation, numerical simulation and scientific visualization. An example of such an environment based on Corba is given in [24]. While such work uses the option of wrapping legacy code, the work presented here concentrates on a full-fledged object-oriented version, for the sake of extensibility and adaptability. Overall, from an existing Fortran application (EM3D [22]), we designed a modular and extensible object-oriented version: Jem3D.

First we define an object-oriented model of a 3D code in Java, and we use it for programming a sequential version. In order to take advantage of parallelism and distribution, we then use *ProActive* a library for parallel, concurrent and distributed computing in Java featuring additional characteristics compared to the standard Java RMI API. We have deliberately chosen not to use an explicit message-passing library (MPI, or Java version of it like MPJ [6], or MPIjava [25]) for taking advantage of distribution: we aim at enforcing code reuse by applying the remote method invocation mechanism instead of explicit message-passing.

As might be predicted, actual benchmarks show slower performances of about a factor of 3 compared to those of a Fortran-MPI version, which is undoubtely a good achievement in a 100% Java environment. More importantly, the aim of this work is to emphasize on the benefits we get on software engineering aspects (possible extension of the Java version, full portability, ease of deployment, etc) through a complete rewriting of the Fortran version. Recent works such as [14] also mention the advantages of using object-oriented practices for finite element analysis; the main difference is that we do not rely on direct parallel solvers. We do not get the performance of executing native code resulting from Fortran or C++ programming; see for instance

[1]ProActive is available in LGPL at http://ProActive.ObjectWeb.org

works that wrap MPI-based legacy codes as Java or Corba components [17, 7]. Even under those conditions, an object-oriented SPMD programming approach entirely based on point-to-point or collective method invocations in Java, we still get good performances and speedup.

2 Related work

2.1 EM3D : a parallel solver for electromagnetic waves propagation

The Fortran EM3D software has been designed for the numerical simulation of electromagnetic waves propagation in the time domain. The software numerically solves the 3D Maxwell equations for homogeneous or heterogeneous linear media. It relies on a finite volume time domain (FVTD) method designed on unstructured tetrahedral meshes, potentially applicable to general hybrid meshes. The FVTD method adopts a cell centered formulation[2] (a control volume is taken to be a tetrahedron) with a centered numerical scheme for the computation of convective fluxes, combined to an explicit leap-frog time integration scheme. The resulting solver is second-order accurate in time and space for regular meshes, and provides unsteady solutions that conserve a certain form of discrete electromagnetic energy [22]. It is interesting to note that such finite volume formulations were originally designed for computational fluid dynamics, such as 3D Euler or Navier-Stokes equations [16]. This clearly motivates the development of a general object-oriented framework that would facilitate the development of various simulation softwares for PDEs (Partial Differential Equations).

Finally, the parallelization of EM3D combines a domain partitioning strategy with a message passing programming model using the MPI (Message Passing Interface). The partitioning of the computational domain is obtained with the ParMETIS tool [15].

2.2 Other object-oriented numerical simulation softwares

During the last ten years, there has been a large number of projects related to the use of object-oriented approaches in the context of numerical calculation especially for grid based applications such as grid generation, grid adaptation and numerical solution of PDEs. Another computational field that has largely benefited from object-oriented programming is concerned with algebraic (linear or non-linear) system solvers such as TNT[3], Hypre [8], JAMA[4] and PETSc [23] (among others).

Concerning grid based processing and numerical solution of PDEs, analysing all the existing projects would be far too long for a conference paper. We simply outline here a number of these projects. Otherwise stated, the projects discussed below have adopted C++ as the programming language.

GrAL (Grid Algorithms Library) [3] is a generic library for grid oriented data structures and algorithms. GrAL was initially developed for applications dealing with the numerical solution of PDEs. However, other possible application areas for GrAL include computational geometry and topology, geometric modeling, computer graphics, or geographic information systems. Mouse [12] is an object-oriented framework for Computational Fluid Dynamics (CFD) computations using finite volume methods on unstructured grids. Mouse has been designed with the aim to facilitate the use of unstructured grids. Mouse is not restricted to CFD, and not limited to the use of finite volume methods, although it offers various classes that ease the implementation of this type of methods. The basic data structures, which are not trivial for unstructured grids, could be used for other types of discretizations as well. Overture [4] is another example of an object-oriented framework dedicated to the numerical solution of PDEs. It provides a portable, flexible software development environment for applications that involve the simulation of physical processes in complex moving geometry. It is implemented as a collection of C++ libraries that enable the use of finite difference and finite volume methods at a level that hides the details of the associated data structures. Overture is designed for solving problems on a structured grid or a collection of structured grids. In particular, it can use curvilinear grids, adaptive mesh refinement, and the composite overlapping grid method to represent problems involving complex domains with moving components.

JMP [13] is a linear algebra object-oriented library, implemented in pure Java. It is aimed at the numerical solution of large linear systems arising from the discretization of PDEs. It can deal both with dense and sparse matrices, offers the access to iterative Krylov methods and standard factorization methods and includes a comprehensive BLAS for both types of matrices with parallelization for the most time-consuming operations. The parallelization of JMP uses threads and is thus only effective on shared memory systems. Clearly, JMP could be used in conjunction with Jem3D for the solution of the linear systems resulting from the adoption of implicit time integration schemes. Such schemes are not implemented in the current version of Jem3D and are still the subject of theoretical studies. However, the main drawback to such a coupling between Jem3D and JMP certainly is the shared memory parallelization of the latter. One solution to this problem could be the re-

[2]Other widely used finite volume methods rely on a vertex centered formulation.

[3]Template Numerical Toolkit, http://math.nist.gov/tnt/index.html

[4]Java Matrix Package, http://math.nist.gov/javanumerics/jama/

writing of JMP in the *ProActive* framework.

An approach similar to the one considered here is presented in the context of composite material manufacturing [14]. In this paper, the authors report on the development of an object-oriented version of a simulation environment named COMPOSE that was originally designed using a classical procedural programming paradigm in Fortran 90. COMPOSE is a parallel finite element software that relies on an implicit time integration scheme which is not subjected to a stability restriction as it is the case with the explicit leap-frog scheme adopted in Jem3D. However, the price to pay for this unconditional stability is the solution of large sparse linear systems which is critical to the overall performances of the software. The object-oriented version of COMPOSE relies on the SPOOCEFEM computing environment that has been designed to simplify the development of parallel finite element software. SPOOCEFEM has been coded in ANSI C++. As with Jem3D, SPOOCEFEM implements a hierarchy of finite-element classes that contains all of the generic finite-element functionnalities required for developing a complete application. Moreover, for a number of specific tasks, SPOOCEFEM makes use of external, special purpose, libraries: PETSc [23] and PSPASES [18] packages that give access to efficient parallel direct and iterative solvers as illustrated by the performance results presented in [14]. Finally, parallel programming relies on MPI.

In summary, with regards to the projects discussed above, the distinctive features that characterize Jem3D are (1) the Java programming language and, (2) the adoption of a distributed computing model that makes it amenable both to cluster and grid computing.

3 Jem3D: an Object-Oriented library for 3D electromagnetic-based simulations

In this section, we describe the main features of a general object-oriented model that can be used for the development of simulation software tools which are based on finite volume type methods on unstructured meshes. The application of this model is currently limited to the Maxwell equations for electromagnetic waves propagation but it can be extended to deal with Euler or Navier-Stokes equations that model compressible flow calculations [16]. Moreover, the model could also be extended to include classical finite element type discretization methods. The main features of the model are: (1) the ability to deal with 2D and 3D computational domains, (2) the possibility of choosing between different types of discretization elements (triangle, quadrangle, tetrahedron and hexahedron) and, (3) the inclusion of the two main classes of finite volume methods i.e. the vertex centered and element centered formulations. As the first application developed using this model/library is a Java version of the EM3D solver, we name *Jem3D* the application.

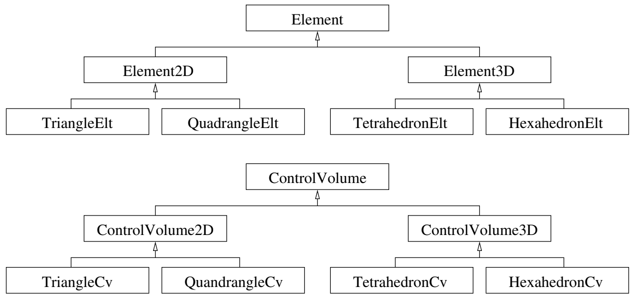

Figure 1. Definition of an element and a control volume in 2D and 3D

3.1 Basic architecture

The proposed object-oriented model essentially consists of two types of classes: classes that are concerned with the definition of the geometry (or computational domain) and classes that are related to the application. Currently, the definition of the latter includes geometric, physical and numerical components. In other words, these classes are strongly linked to the physical context under consideration (electromagnetic waves propagation in the present case).

3.2 Geometry definition

The finite volume methods adopted in [22] and [16] rely on the use of an unstructured mesh for the discretization of the computational domain. The construction of such meshes can be based on various types of discretization elements. The standard situation is such that only one type of discretization element is considered for the definition of a given unstructured mesh. However, in the general case, the computational domain could be discretized by combining several types of elements (hybrid discretization). The classes considered here are concerned with the definition of the discretized geometry with an unstructured mesh. In order to do so, one essentially needs two basic geometric entities: the vertex and the element. The element is used to connect a number of vertices and an unstructured mesh is defined by filling the computational domain with elements. These two geometric entities are included in our object-oriented model through the definition of several classes: *Vertex2D* and *Vertex 3D* (which extends *Vertex2D*) are simple concrete classes for the definition of a vertex in 2D and 3D; *Element*, *Element2D* and *Element3D* are abstract classes for the definition of an element in 2D and 3D (see Figure 1).

3.3 Application aspects

Starting from the discretized geometry, it is then necessary to define the classes related to the numerical methods (finite volume methods in the present case). Finite volume methods typically yield the computation of a flux balance through the boundary of a control volume (also called a cell). Indeed, a control volume is a geometric entity that can be seen as another building block for the discretization of the computational domain but it is not the natural basic entity for the definition of an unstrutured mesh. In some sense, it is introduced artificially since it represents the calculation support of finite volume type methods. Note that for finite element type methods, the calculation support is simply given by the element. In the finite volume framework, the unknowns of the problem are averages of the phyical quantities computed over control volumes while in finite element methods, the unknowns are the values of the phyical quantities associated to the vertices of the mesh.

In our object-oriented model, the control volume is defined through a hierarchy of classes partially shown in Figure 1. At that point, it is worthwhile to make two remarks:

- as for the vertex and the element entities, the definition of the control volume includes classes dedicated to the 2D and 3D cases. In addition, we have taken into account the two main families of finite volume methods i.e. the vertex centered and element centered formulations. In a vertex centered formulation, a control volume is constructuted around a vertex using partial contributions from the set of elements attached to this vertex. In an element centered formulation, the control volume is simply taken to be an element (triangle, quadrangle, tetrahedron or hexahedron). In Figure 1, the latter formulation is illustrated with the choice between *HexaedronCv* and *TetrahedronCv*.

- In practice, the flux balance is evaluated as the combination of elementary fluxes computed through a series of facets that describe the boundary of the control volume. This yields another hierarchy of classes for the definition of various types of facets (see Figure 2).

Finally, we note that the EM3D solver is based on an element centered formulation where the control volume is a tetrahedron and the facet is a triangular face. Therefore, at the lowest level of the hierarchy of classes for the definition of a facet, we currently have the various types of triangular faces that are considered in the EM3D solver: either an internal face or a boundary face and, for a boundary face, several subclasses corresponding to the different types of boundary conditions.

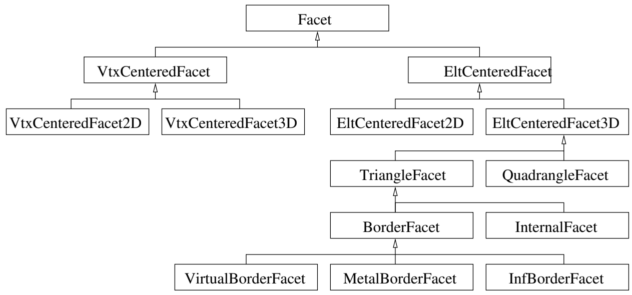

Figure 2. Definition of a facet in 2D and 3D

3.4 Overall skeleton and control

The overall skeleton of the EM3D solver is shown in Figure 3 and so will be reproduced as such in Jem3D. Let $t^n = t^0 + n\Delta t$, $\mathbf{E}$ and $\mathbf{H}$ respectively denote the discrete time, the discrete electric field and the discrete magnetic field (both fields are vectors of size $3 \times N_{\text{CV}}$ consisting of the x, y and z components of the physical quantity computed on each control volume). In the leap-frog time integration scheme adopted in EM3D, each time step allows the calculation of $\left(\mathbf{E}^{n+\frac{1}{2}}, \mathbf{H}^{n+1}\right)$ from $\left(\mathbf{E}^{n-\frac{1}{2}}, \mathbf{H}^{n}\right)$. In practice, the main time stepping loop of Figure 3 is decomposed in three phases:

1. the flux balance for the magnetic field is computed from the distribution of the magnetic field obtained at the previous time step (i.e. $\mathbf{H}^n$). This flux balance is used to update the electric field (i.e. to compute $\mathbf{E}^{n+\frac{1}{2}}$ from $\mathbf{E}^{n-\frac{1}{2}}$ using the flux balance for $\mathbf{H}^n$).

2. the flux balance for the electric field is computed from the distribution of the electric field resulting from the previous phase (i.e. $\mathbf{E}^{n+\frac{1}{2}}$). This flux balance is used to update the magnetic field (i.e. to compute $\mathbf{H}^{n+1}$ from $\mathbf{H}^n$ using the flux balance for $\mathbf{E}^{n+\frac{1}{2}}$).

3. the discrete electromagnetic energy (which is a scalar value) is computed from the distributions $\mathbf{E}^{n+\frac{1}{2}}$ and $\mathbf{H}^{n+1}$. This particular quantity is used to monitor the simulation in the sense that, according to the results of the theoretical analysis [22], it should remain constant.

The first and second phases are implemented using loops over the lists of triangular faces (see the discussion on the facet entity in subsection 3.3) using different numerical schemes for the calculation of fluxes through internal and boundary faces. Since the original EM3D code is programmed in Fortran 77, the informations related to the definition of internal and boundary faces (as well as for vertices and tetrahedra) are stored using array data structures.

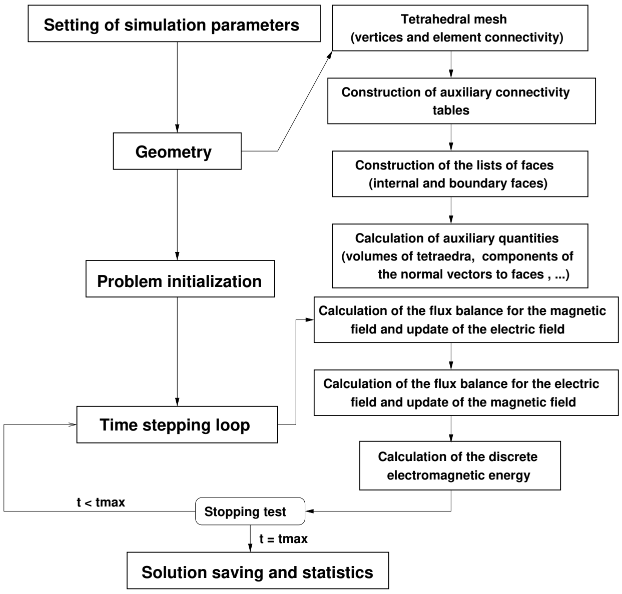

Figure 3. Overall application skeleton

In the Java version of EM3D, lists of vertices, elements, control volumes and facets are implemented using the `ArrayList` class from the standard Java API. `ArrayList` is a resizable-array implementation of the `List` interface. Like an array, it contains components that can be accessed using an integer index. However, the size of a `ArrayList` can grow or shrink as needed to accommodate adding and removing items after the `ArrayList` has been created. This class is equivalent to `Vector` except that it is unsynchronized: it permits simultaneous and faster access.

The components of our object-oriented model as described in subsections 3.2 and 3.3 can be viewed as contributing to a general library on top of which a particular application can be built. Such an application based upon a vertex centered formulation has yield to Jem3D, the Java version of the EM3D solver presented in subsection 3.4, but several others could be considered in the future.

4 Design of the parallel and distributed version of Jem3D

This section explains how, using the *ProActive* library, we have programmed a parallel and distributed version of Jem3D starting from the sequential one.

4.1 The *ProActive* library

As *ProActive* is built on top of the Java standard API[5], 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.

[5] mainly Java RMI and the Reflection API

4.1.1 Base model

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has 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* [5]. There is a short rendezvous 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.

4.1.2 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 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 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/Node1`.

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

```
A a = (A) ProActive.newActive("A",
         params,"rmi://lo.inria.fr/node");
```

creates a new active object of type `A` on the JVM identified with `Node1`. 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
```

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 [2]. Those virtual nodes are described externally through XML-based descriptors which are then read by the runtime when needed.

4.1.3 Group communications

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

Given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation; 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 presented above, here is an example of a typical group creation:

```
  // A group of type "A" and its 2 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});
```

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 do allow and handle *polymorphic* groups. For example, an object of class B (B extending A) can be included to a group of type A. However based on Java typing, only the methods defined in the class A can be invoked on the group.

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

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

Such a call is asynchronously propagated to all members of the group using multithreading. Like in the *ProActive* basic model, a method call on a group is non-blocking and provides a transparent future object to collect the results. A method call on a group yields a method call on each of the group members. If a member is a *ProActive* active object, the method call will be a *ProActive* call and if the member is a standard Java object, the method call will be a standard Java method call (within the same JVM). 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 *is also a group*. The result group is transparently built at invocation time, with a future for each elementary reply. It will be dynamically

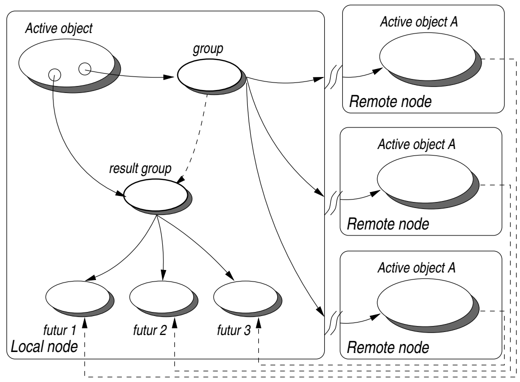

Figure 4. Method call on group

updated with the incoming results, thus gathering results, as shown in Figure 4. The *wait-by-necessity* mechanism is also valid on groups: if all replies are awaited the caller blocks, but as soon as one reply arrives in the result group the method call on this result is executed. For instance in

```
  // 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.

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), group synchronization and barriers (`waitOne`, `waitAll`, `waitAndGet`); see [1] for further details and implementation techniques.

4.2 Distribution and parallelization

The following sections explain how, using active objects, asynchronous point-to-point and group communications, the sequential version of Jem3D can be distributed on a set of machines.

4.2.1 Basic ideas and principles

Figure 5 describes the architecture of the sequential version of Jem3D: all (triangle) facets, whatever be their real type (internal or not), are grouped in an `ArrayList` of facets; all (tetrahedron) control volumes are grouped into an `ArrayList`. As each internal facet belongs to two control

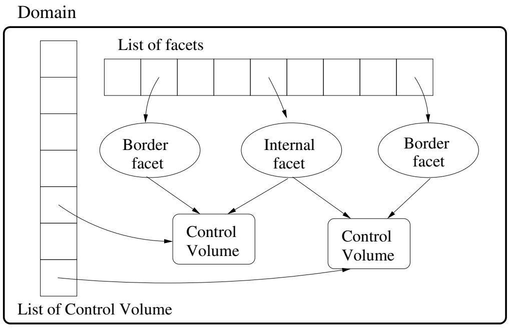

Figure 5. Architecture of the sequential version of Jem3D

volumes (CV), one can for instance see in Figure 5 the corresponding two references (from a face to 2 CVs).

After the initialization phase, the main loop repetitively executes the three phases presented in Figure 3, by going over the `ArrayList` of facets. The three phases read or update some values (i.e. the *X,Y,Z* coordinates of the electric and magnetic fiels) of the corresponding CV(s) (Figure 3).

The partitioning follows a standard decomposition of the entire domain into a set of geometric sub-domains. As we will see, the object-oriented approach brings a specific advantage: sequential references to some data-structures (e.g. facets, CVs) can be turned into remote references in a transparent manner for the code using them.

The partitioning first occurs on facets: each one is assigned to a unique sub-domain. As a consequence, some CVs will be shared by 2 sub-domains (or sometimes more); a shared CV is referenced by facets belonging to different sub-domains. Of course, specific programming techniques have to be used in order to read and update shared CVs.

4.2.2 Partitioning, local and remote objects

Figure 6 shows the architecture for the distributed version of Jem3D. The underlying idea for the parallelization is to apply a standard and natural geometric decomposition of the 3D computational domain into sub-domains. As such, some facets will contribute to control volumes that may be located onto neighbor sub-domains.

We introduce the `VirtualBoderFacets` (VBF) to represent these facets that belong to two sub-domains. In a couple of neighbour sub-domains, both have a reference to a VBF designating the shared facet. Each VBF contributes to the computation. Two neighbour VBFs which are copies of the same facet must exchange and combine their values to obtain the value of the facet. For the update access, it is the sub-domain's responsibility to trigger a remote method call onto the corresponding sub-domain – implemented as an active object–, which itself sets values in the twin VBF. Eventually, the value of the facet is set in both VBFs.

Thanks to polymorphism and dynamic binding, there is no need to explicitly deal with the effective real types of facets: internal or virtual. As a result, the control volumes that reference virtual border facet, as well as the loop that uses them, can execute unchanged.

Our architecture features a totally decentralized approach. The application is fully *peer-to-peer*: each sub-domain communicates with the others without any centralized supervisor. As centralized points are usually bottlenecks due to overload problems, we achieve a better scalability.

4.2.3 Optimizations

Regarding a read access, a naive solution would have been to let each control volume independently trigger a remote method call to read (*pull*) the values of its corresponding control volume (through an access via the remote sub-domain active object). As the algorithm implemented in the sequential version loops over facets in each phase, this implies that the computation could proceed only when a given facet effectively gets the remote values, adding up RMI and network latency. As we actually know who will need a given value, the idea is to *push* it rather than pulling it, avoiding one way of the communication needed for a pull.

In order to achieve that behavior, a sub-domain maintains a link to all its neighbors with witch it shares a facet, in order to be able to push new values to the corresponding virtual border facets. The set of neighbors is stored using a group from the *ProActive* API. Such a group is directly operable with method calls: only one method call is enough to reach all members of the group. The main point is that this avoids to program a data structure that would require an iterator in order to visit each neighbor sub-domain, and as such, perform the communications sequentially.

Each virtual border facet has to receive the value of its twin. It has been again possible to take advantage of the group communication feature in order to simply program this operation. More precisely, at initialization time, each sub-domain executes the following (refer to Figure 6 for illustration):

```
   // Builds group of neighbor sub-domains
SubDomain neighbors =
   ProActiveGroup.newGroup("SubDomain",
                           {sd1,sd2,...});

   // For each sub-domain j, builds up a
   // VBFFieldExchange collecting all
   // references to virtual border facets
   // that are shared by the current
   // sub-domain with sub-domain j
VBFFieldExchange VBFValues_j = ...
```

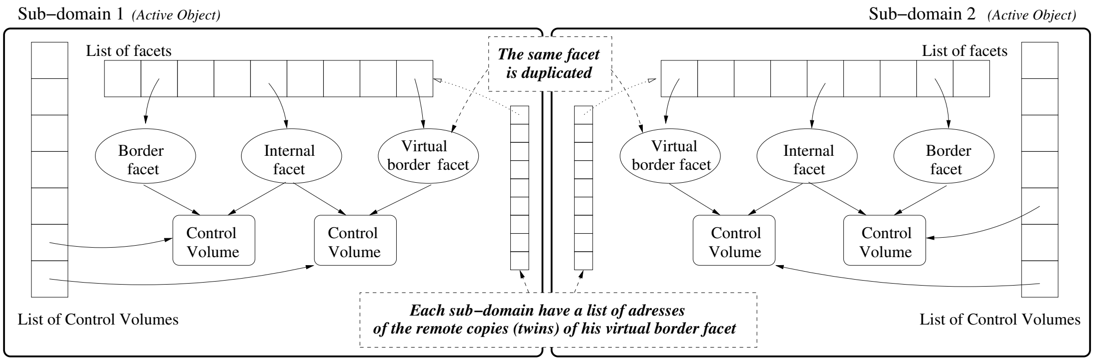

Figure 6. Architecture of the distributed version of Jem3D

```
    // A group of VBFFieldExchanges to be
    // scattered to shared VBFs
VBFFieldExchange exchangeValues =
        ProActiveGroup.newGroup(
                      "VBFFieldExchange",
                      {...,VBFValues_j,...});
ProActiveGroup.setScatterGroup(
                              exchangeValues);
```

`SubDomain` is the class name of the sub-domain active object. The variable `neighbors` stands for the *neighbor sub-domains group*, while `exchangeValues` for the *neighbor-shared VBF values group*. The `setScatterGroup` is a *ProActive* feature allowing to specify that a group parameter, subsequently used in a group communication, has to be isomorphically dispatched to the members of the group: the i^{th} element of group parameter goes (as the remote method call parameter) to the i^{th} target object in the group.

Then, at the beginning of each phase of the main loop after an update of the VBFs, the following simple instruction is executed in order *to push* appropriate values on each remote and shared VBF:

```
neighbors.push(exchangeValues);
```

This means that on each sub-domain *j* referenced in the group `neighbors`, it calls the method `push` taking as parameter the corresponding `VBFFieldExchange` referencing VBFs that are shared with sub-domain *j*. Method `push` will set values of each VBF in the corresponding remote VBF on sub-domain *j*. Subsequently, those values will be available locally by the control volumes when needed.

5 Benchmarking

To do measurement up to 32 processors, the benchmarks use a cluster of 16 Intel Pentium IV bi-Xeon @ 2Ghz 1 GB (RDRAM) - 512 Kb L2 cache, Linux RedHat 2.4.17, interconnected with a 1.5 Gbit/s Ethernet. In order to measure performances on 64 processors, we add a second cluster of 16 bi-Pentium III @ 933 Mhz 512MB (SDRAM) - 256 Kb L2 cache, Linux RedHat 2.4.17, interconnected with a 100 Mb/s Ethernet. Each computer belonging to the second cluster communicates with computers in the first cluster through a 100 Mb/s Ethernet link. We use the Sun Java Virtual Machine 1.4.0.

A bench aims at computing the time evolution of the eigenmode (1,1,1) in a cubic metallic cavity. Reported results are the total execution time for 100 time steps. To give an idea of the data involved, a mesh size of e.g. 81x81x81 represents 521,441 vertices, 3,072,000 tetrahedra, 6,220,800 faces. All of them are represented at runtime by objects in the Java version.

First, let us compare the sequential Fortran version, with the Java version. On each of the configurations we have tested (different numbers of processors, different amount of data) we have noticed an average ratio of execution time for a loop ranging from 3.5 to 3.7.

We have benchmarked the Java parallel version. Results are reported in Figure 7. Both graphics present the average duration in seconds of several executions of the benchmark (benchmark which loops in the main loop for 100 time steps). The upper one plots experiment results when running on the 16 most powerful computers (cluster 1), whereas the lower one is when running on all the computers (cluster 1 and 2). Due to the synchronization step at the the end of each loop, the measured time is the slowest computer (i.e. the longer time). Moreover, the duration highly decreases up to 8 processors for small data sizes (less than 55x55x55); then adding ressources becomes rather ineffective: from 8 to 18 processors speedup only improves from 5 to 7 on a 55x55x55 mesh. For larger size problems, starting as low as 81x81x81, the parallelization is usefull all the way up to 64 processors.

Figure 8 presents the time speedup of hundred iterations of the main loop, depending on the number of processors involved in the computation. The upper graphic shows mea-

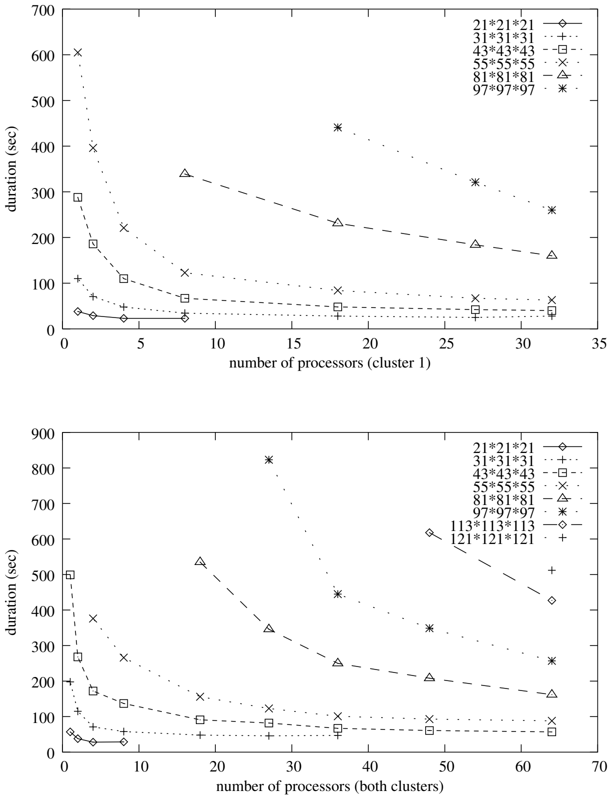

Figure 7. Average duration of 100 iterations

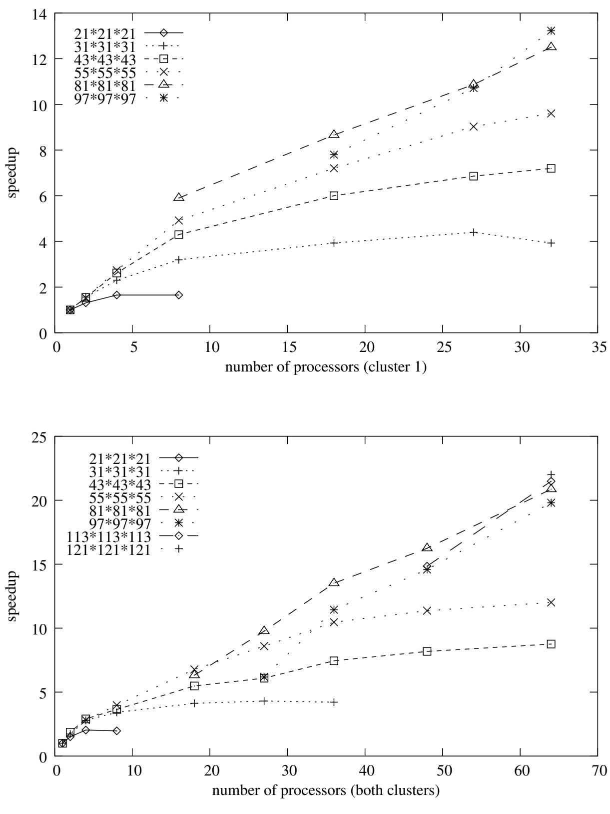

Figure 8. Speedup

surements using the most powerful cluster, and the lower one using both clusters. For some of the problem sizes, the reference execution time (i.e. the sequential execution) is *extrapolated*, because these problem sizes were so huge that the problem could not been solved sequentially on one processor; in practice, the experimentations lead to an out-of-memory error. The extrapolation is computed in the following way: first, we calculate the time spent in the treatment of a control volume, which is to be considered as the elementary data in the application, by dividing the total execution time of the benchmark by the number of control volumes involved. Secondly, we estimate the execution time for large size problems by multiplying the elementary time by the number of control volumes in the problem. The curves expose an efficiency in the range of 30% to 35% for the larger problems using all available processors. Reducing the number of processors, the efficiency steadily increases up to 75% on two processors for all cases.

6 Conclusion

The Java version of EM3D has great potential for extension and adaptability: going from element to volume methods, using structured, unstructured, or hybrid meshes. At the same time, the performance penalty for such an abstract architecture implemented in Java seems reasonable: in a factor of 3 to 4 compared to the Fortran version. This is a good result according to [9] that shows Java applications is a factor 3.3 to 12.4 slower than the corresponding Fortran operations. Moreover, this Java version is rather recently compared to the Fortran one, and there is still a lot of room for code optimization. Using a high-level library (*ProActive*) the parallel version was easy to obtain, maintaining the structuring of the sequential one. As a consequence, if an evolution of the sequential version occurs, the parallel one should remain, and evolve automatically.

Getting at performance figure, first it is important to note that the parallel object-oriented approach, using fully standard and portable elements of the Java platform, is already effective on the problem size. The parallel version allowed us on the clusters to get results with data size significantly larger than the sequential version (121x121x121 versus 43x43x43). The former number is to be compared to the largest size the Fortran version can currently execute: 161x81x81 (which is equivalent to 101x101x101 mesh). Even if this is due in the current Fortran program to a problem of static array allocation which could be improved with some restructuring, it probably tells something about the flexibility of a more dynamic approach.

With respect to speed up, we managed to reach values in the range of 22 on 64 processors. With respect to efficiency, the best values are of course on smaller configurations: in the range of 50 % for about 10 processors, 75%

on 2 processors. Those performance figures, although already very good in an object-oriented world, leave room for improvement. Analysis of the executions confirms that progress should come from two yet-to-be-improved pieces of the current platform: serialization and standard RMI. As it is well known [10, 21], the standard Java RMI mechanism is rather slow for cluster computing. So, in order to reach better scalability using the parallel version, the first step will be to use fast implementations of the transport layer. Several techniques have been proposed in the past, for instance [11, 20, 19], we are planning to experiment with such solutions in future work.

Finally, in order to experiment with very large data set and large number of processors, we run benchmarks on the INRIA production network, such configuration being sometimes called *P2P Intranet on desktop machines*. It made it possible to solve a 150 to the cube mesh, that is to say over 100 million facets, using 252 processors at the same time, in 1600 seconds. Future work will also include further testing and benchmarking of this setting.

References

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

[2] 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*, pages 93–102, 2002.

[3] G. Berti. *Generic software components for scientific computing*. PhD thesis, TU Cottbus, 2000.

[4] D. Brown, W. Henshaw, and D. Quinlan. Overture : object-oriented tools for overset grid applications. *AIAA paper No. 99-9130*, 1999.

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

[6] B. Carpenter, G. Fox, S. Ko, and S.Lim. mpiJava 1.2: API Specification. `http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html`, October 1999.

[7] A. Denis, C. Pérez, T. Priol, and A. Ribes. Padico: A component-based software infrastructure for grid computing. In *17th International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, Apr. 2003. IEEE Computer Society.

[8] R. Falgout and U. Yang. Hypre: a Library of High Performance Preconditioners. *Lecture Notes in Computer Science*, 2331:632–641, 2002.

[9] M. A. Frumkin, M. Schultz, H. Jin, and J. Yan. Performance and Scalability of the NAS Parallel Benchmarks in Java. In *JAVAPDC, workshop on Java for Parallel and Distributed Computing at IPDPS*, 2003.

[10] V. Getov, G. V. Laszewski, M. Philippsen, and I. Foster. Multiparadigm communications in java for grid computing. *Communications of the ACM*, 44(10):118–125, 2001.

[11] V. Getov and M. Philippsen. Java Communications for Large-Scale Parallel Computing. *3rd International Conference on Large-Scale Scientific Computation. Lecture Notes in Computer Science*, 2179:33–45, 2001. Invited paper.

[12] D. Hanel, A. Dervieux, O. Gloth, L. Fournier, S. Lanteri, and R. Vilsmeier. Development of Navier-Stokes solvers on hybrid grids. *Notes on Numerical Fluid Mechanics*, 75:49–66, 2001.

[13] B. Heismund. JMP users guide, Department of Mathematics, University of Bergen. `http://www.mi.uib.no/~bjornoh/jmp/0.7.2/jmp.pdf`, 2003.

[14] B. Henz and D. R. Shires. An Object-Oriented Programming Framework for Parallel Finite Element Analysis with Application: Liquid Composite Molding. In *PDSECA, workshop on Parallel and Distributed Scientific and Engineering Computing with Applications at IPDPS*, 2003.

[15] G. Karypis and V. Kumar. Parallel Multilevel k-way Partition Scheme for Irregular Graphs. *SIAM Review*, 41(2):278–300, 1999.

[16] S. Lanteri. Parallel Solutions of Compressible Flows Using Overlapping and Non-Overlapping Mesh Partitioning Strategies. *Parallel Comput.*, 22:943–968, 1996.

[17] M. Li, O. Rana, and D. Walker. Wrapping MPI-based Legacy Codes as Java/CORBA components. *Future Generation Computer Systems*, October 2001.

[18] G. K. M. Joshi and V. Kumar. *PSPASES: scalable parallel direct solver library for sparse symmetric positive definite linear systems*. Department of Computer Science, University of Minesota, 1999.

[19] J. Maassen, T. Kielmann, and H. Bal. Efficient replicated method invocation in java. In *ACM 2000 Java Grande Conference*, pages 88–96, 2000.

[20] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, 2001.

[21] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[22] S. Piperno, M. Remaki, and L. Fezoui. A Nondiffusive Finite Volume Scheme for the Three-Dimensional Maxwell's Equations on Unstructured Meshes. *SIAM J. Numeri. Anal.*, 39(6):2089–2108, 2002.

[23] L. M. S. Balay, W. Gropp and B. Smith. *PETSc users manual*. Argonne National Laboratory, 2001.

[24] M. Shields, O. Rana, and D. Walker. A Collaborative Code Development Environment for Computational Electro-Magnetics. In *IFIP TC2/WG2.5 Working Conference on the Architecture of Scientific Software*, pages 119–141. Kluwer Academic Publishers, 2001.

[25] B. C. and al. MPI for Java - Position Document and Draft API Specification. `http://www.npac.syr.edu/projects/pcrc/reports/MPIposition/position.ps`, 1998. citeseer.nj.nec.com/carpenter98mpi.html.