Int. Conf. on Computational Methods for Coupled Problems in Science and Engineering
COUPLED PROBLEMS 2005
M. Papadrakakis, E. Oñate and B. Schrefler (Eds)

GRID COMPONENT TECHNIQUES: COMPOSING, GATHERING, AND SCATTERING

CAN IT BE USED FOR COUPLED PROBLEMS ?

Laurent Baduel*, Francoise Baude*, Denis Caromel*, Ludovic Henrio*,+
Fabrice Huet*, Stephane Lanteri*, Matthieu Morel*

*INRIA, I3S-CNRS, 2004 Rt. des Lucioles, BP 93, F-06902 Sophia Antipolis Cedex
e-mail: First.Last@inria.fr - Web page: http://www.inria.fr/oasis
+Harrow School of Computer Science - Univ. of Westminster - Harrow HA1 3TP, U.K.

Key words: Grid Computing, Components, ProActive Open Source Middleware

Abstract. *Grids raise new challenges for the programming, the composition, and the deployment of numerical applications. Heterogeneity, medium to high latency, various underlying systems and protocols call for new paradigm and techniques. Within this framework, the development of numerical methods, must integrate those new factors, representing both new difficulties and new opportunities.*

In this article, we describe an Open Source middleware for the Grid, ProActive, part of the ObjectWeb Consortium, featuring distributed objects and components. Using ProActive, we demonstrate how to design and program a 3D electromagnetic application: an object-oriented time domain finite volume solver for 3D Maxwell equations. The benchmarks show very good performances, on both single clusters and Grids. For instance a speedup of 100 on 150 machines on a Grid. Moreover, we were also capable of executing large domains (more than 100 million facets) on up to 300 processors.

Finally we move to features being developed that could be an asset for programming Coupled Problems. Components can feature interfaces for automatically scattering, gathering, redistributing data upon communications. One purpose of this paper is to present techniques that could be of interest for coupled problems, and get feedbacks and requirements from people working on coupled problems.

1 INTRODUCTION

Component programming for Grid and peer-to-peer computing is gaining growing interest, as it is considered of being capable to tackle the complexity and heterogeneity of target applications. Furthermore, the support and evolution of such applications might be eased by the component techniques. Also, components composition appears to be an appropriate mean to couple codes that individually solve a specific problem in order to collectively solve a new problem (e.g. obtain multi-component integrated solutions in

turbulence simulations [27], build distributed cyber infrastructure to enable "better than real-time" prediction of mesocale weather events [19]).

Examples of popular programming models for grid computing currently in use include MPI for message passing, and GridRPC [24] for remote procedure calls. But, an alternative, to which we adhere is to consider Grid programming as requiring a two-level programming approach [15]: nuggets or code modules are generated by conventional programming, that must be augmented for the Grid by the integration of the distributed nuggets together into a complete executable. Of course, each nugget may be something as complex as a parallel and distributed application in itself. The user can be offered many different paradigms for expressing this integration. One common model is a graphical interface where nuggets are chosen from a palette and linked via their ports or channels. A perhaps more powerful way is to program the linkage, via scripting or compiled programming languages. Finally, we rely on a framework so as to instantiate those components and to allow them to be composed into applications.

Examples of such two-level approaches include the CCA model [18] which defines components, and where instantiation and composition are implemented within a framework (for instance, XCAT [7, 22]); the ICENI project [17]; the GridCCM project [13] which relies on the CORBA Component Model for the component definition and whose specificity is to efficiently embed parallel MPI codes.

In this paper, we present ProActive[1], a solution for programming, composing, and deploying on the Grid. The paradigm is expected to be suitable for coupled problems, and to solve them efficiently on the grid. Part of the originality is comprehensiveness, in the sense that ProActive provides:

- a *programming model* for the nuggets which features, among others, object orientation for code factorization and reuse [10] , OO SPMD [3], asynchronicity in remote method invocation for latency hiding [1],
- a *composition model* [5] to build applications by composing such nuggets, which features a hierarchical and dynamic composition model of inner components, in order to tackle the software architecture complexity while easing its dynamic evolution due to changing runtime conditions or software modification; server and client interfaces, that can be single or collective, for expressing possibly sophisticated component interactions while implementing them efficiently (e.g. in an asynchronous manner),
- various *deployment tools* [4] in order to run the resulting composed applications on a wide range of computing platforms including grids, without any dependency in the source code, so without any impact on it may the target platforms change.

Overall, ProActive complete integrated grid computing solution should alleviate the programmer's burden.

[1]ProActive is an Open Source middleware developed within the international consortium ObjectWeb. Freely available at http://ProActive.ObjectWeb.org

The paper begins with a general description of the ProActive solution. In Section 3, we move on the description of a numerical application, Jem3D, that solves an electromagnetism problem in parallel, using the proposed object-oriented distributed approach. As this application is very demanding regarding parallelism, distribution and induced communications, the fact that good performances are obtained, even on a truly distributed Grid, is a very positive point (see Section 4). The proposed component-oriented model is then described in Section 5. The overall goal of defining this model is to be able to build components that could be applied in the domain of numerical problem solving such as Jem3D. Then, the aim is to be able to couple such components to other numerical or visualization applications also available in the form of components, effectively yielding to efficient coupled problems solving.

2 PROACTIVE

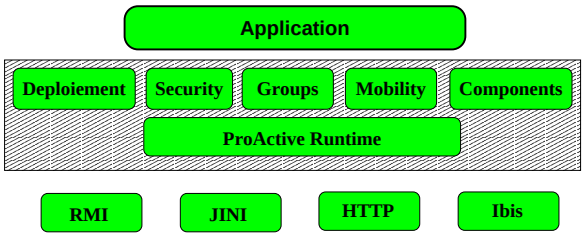

Figure 1: Application running on top of ProActive

ProActive is a Java middleware for parallel, distributed and concurrent programming which features high level services like weak migration, group communication, security, deployment and components. The current implementation can use 4 different communication and registry/discovery layers: RMI, Jini (for environment discovery), a HTTP based protocol and Ibis [28], a high performance communication layer.

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 active object has its own thread of control, and decides in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are always asynchronous with transparent *future objects* and synchronization is handled by a mechanism known as *wait-by-necessity* [9]. All active objects are running in a JVM, and belong to a *Node* which provides an abstraction for their physical location. At any time, a JVM hosts one or several nodes.

2.2 Group communications

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

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

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

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

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

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

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

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

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

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

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

2.3 Deployment Descriptors

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

```
VirtualNode:
  jem3DNode

Mapping:
  jem3DNode --> VM1 , VM2

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

Figure 2: A simple deployment descriptor

An example of deployment file is given in Figure 2. For the sake of clarity, we have used a pseudo-code syntax instead of the less readable XML one. We have indicated in *italic* the symbolic names which are used as references in the file. These names are used to structure the descriptor and can be of arbitrary value. In **bold** references are the actual classes provided by ProActive. The application which will use this file will be able to use the symbolic name *jem3DNode* in the source code to access these resources. When used, this virtual node will be mapped onto two virtual machines, *VM1* and *VM2*, specified in the infrastructure part. The creation of these virtual machines is as follows. The first one will be created locally. The second one will trigger a ssh connection to host1 and then perform the creation of a new local virtual machine there. In this part it is possible to specify various environment variables such as CLASSPATH to be used for the creation of the virtual machine.

3 A 3D COMPUTATIONAL ELECTROMAGNETISM APPLICATION

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

3.1 Overview

Jem3D numerically solves the 3D Maxwell equations modeling time domain electromagnetic wave propagation phenomena. It relies on a finite volume approximation method designed to deal with unstructured tetrahedral discretization of the computation domain (see [25] for more details).

A standard test case for which an exact solution of the Maxwell equations exists (therefore allowing a precise validation of Jem3D with regards to both the numerical kernels and the parallelization aspects) consists in the simulation of the propagation of an eigenmode in a cubic metallic cavity. For this test case, the underlying tetrahedral mesh is automatically built in a pre-processing phase of a typical run of Jem3D. This mesh is simply obtained by first defining a Cartesian grid discretization of the cube and then, dividing each element of this grid in six tetrahedra. Ongoing Jem3D developments aim at handling general (irregular) tetrahedral meshes and complex geometries. In the sequential version of the application, the cube is divided into tetrahedra where local calculation of the electromagnetic fields is performed. More precisely, the balance flux is evaluated as the combination of the elementary fluxes computed through all 4 facets of the tetrahedron. After each step, the local values calculated in a tetrahedron are passed to its neighbors and a new local calculation starts. The general algorithm is given in Figure 3 and displays two phases in the running of the application: the *initialization* and the *calculation*. The complexity of the calculation can be changed by modifying the number of tetrahedra in the cube, which is done through the mesh size.

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

As a consequence, a greater mesh size requires a greater number of tetrahedra, more memory, and a more intensive computation.

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

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

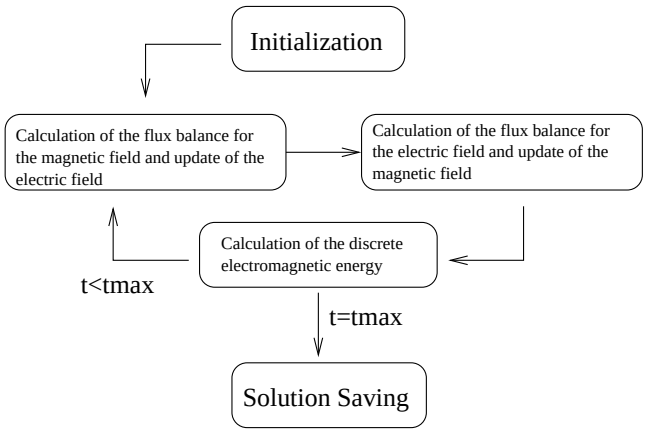

Figure 3: Simplified view of the algorithm used in the jem3D application

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

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

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

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

3.2 Architecture of the application

Jem3D is written completely in Java on top of ProActive and does not use either third-party or native libraries to perform the calculation. It can run on any standard Java platform where ProActive is available. Subdomains are *active objects* and communication between them is done through group communication mechanisms. Each communication between subdomains consists of a linked list whose elements, one for each border face,

contain one array of 3 doubles. A special object, called the *collector* ensures the initial synchronization of the subdomains and can perform monitoring and steering of the application, if needed.

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

4 EXPERIMENTS

We will, in this section, study the time taken by our application to perform 100 loops of the *calculation* (see Figure 3) using different mesh size and number of nodes. We will compare the results obtained when using RMI and Ibis. As a reference, we will use a Fortran/MPI version of the same algorithm.

4.1 Experimental test bed

We have conducted our experiments on the Distributed ASCI Supercomputer 2 (DAS-2[2]). The nodes are composed of Dual Pentium III CPU running at 1Ghz with 1GB or more of memory, running RedHat Linux and linked with fast-ethernet. Two remote sites of the DAS-2 are connected through 10Gbits/s connections and the latency between them varies from $1ms$ to $3ms$. We have used the Sun JDK 1.4.2 for all our experiments.

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

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

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

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

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

4.2 Comparison with the Fortran/MPI version

The original algorithm used in Jem3D was actually exploited first in a Fortran/MPI version called EM3D. We wanted to perform a comparison in order to provide us with some insight on the relative performance of these two versions. The aim is not to compare

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

Java with Fortran but rather two implementations of the same algorithm using different architectures and see whether the distribution of the calculation can be as efficient in Java as in Fortran/MPI.

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

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

Table 1: Java vs Fortran on 1 node

our comparison will be the execution time and memory usage of the sequential versions of the algorithm. Since we couldn't modify the Fortran version, the memory, for both applications was measured using *top* under Linux and the results are shown in Table 1. The ratio between the execution time of the Java and Fortran versions is around 2.38 which we believe is a good, taking into account the differences existing between the two versions and previously published benchmarks [20] that reported a ratio between 1.24 and 2.70 for some applications. The memory usage ratio is around 1.33 which, while still being acceptable, can probably be improved since it has not been, to the best of our knowledge, a goal in the design of Jem3D. We have performed some simple optimizations on the Java code, like creating dynamic data structures with the final size instead of having them grow when needed, thus avoiding the creation of temporary variables and overprovisionned structures.

Considering now the distributed version, whose results can be seen in Figure 4, the Java version is, to no surprise, still slower than the Fortran one. However, where the ratio of execution time (Java/Fortran) reached with RMI was almost all the time around 3.5, we achieved a much better performance with Ibis, lowering it to around 2.5, very close to the sequential one. The speedup achieved is, in both case, lower than the Fortran one, however, as we increase the mesh size, Ibis scales better, increasing its speedup to as much as 12 on 16 nodes, compared to 8.8 with RMI and 13.8 with Fortran.

The Sun implementation of RMI seems to be a limiting factor for the performance of distributed applications. Using a different communication layer, it is possible to increase

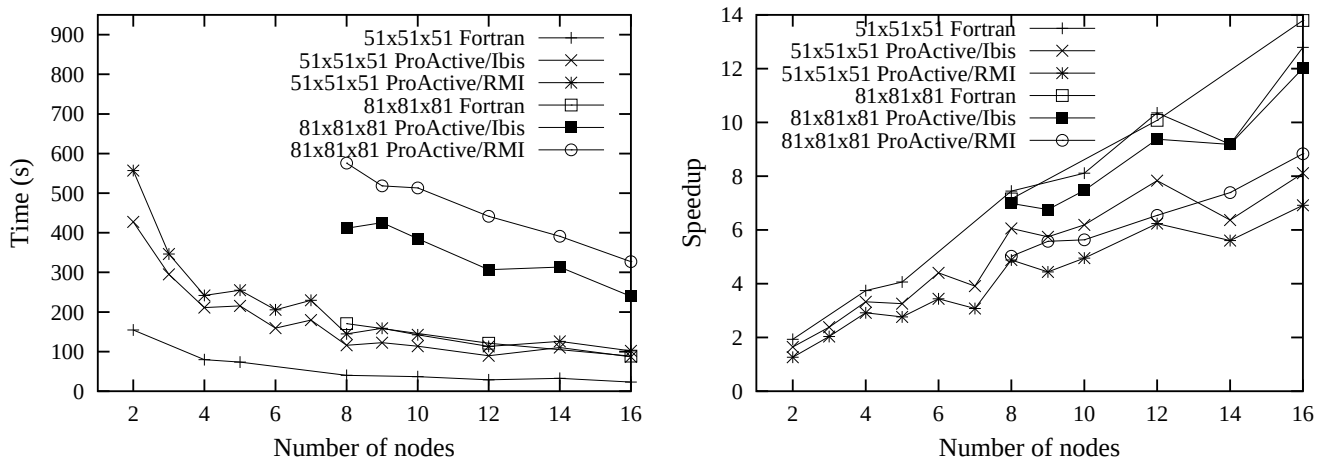

Figure 4: Execution time and speedup for the Java and Fortran versions

the performance and to achieve very good speedups, very close to those obtained using Fortran/MPI.

4.3 Desktop experiments

Figure 5 reports experiments using a large number of machines, up to 300 processors. The configuration used is the so called *desktop intranet peer-to-peer*: all available desktop machines within the INRIA research laboratory where used to run large electromagnetic domains (up to a mesh size of 151x151x151, i.e., 100 million facets). The rather irregular nature of curves is explained by two factors. First, we were using the fastest machines as much as possible, leading to poorer efficiencies as we reached large configuration. Second, since machines were not dedicated, users unpredictable behavior impacts the curves. Indeed, this experiment is not after regular speedup, but rather to demonstrate the capacity to deal with large domains using large number of non-dedicated machines.

4.4 Multi-cluster experiments

Following [14] stating that a Grid is a system that coordinates resources without centralized control, using standard protocols and interfaces to achieve a non trivial quality of service, we experimented on the DAS-2 Grid. Taking full advantage of it, we were able to perform multi-cluster experiments by requesting nodes on each of its parts. As an example, the distribution for an experiment using 8 subdomains distributed on 4 clusters (fs0, fs1, fs2, fs3) is shown in Figure 6. Each subdomain has 3 neighbors with which it exchanges locally calculated values. They all report to the Collector but, for the sake of clarity, we did not indicate all the communications in the figure. In the current version we do not take advantage of the network architecture and it might happen that the repar-

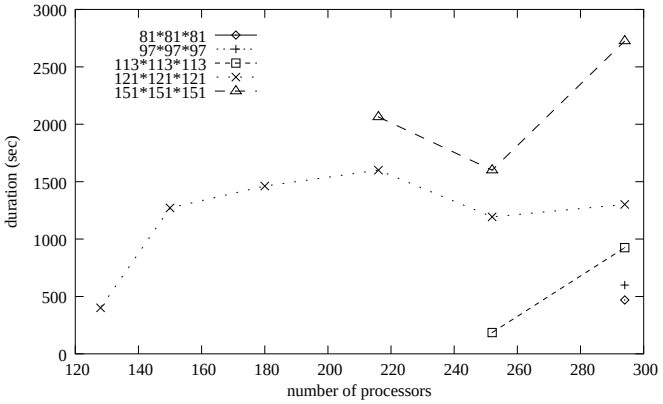

Figure 5: Desktop benchmarks on up to 300 machines, 100 million facets

tition of the calculation is sub-optimal, having multiple communications over slow-links. However we did not find any conclusive proof in our experiments that such a situation occurred. Indeed, the application demonstrated a very good speedup, even when distributed over multiple clusters.

In order to have the application run, we had to perform only the following two steps: (*i*) install the needed classes on each file system [3], (*ii*) write a descriptor file which describes the architecture used for the experiments. The descriptor files basically states that: Jem3D will run on 5 cluster, 1 local (fs0), 4 remotes (fs1, fs2, fs3 and fs4). The deployment process will start from fs0. There, a PBS command will be issued to book some nodes. To request nodes on the other clusters, first an ssh connection will be established with each one of them and then, a PBS command will be issued. Once the nodes granted by the scheduler, JVMs will be created as specified in the *descriptor file*, along with a ProActive Runtime which will allow the application to deploy itself. All these operations are usually handled in scripts or various programs and often requires some human intervention. Using the deployment mechanism, this can be done automatically from *inside* the application which, as a side effect, provides a very useful feature. Since the application can check, using some ProActive API, the number of nodes which have been allocated, it can adapt its execution to the available resources. As an example, if Jem3D requests 20 nodes on each of the four clusters but get only 60 out of 80 because of co-allocation issues, it could either reduce the size of its calculation or postpone it until enough resources are available, making it self-adaptive to its environment.

The results of this experiments are shown in Figure 7. Running on a single cluster, the maximum efficiency reached was 96% on 2 nodes with a mesh of $61 \times 61 \times 61$ whereas, on multiple clusters, we had 85.4% on 80 nodes with $201 \times 201 \times 201$, which is very good considering that we did not have any control on the distribution of the elements of the

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

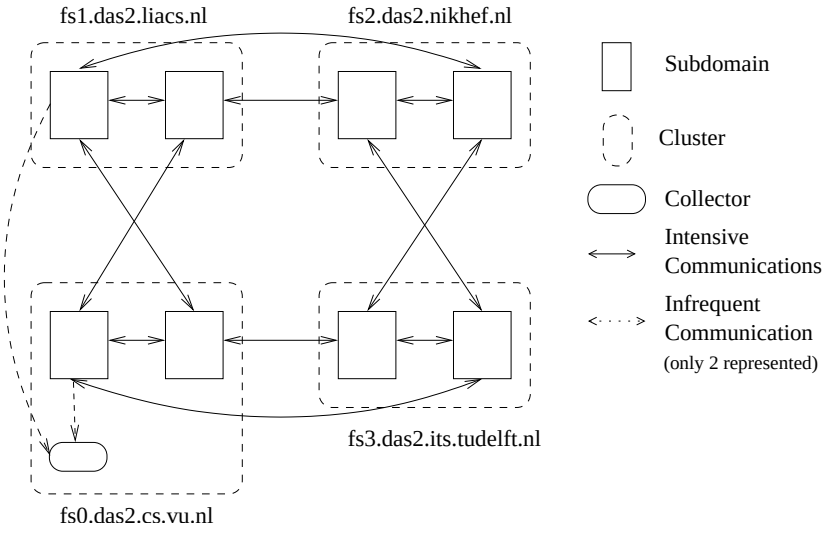

Figure 6: Distribution of the calculation on the DAS-2 Grid with 8 subdomains

calculation.

5 COMPONENTS

Component based programming is gaining growing interest, as mentioned in the introduction of this paper. One important remark is that, to our knowledge, other component models for Grid programming enable an assembly of components which is only **flat**. We propose a novel approach through which grid applications will be built by assembling components in a **hierarchical** way instead. This assemblage may dynamically evolve through several **reconfigurations**.

Our claim regarding component-oriented Grid programming is as follows:

1. a set of components, assembled or not, may usefully yield to a new composite component that can recursively be composed with other components. This is a way to enforce code reuse and scalability of the composition task, because it structures this task. More precisely, it enables the user which builds an application by composition, to naturally proceeds in a hierarchical and structured manner. With such an approach, we aim at easing the programming, the deployment, and eventually the monitoring of complex Grid applications;

2. inclusion, bindings, and also location of components, must be reconfigurable. This is a way to adapt to the dynamicity of Grid runtime environments. For instance, if a component fails, it should be possible to replace it by a new instance, then to rebind the enclosing component to this new enclosed component, without any other consequence for the rest of the application.

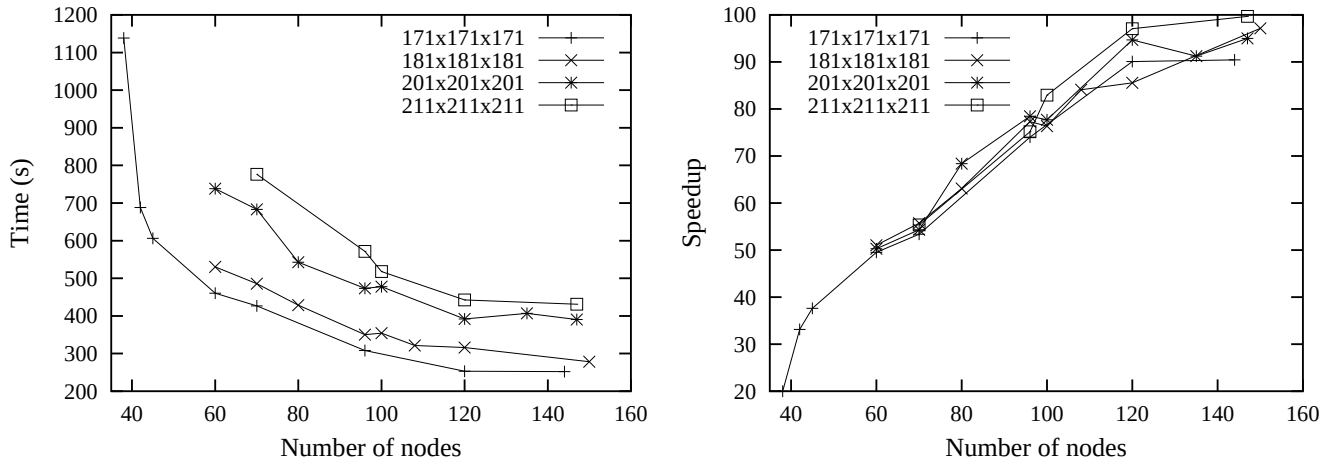

Figure 7: Benchmarks with all available nodes on the DAS-2 Grid

As a concrete illustration of those ideas, we describe the first version of a component model for Grid computing we have defined and implemented within the ProActive middleware [5]. The specification of the components is conformable to the Fractal component model [8, 16], a generic and extensible software composition framework. We provide an implementation of the Fractal specification API within the ProActive library [26].

We first give a brief but complete overview of the proposed component model, while in the second section, we expose a mean for achieving parallelism and coupling of parallel components.

5.1 Overview of ProActive components

By implementing the Fractal component model within the ProActive library, components at the **primitive** level (8.a), are themselves formed of one or several active objects (i.e. a primitive component is a nugget that may be parallel and distributed). Standard meta-information (e.g. XML) technique has to be used for identifying provided and used ports or interfaces. Those ports are typed, and no IDL is required as the components are all defined using the Java language. Currently, we are working on the design of a generic wrapper implemented as active objects, whose aim is to encapsulate legacy parallel codes (i.e. Fortran-MPI, C-MPI codes). But the Grid specificity calls for specific information related to the parallel and distributed nature of codes. First, an abstraction for the mapping of such codes must be included in the component meta-information for the deployment. Second, the client and server interface specifications must authorize collective behavior. Other properties such as quality of service requirements for instance may also be identified in the future.

Primitive components being defined, the next step to master complexity and scale of Grid applications is to be able to compose those building blocks into new components

called **composite components** (8.b). The resulting encapsulated composite components can be seen as functional and autonomous subsystems. Recursively, a composite component can be defined as the composition of primitive or composite components.

A specialization of composite components is **parallel** components (8.c): all the inner components of the composite component are of the same component type; invoking a provided interface on the parallel component triggers its parallel propagation to all the inner components.

These three kinds of components are the building blocks of components systems, which can be initially described in a declarative manner using an architecture description language (ADL) [11]. The ADL specifies, in a standardized XML format, components definitions, assemblies and bindings, and distributed deployment information. Distributed deployment information currently consists of affecting virtual nodes (see 2.3) to components. It is the component framework, responsible for deploying the application on the target architecture, that maps virtual nodes names to concrete physical locations.

A component is deployed and run through the container that ProActive transparently offers via a set of meta-level objects. In particular, meta-level objects implement all the component controllers specified by the Fractal model. These controllers handle non-functional properties such as *bindings*, for linking components, *content*, for assembling hierarchies of components, or *life-cycle*, for starting and stopping components.

It is possible to dynamically invoke those controllers: this allows dynamic modifications of the initial description of a component system, incorporation of newly created or discovered components, reconfiguration of assemblies, etc.

5.2 Example

As an illustrative purpose we sketch a component-oriented vision of Jem3D. An example of a primitive component in Jem3D would be, for instance, the collector (see section 3.1 and figure 5). One of its server ports could serve for subdomains to report back information to it. Symmetrically, one of its client ports could serve for pushing parameters for the numerical computation to the subdomains. One or several interfaces of the connector could be published (as server or client) in order to connect it with an external configuration and visualization tool of the computed collected information. One primitive component could encapsulate all the interconnected subdomains and be bound to the collector for reporting back information. Overall, the whole Jem3D application would be formed of one composite component, gathering the collector and the subdomains. The interfaces of the collector pertaining to configuration and visualization would be the only ones visible at the level of the composite. May a different implementation for parallel computation on subdomains be used, only the inner corresponding component implementation should be modified, without impact to the way visualization and steering tools are bound to Jem3D.

Starting from an object oriented version, designing a component-oriented one mainly pertains picking which interfaces are published (in a sense, on which contracts components do adhere, whatever their implementation could be).

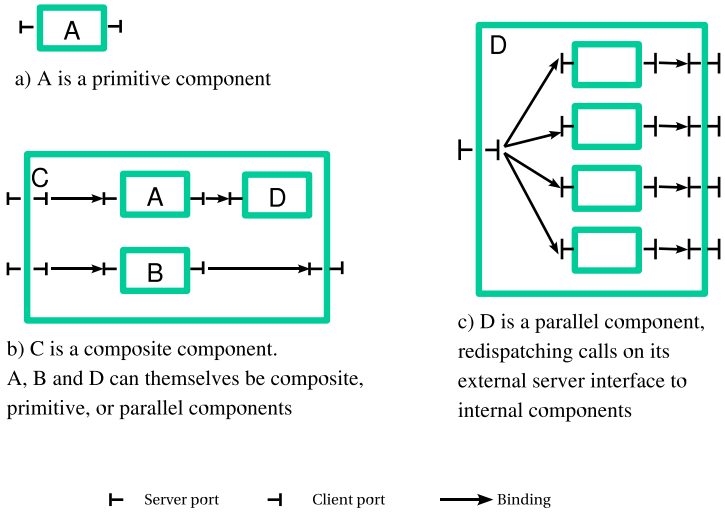

Figure 8: The 3 types of components

5.3 Collective ports

As the proposed model targets high-performance Grid computing, there is an additional need compared to the original Fractal component model: parallelism. This means first, expressing parallelism at the component level and second, be able to implement it efficiently. For the first point, we introduce the notion of a parallel component which automatically offers collective server ports (see figure 8.c). We also need to offer the notion of collective client port for a parallel component. This seems to be useful to couple parallel components (see section 5.3.2). For the second point, we rely on the usage of groups as provided in ProActive (see section 2.2). We will further detail both points in the sequel of collective-ports.

5.3.1 Definition

The notion of a **collective port** corresponds to the grouping of ports of the same type, with the following behavior: a service invocation on the port must be propagated as parallel as possible to the bound components, so that the services really get a chance to run in parallel. In the Fractal general model, a collective interface represents a set of interfaces of the same type, with a common naming prefix. What we add here is the possibility of performing parallel invocations on this set of interfaces.

For this, we rely on the group communication mechanism of ProActive which achieves asynchronous remote method invocations for a group of remote active objects of the same type, with automatic gathering of replies (see section 2.2). More precisely, component interfaces bound to the collective port are added as members of a group of component

interfaces, and the binding is effective by storing a proxy to this group. The effect is that any call through this proxy gets broadcasted to all members.

Parameters of each individual call are the same, i.e. they are broadcasted, as it is the default way of passing parameters in the group communication mechanism defined in ProActive. But, as in ProActive group communications, it is also possible to change the semantic for a parameter passing, by defining it as scattered. This requires that the parameter be itself a group of values, and then, those values are scattered among the resulting calls on the group.

The gathering of replies that come back from the collective port service invocation can be further associated with synchronizations (`waitAll`, `waitAny`, `wait(int number)`, etc) and further manipulated by programming *combination/reduction* operations.

Currently, configuring the behavior of parameter passing and reply management must be hand programmed. Our future work is to extend the ADL in such a way that those specificities get automatically programmed according to new additional attributes for the binding definitions of the component assemblies.

5.3.2 Composition of collective communications

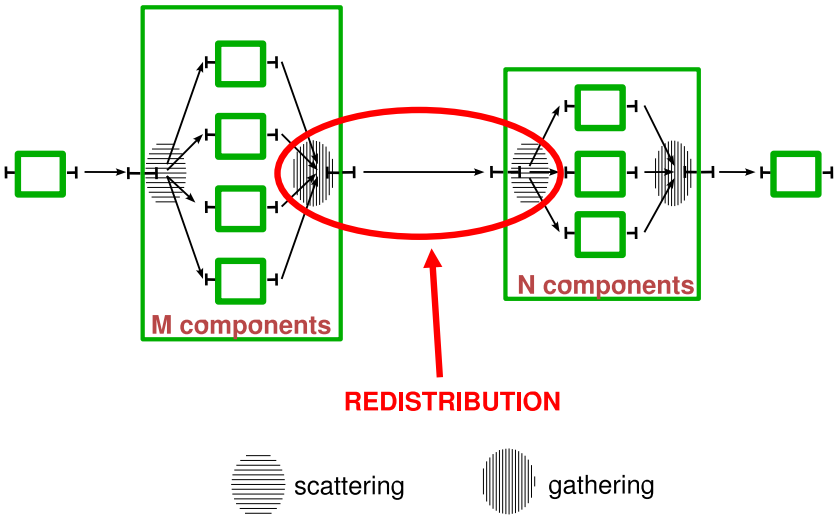

Figure 9: Redistribution from M components to N components

In order to couple parallel components, we plan to provide facilities for composing collective ports. Every parallel component has a set of meta-objects associated to it and

could serve as a sophisticated re-dispatcher: for the set formed from each client port of interest of each inner component of the parallel component (thus defining the notion of a collective client port of a parallel component), to the server collective port of interest it is bound to. The objective – but maybe not the solution – is similar to what is achieved by introducing collective communications as tees in the ICENI Grid oriented component model [23]: switch, combiner, splitter, gather, broadcast; the same regarding the collective port extending CCA ports, experimented in [21], which is in fact implemented as a combination of translation components (i.e. customizable components, efficiently called by the framework, to tackle translation/redistribution of data, collective invocation and returns, e.g. a MxN component). Our challenge is to provide a solution adapted to the component-oriented model we propose, that is, without the explicit introduction of additional components (either generic or programmer-modifiable), but only through the definition of Fractal ports and the usage of the ProActive group communication mechanism. An illustration of this objective is given in figure 9: the idea is to automate the broadcasting or scattering of invocation parameters, and symmetrically the gathering of reply parameters. In the case where the communications occur from M components to N components this also requires a redistribution policy of the parameters and replies from M invokers to N receivers. Schematically, this could end up by also synchronizing M calls and gathering their parameters, then scatter them onto N calls. Regarding the replies, the inverse operation is needed: gathering N replies and automatically scattering them onto M invokers. In this sense, our design of collective ports composition is related to the one in [13], which is based on collective RMI calls extended with the usage of a MxN redistribution scheme introduced in recent CCA compliant implementations of parallel data redistribution [12, 6]. Within the context of ProActive, asynchronous group communications with group of futures open the way to a wide range of new perspectives.

6 CONCLUSIONS

In this article, we first presented ProActive, a Java middleware for writing parallel and distributed applications. One of the objectives of the active object model implemented in ProActive is to relieve the programmer from most of communication and parallelization concerns while he is writing the application code. To fulfill this objective, all the deployment and configuration information are concentrated in deployment descriptor files.

The resulting model proved to be efficient, even compared to a Fortran/MPI implementation. Moreover, ProActive allowed us to run the Jem3D application on a Grid of 150 nodes, with a speedup of 100.

A natural extension of this work was to consider component programming. Components provide an abstraction of pieces of program easing their composition and support. Components allow to tackle the complexity of Grid applications by tackling each different concern in a different component. Moreover, they provide controllers which should reveal a valuable tool in order to tackle particular Grid problems and thus relieve the program-

mer from the scalability and heterogeneity aspects while writing his application specific code.

Concerning scalability, first the hierarchical nature of our component model provides scalability at the application programming level. This approach is even more interesting when it comes to coupled problem as components allow to couple programs solving different problems in a very natural and sound way. The user builds his application in a much hierarchical and structured manner: building components by interconnecting primitive ones, and then reusing these composite to build even bigger composites recursively. Second, parallel components provide group communications between component providing parallel invocation on a set of components which is a necessary first step for a high-performance Grid computing component platform.

Finally, an exhaustive coupling mechanism between collective ports requires to implement a gathering/scattering mechanism mechanism; introducing such a mechanism in an automatic, transparent and efficient way in our component model is a challenging perspective.

REFERENCES

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

[2] L. Baduel, F. Baude, D. Caromel, C. Delbé, S. El Kasmi, N. Gama, and S. Lanteri. A Parallel Object-Oriented Application for 3D Electromagnetism. In *18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, USA, April 2004. IEEE Computer Society.

[3] Laurent Baduel, Françoise Baude, and Denis Caromel. Object-Oriented SPMD. In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, May 2005.

[4] F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications. In *11th International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[5] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications*, Catania, Italy, November 2003. Springer Verlag, Lecture Notes in Computer Science, LNCS.

[6] F. Bertrand and R. Bramley. DCA: A Distributed CCA framework based on MPI. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments at IPDPS*, Santa Fe, New Mexico, USA, April 2004. IEEE Computer Society.

[7] R. Bramley, K. Chin, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yochuri. A Component-Based Services Architecture for Building Distributed Applications. In *9th International Symposium on High Performance Distributed Computing*, Pittsburgh, Pennsylvania, USA, August 2000. IEEE Computer Society.

[8] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *7th International Workshop on Component-Oriented Programming at ECOOP*, Malaga, Spain, June 2002.

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

[10] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. In Geoffrey C. Fox, editor, *Concurrency Practice and Experience*, volume 10, pages 1043–1061. Wiley & Sons, Ltd., September-November 1998. http://ProActive.ObjectWeb.org/.

[11] P.C. Clements. A Survey of Architecture Description Languages. In *International Workshop on Software Specification and Design*, pages 16–25, Schloss Velen, Germany, March 1996.

[12] K. Damevski and S.G. Parker. Parallel Remote Method Invocation and M-by-N Data Redistribution. In *4th Los Alamos Computer Science Institute Symposium*, October 2003.

[13] A. Denis, C. Pérez, T. Priol, and A. Ribes. Bringing high performance to the CORBA component model. In *SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, California, USA, February 2004.

[14] I. Foster. What is the Grid? A Three Point Checklist. GridToday, July 2002.

[15] G. Fox, M. Pierce, D. Gannon, and M. Thomas. Overview of Grid Computing Environments. Informational, Global Grid Forum, February 2003.

[16] The Fractal Project. http://fractal.objectweb.org.

[17] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12):1753 – 1772, December 2002.

[18] D. Gannon and al. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. *Cluster Computing*, 5(3):325–336, July 2002.

[19] Dennis Gannon, Sriram Krishnan, Aleksander Slominski, Gopi Kandaswamy, and Liang Fang. Building Applications from a Web Service based Component Architecture. In *ICS 2004 Workshop on Component Models and Systems for Grid Applications*. Springer, 2004.

[20] V. Getov, P. Gray, and V. Sunderam. MPI and Java-MPI: Contrasts and Comparisons of Low-level Communication Performance. In *Supercomputing '99*, Portland, Oregon, USA, November 1999.

[21] K. Keahey, P. Fasel, and S. Mniszewski. PAWS: Collective Interactions and Data Transfers. In *10th International Symposium on High Performance Distributed Computing*, San Francisco, California, USA, August 2001.

[22] S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments at HIPS*, Santa Fe, New Mexico, USA, April 2004.

[23] A. Mayer, S. Mcough, 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*, pages 100–111, Baltimore, Maryland, USA, November 2002. volume 2536 of LNCS.

[24] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing, July 2002. discussed at Global Grid Forum 5.

[25] 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.

[26] ProActive. http://www.inria.fr/oasis/ProActive.

[27] W. Reynolds and M. Fatica. Standford Center for Integrated Turbulence Simulations. *Computing in Science and Engineering*, 2(2):54–63, 2000.

[28] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, to appear.