Collective Interfaces for Distributed Components

Françoise Baude, Denis Caromel, Ludovic Henrio and Matthieu Morel
INRIA - I3S - CNRS - Université de Nice Sophia Antipolis
Email: First.Last@sophia.inria.fr

Abstract

We propose to address collective communications in distributed components through collective interfaces. Collective interfaces handle data distribution, parallelism and synchronization, and they expose collective behaviors in the definition of components. We show, as an illustration, that collective interfaces allow the encoding of SPMD programming in a better structured and less error prone way. We verify the scalability and performance of collective interfaces in an experiment on up to 100 machines.

1 Introduction

Collective communications provide facilities to manage interactions between distributed entities. They define one-to-many, many-to-one and many-to-many interactions.

Many algorithms in distributed computation require collective communications, for instance: master-slave, divide-and-conquer or SPMD models. In this work, we focus on the data distribution, parallelism, and synchronization properties offered by collective communications. These properties are fundamental in distributed and Grid computing.

In the component programming paradigm, software components are defined as independent units of composition and deployment. Interactions with other components only occur through strictly defined interfaces.

Our objective is to address the specification of collective interactions in order to: simplify the programming model, simplify the assembly of complex component systems, provide a comprehensive description of off-the-shelf components. We propose to *expose* collective behaviors in the specification of the components, at the level of component interfaces, by introducing *collective interfaces*. Thanks to collective interfaces, component systems designers can specify parallelism, synchronization and data distribution, with a high-level view, and components exhibit their collective behavior.

In this paper, we propose a definition of collective interfaces in component models. Our proposal clarifies the exchange of data thanks to strongly typed collections. Our proposal also enforces separation of functional and non-functional concerns by separating parallelization, synchronization and data distribution code from the business code of components.

Our work is based on the Fractal component model [7], a simple and extensible model that enforces a strict separation between functional and non-functional concerns. Fractal is the basis of the Grid Component Model (GCM) currently being defined within the CoreGRID european project [9]. We introduce *multicast* and *gathercast* interfaces, that respectively define one-to-many and many-to-one interactions. We show how collective interfaces can advantageously facilitate code coupling, notably SPMD programming and the coupling of parallel codes. We demonstrate some of the advantages of collective interfaces in a simple SPMD application that uses both multicast and gathercast interfaces. We also evaluate the performance of our implementation of collective interfaces, based on an implementation of the GCM with the ProActive library, by performing measurements of our SPMD application on a cluster.

2 Context

Our proposal for collective interfaces specifically addresses multipoint interactions. We concretely defined an extension of the Fractal component model, integrated in the GCM specification, and we implemented collective interfaces in the ProActive library.

2.1 Collective Communications

Collective communications designate multipoint interactions between software entities: senders hold references on receivers. We are interested in the distribution of data, as well as in the *parallelism* and *synchronization* features offered by collective communications, therefore communication paradigms such as publish-subscribe or event-reaction are out of the scope of our proposal. Such features are usually implemented by a tier broker and can be easily integrated in component models by defining send and receive

interfaces.

Collective operations provide a higher and more sophisticated programming abstraction than sets of single invocations or send-receive operations. They allow a better structuration of communications, by notably bringing simplicity, programmability and expressiveness to programs [12].

2.2 The Fractal Component Model

Our proposal for collective interfaces is defined as an extension of the Fractal component model, but the concepts introduced are not restricted to this model, and can be applied to other component models as well.

Fractal is a modular and extensible component model proposed by INRIA and France Telecom, that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms or graphical user interfaces. The first version, specified in 2002, includes an API in Java.

The main features of the model are: hierarchical components, separation of concerns, reflection and openness. Fractal enforces separation of concerns between functional and non-functional features, by distinguishing functional and non-functional (control) access points to components.

Components communicate through their interfaces; client and server interfaces are bound to define communication paths. The binding mechanism uses an inversion of control pattern to inject dependencies between components. Fractal supports primitive bindings and composite bindings. A *primitive binding* is a binding between one client interface and one server interface of compatible types. A *composite binding* is a communication path between an arbitrary number of component interfaces, and it consists of an assembly of primitive bindings and binding components (binding components are convenient for modeling stubs, skeletons, adapters, etc.).

The GCM is an extension of the Fractal model geared at Grid computing being defined by the CoreGrid European project. By defining the GCM, the objective of the CoreGrid community is to design a standardized and effective component model for the Grid.

2.3 ProActive/GCM

The ProActive library is a middleware for Grid computing. It is based on the concept of *active object*. An active object is a remotely accessible object, with its own thread for managing its activity. Active objects communicate through asynchronous invocations and have no shared memory. ProActive features transparent distribution and parallel invocations. It offers middleware services such as load balancing, fault tolerance and migration. It also provides a deployment framework interfaced with most common distributed and Grid protocols, notably schedulers (e.g, LSF, GRAM).

The ProActive library has an extensible design. It is based on a meta-object protocol that deals with remote references to active objects, and reifies method invocations. Meta-objects enable non-functional features such as asynchronism or fault-tolerance. The meta-object architecture was enhanced to implement the Fractal component model. It is now being extended to support the GCM model, resulting in the ProActive/GCM framework.

We already used a prototype of the ProActive/GCM to build and deploy a numerical computation application for electromagnetism, on 4 clusters and more than 300 processors [19]. Although this prototype did not use collective interfaces, this experimentation allowed us to show the scalability of the model.

2.4 Related Work

There are two main approaches for collective communications: message-based, and invocation-based.

Message-based collective communications are commonly used in SPMD programming, and provided by frameworks such as MPI [17]. MPI provides collective communication operations, with the operations *scatter*, *broadcast*, *gather*, *reduce*, and *barrier*. Message-based communication with MPI however remains a low-level paradigm, complex to code and hard to debug.

Collective communications can also be performed using *group method invocations*: this allows a high-level design of collective communications, and parallelism can be provided by the implementations. A group method invocation consists in the invocation of the same method on a group of objects. This requires the *identification of the group*, and the *specification of the distribution of parameters*. The CORBA middleware was used for several experiments on group invocations [10, 21, 13]. Other frameworks are based on a strongly typed language such a Java: GMI [15] requires explicit management by the programmer, whereas ProActive Typed Groups [2] provide transparent asynchronous group invocations, but distribution strategies are not easily configurable. Those strategies can be set [4] but the code is tangled into the functional code. Moreover, to our knowledge, frameworks for multipoint collective communications based on invocations only provide one-to-n communications, and *do not provide n-to-one or n-to-n communications*.

Few component models explicitly define semantics of collective communications. They use messages or invocations, and most rely on *intermediate components*. ICENI [16] proposes a range of dispatch modes, including scat-

ter, gather, dispatch and reduce, but they are only applicable to arrays and require XML data. Several CCA frameworks [6] propose collective communications in order to provide efficient and modular code coupling, usually using intermediate components, and also through group method invocations. GridCCM [20] extends the Corba Component Model in order to efficiently couple parallel components. The Dream project [14] also defines a set of binding components for Fractal, for message-based communications.

3 Proposal

Problems of relying on intermediate entities are that: design is complexified; functional components cannot be used directly off-the-shelf because they not themselves specify nor implement collective behaviors. As an alternative, we argue that it is possible to define some of the semantics associated with multiway bindings at the level of the interfaces themselves, through the definition of collective interfaces.

In the type system of the Fractal model, the type of a component is defined by the types of the interfaces of this component. The type of an interface is defined by its name, its signature (in Java, the fully qualified named of a Java interface), its role (client or server), its contingency (optional or mandatory) and its cardinality. The Fractal model only defines two kinds of cardinalities: *singleton* and *collection*. They both only enable one-to-one bindings; the difference is that singleton interfaces (Figs. 1.a. and 1.b.) are unique and exist at runtime, whereas collection interfaces (Figs. 1.c. and 1.d) dynamically create instances of the specified interface type at binding time.

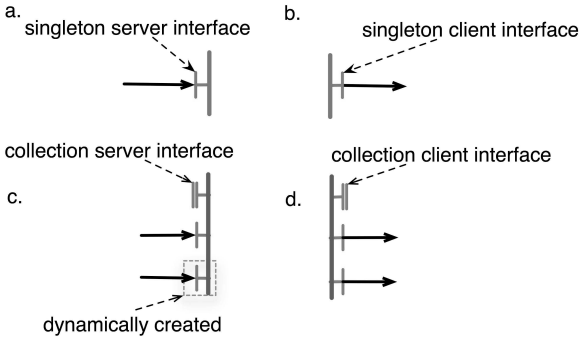

Figure 1. Singleton and collection interfaces in the Fractal model

The cardinalities of interfaces in the Fractal model only allow one-to-one communications between component interfaces. It is possible to introduce binding components that act as brokers and may handle different communication modes, allowing multiway communications. Unfortunately, binding components are separated from the definition of the business components that use it. This impedes the expression of a collective behavior in the specification of a business component, and may prevent communication optimizations.

In the context of the GCM, we proposed to introduce collective interfaces as a means to express the collective nature and behavior of component interfaces. Relatively to Fractal, the GCM adds new cardinalities in the specification of the type of a component interfaces, namely *multicast* and *gathercast*. A multicast or gathercast interface gives the possibility to manage a group of interfaces as a single entity, and it exposes the collective nature of the interface. The role and usage of multicast and gathercast interfaces are complementary: multicast interfaces are used for parallel invocations, parameter dispatch and result gathering, whereas gathercast interfaces are used for synchronization, parameter gathering and result dispatch.

3.1 Definitions

3.1.1 Multicast interfaces

Multicast interfaces provide abstractions for one-to-many communications, and are defined as follows:

> *A multicast interface transforms a single invocation into a list of invocations (Fig. 2.a).*

When a single invocation is transformed into a list of invocations, these generated invocations are forwarded to connected server interfaces (Fig. 2.a). The semantics of the propagation of the invocation and of the distribution of the invocation parameters are customizable, and the result of an invocation on a multicast interface is a list of results, or a reduction of it. Invocations forwarded to the connected server interfaces may occur in parallel, which is one of the main reasons for defining this kind of interface: it enables *parallel invocations*.

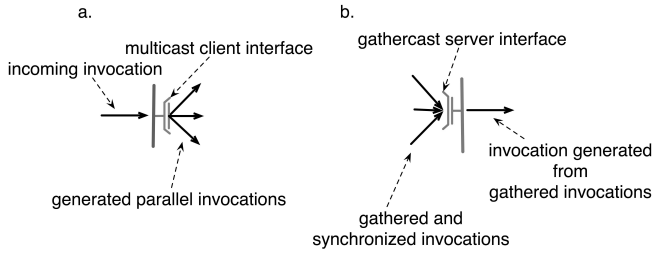

Figure 2. Multicast and gathercast interfaces

3.1.2 Gathercast interfaces

Gathercast interfaces provide abstractions for many-to-one communications:

A gathercast interface transforms a list of invocations into a single invocation (Fig. 2.b).

A gathercast interface coordinates incoming invocations before continuing the invocation flow: it can define synchronization barriers and gather incoming data. Invocation return values are automatically redistributed to the invoking components.

Synchronization barriers and gathering operations are customizable, as well as redistribution policies for invocations return values.

3.2 Semantics

The semantics define the management of invocations and data.

3.2.1 Invocations

Multicast interfaces generate invocations and forward them to connected interfaces; gathercast interfaces gather invocations and generate a single invocation forwarded to a connected interface.

An invocation on a multicast interface is transformed into a list of invocations. These invocations are executed on the components that are connected to a client multicast interface. A multicast invocation is a one-to-all or one-to-some communication. The communications can be synchronous or asynchronous and can be performed in parallel.

A gathercast interface gathers lists of invocations from connected components. Invocations on a gathercast interface are subject to a *transparent synchronization*, in order to collect data and create a new invocation.

3.2.2 Distribution of data

Collective interfaces transfer both invocation parameters and invocation results. In strongly typed languages, such as Java, data parameters are conveniently aggregated in *parameterized lists*; indeed, strong typing for collective interactions helps understanding data flows. For instance, `List<A>` designates a list of elements of type `A`.

Regarding the distribution of parameters: in multicast interfaces, it can follow two distinct modes: *broadcast*, or *scatter*. A broadcasted parameter is copied and sent to all connected components (Fig. 3.a). A scattered parameter is split and distributed across connected components, according to customizable rules (Fig. 3.c). For instance, block (one-to-one) or round-robin distributions may be proposed.

In gathercast interfaces, invocation parameters are aggregated as lists (Figs. 3.b and 3.d), or according to reduction rules (Fig. 3.f). Reduction rules define transformation operations on invocation parameters, e.g, it can consist in simply selecting a single parameter among invocation parameters from client interfaces, or in performing an election among those parameters.

Regarding the distribution of results, in multicast interfaces, results of invocations on connected interfaces are aggregated as lists (Figs. 3.a and 3.c), or reduced (Fig. 3.e). In gathercast interfaces, the result of an invocation on a gathercast interface is a list; this list may be either copied and sent to each connected interface (Fig. 3.c), or dispatched to connected interfaces, in a customizable manner (Fig. 3.d).

Figure 3 shows the symmetry between the multicast and gathercast interfaces regarding data distribution.

The ability to perform various schemes of data distribution has an incidence on the signatures of methods, as illustrated in Fig. 3: interfaces may be bound only if they are type-compatible, and the type compatibility is deduced from by the distribution mode (scatter, broadcast, reduction). Type compatibility for initial component designs may be checked statically. Because the Fractal component model is dynamic, bindings may change during execution, therefore type compatibility must also be checked at runtime.

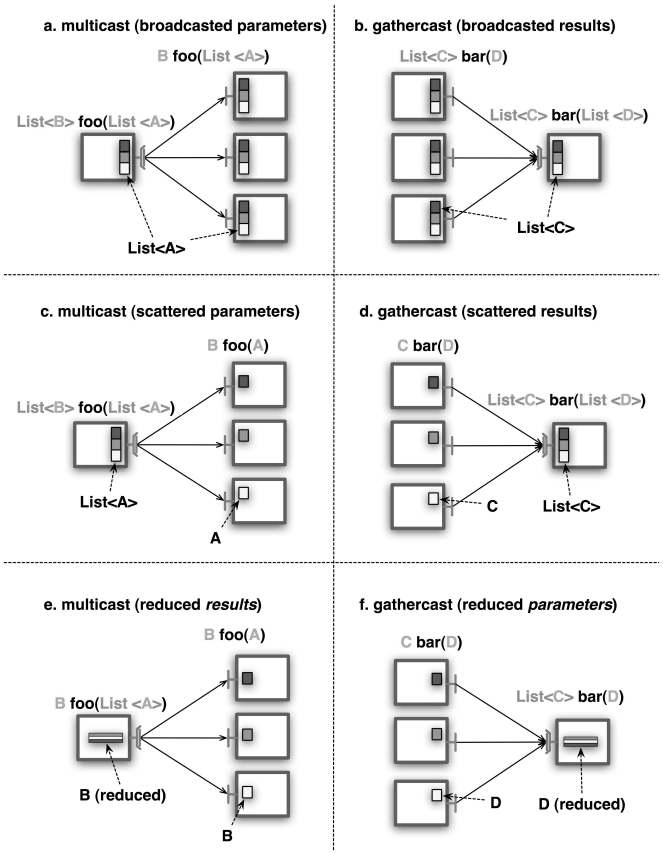

Figure 3. Data distribution and signatures compatibility

3.3 Configuration

Invocation forwarding and data transfer follow distribution functions, that can be formally specified, and that can be configured at the level of the interface.

The proposed model is open: it is possible to define various modes of synchronization, parallelism and data distribution, provided they are compatible with the semantics of collective interfaces.

The configuration of collective interfaces may be performed through dedicated component controllers. More simply, we propose to take advantage of attribute-based programming techniques, such as annotations in Java, in order to specify the distribution behaviors at the level of the interfaces themselves (we can annotate interfaces, methods or parameters).

The following snippet illustrates the configuration of a multicast interface. It uses a distribution mode that we propose in our framework (one-to-one); as the distribution mode is defined at the level of the interface element, it is applied to all parameters of all methods of the interface. Therefore, each element of type `B` contained in `lb` will be distributed to one connected interface. Elements of type `C` are not affected because the distribution function is only applied to declared parameterized lists: they are copied to each connected interface.

```
@ClassDispatchMetadata(
   mode=@ParamDispatchMetadata(mode=
       ParamDispatchMode.ONE_TO_ONE))
public interface MyInterface {
  public List<A> foo(List<B> lb, C c);
}
```

3.4 Implementation

We propose an implementation of collective interfaces, available as open source in the ProActive framework. It offers a configurable framework for multicast and gathercast interfaces. It offers respectively one-to-all and all-to-one modes, and gathering occurs by aggregation into lists. Reduction is being developed.

4 Component-based SPMD programming

The objective of this section is to show how gathercast and multicast may be combined in order to design component-based applications in a SPMD programming style.

4.1 SPMD Programming Concepts

SPMD is a programming model for parallel computing, and arguably the most widely used pattern in High Performance Computing. It stands for Single Program Multiple Data, as each task executes the same program but works on different data. Each copy of the program runs independently, on different data, and synchronization is provided through explicit messages.

The SPMD model maps easily and efficiently on distributed and parallel applications and distributed memory computing. It is the paradigm of choice for scientific parallel computing, with popular programming environments available for developers such as MPI.

Collective operations in the design of SPMD programs are a higher and more sophisticated programming abstraction than simple operations or send-receive operations. They allow a better structuration of communications, by notably bringing simplicity, programmability and expressiveness to programs [12].

4.2 From Message-Based SPMD to Component-Based SPMD

The popularity of SPMD programming led researchers to investigate ways to combine the SPMD paradigm with popular modern languages, in particular object-oriented languages. MPI 2 started by defining most of the library functions as class member functions in C++, although at a fairly low-level. Later, various approaches were undertaken, including: wrapping of MPI primitives [5] and implementations of the MPI specification [8]. Some approaches tried to adapt to the object-oriented paradigm by providing group method invocations in place of MPI collective operations [18]. In our research group, we experimented an extension of typed group communications with active objects: Object-Oriented SPMD [3]. OOSPMD programming relies on explicit synchronization barriers. Those barriers are implemented thanks to synchronization facilities offered by active objects.

In the context of Grid computing, the interaction between parallel codes, the deployment issues and the complexity of coupled applications using SPMD codes led to consider comprehensive programming models such as component-based programming. Some approaches were proposed in order to facilitate coupling of distributed parallel SPMD codes, notably CCAFFEINE [1] and GridCCM [20]. These approaches however only address the coupling of SPMD codes, not the design of SPMD application per se.

4.3 Component Based SPMD with Gathercast and Multicast Interfaces

We propose a novel approach for bringing the SPMD programming model into the component-based programming paradigm. This approach relies on collective interfaces.

4.3.1 Principles

The basic requirements of the SPMD model may be listed as:

1. Distributed replicated software entities.
2. Asynchronous communication between these entities
3. Collective communications: broadcast, scatter, gather.
4. Synchronization capabilities between selected distributed entities.

These requirements are answered as follows in the proposed model:

1. ProActive/GCM components are inherently distributed entities.
2. Components communicate through asynchronous method invocations.
3. Collective communications are provided through multicast and gathercast interfaces.
4. Synchronization is provided by gathercast interfaces, and the set of entities that must be synchronized is defined by the component assembly.

4.3.2 Usage and Benefits

SPMD capabilities are easy to implement in our model, however the design approach is a bit different from standard SPMD programming. Indeed, *all SPMD features are specified outside of the functional code*, through the gathercast and multicast interfaces. SPMD groups are defined by the assembly of the components, following the inversion of control pattern of Fractal binding mechanism. This brings reusability and flexibility for SPMD programs: programs can be adapted by simply changing bindings, and can be reused as the communication logic is separated from the application logic.

There is neither any explicit loop nor any explicit barrier: synchronization is automatic when invoking gathercast interfaces, and the computing process iterates by recursion triggered by neighbors: gathercast interface triggers the computation once the synchronization of incoming invocations is completed, then results are sent to the neighbors.

The assembly using gathercast interfaces provides an automatic logical synchronization (in Lamport's terms), therefore the global state of the application is always coherent at the end of the computation, even though some components may compute more rapidly than others.

In usual SPMD algorithms, the code of the application needs to explicitly handle the synchronization. The number of neighbors may vary and depends on the topology, for instance in a 2D grid like in the Jacobi example presented in Sect. 5, the entities may have 2, 3 or 4 neighbors. The synchronization between neighbors requires tedious coding efforts, through either a master entity, explicit barriers, or explicit data buffering techniques.

In the approach we propose, the synchronization between components is automatically and transparently handled, through a buffering mechanism provided by gathercast interfaces. There is no need to write tedious synchronizations in the code of the SPMD entities. Besides, parallel invocations and data distribution are automatically handled. As a result, collective interfaces enforce a strict separation between, on one hand, the code handling communication, data transfer, and synchronization, and the other hand, the code handling the applicative logic.

5 Experiment

We developed a classical and representative example in order to demonstrate and evaluate the capabilities of multicast and gathercast interfaces, as well as the SPMD programming possibilities provided by these interfaces.

For these purposes, we implemented a version of the Jacobi method for solving linear matrix equations. The Jacobi method is an algorithm in linear algebra for determining the solutions of a system of linear matrix equations with largest absolute values in each row and column dominated by the diagonal element. Each diagonal element is solved for, and an approximate value plugged in. The process is then iterated until it converges. The original sequential version is solved by initializing a large matrix, performing a computation on the matrix, then repeating this computation until convergence. It can be easily refactored into a distributed application by partitioning the original matrix. It is a simple example, easily implementable in a SPMD fashion, thanks to the decomposition into sub-matrixes, which compute locally their internal values and swap boundary values with their neighbors, as shown in Fig. 4. Moreover, this program already served as a use case for previous work on OOSPMD programming, which was compared to MPI, and provides us a reference.

For benchmarking purposes, we worked with a fixed number of iterations and fixed matrix values, and we did not wait for convergence.

The benchmarks are intended to answer three questions:

- Is a component-based approach usable and scalable for solving SPMD-style problems?
- What is the performance of the component-based approach compared to an approach using explicit barriers for synchronization?

- What is the performance of multicast communications against sequential one-way invocations through collection interfaces?

5.1 Design

5.1.1 Typing

There is only one type of components in this application: the sub-matrix type. It defines a mandatory gathercast server interface, and optional multicast and collection interfaces, depending whether borders data is sent using multicast (Fig. 4) or with successive invocations on a collection interface. Border data values are encapsulated as `LineData` objects.

5.1.2 Borders reception and synchronization

The server interface of cardinality gathercast is defined as follows:

```
public interface GathercastDataReceiver {
      public void exchangeData(List<
          LineData> borders);
}
```

As we can see from the signature of the only method, a list of `LineData` elements is expected from connected client interfaces. Once all `LineData` elements are received by the gathercast interface, they are gathered into a list of `LineData` elements, and the method defined by `GathercastDataReceiver` is invoked on the sub-matrix component.

5.1.3 Borders sending

Sending border values can be performed in two manners, and this is determined during the binding process: either multicast or collection interfaces are used.

The multicast interface exhibits the following signature:

```
public interface MulticastDataSender {
      @MethodDispatchMetadata(mode =
          @ParamDispatchMetadata(mode=
          ParamDispatchMode.ONE_TO_ONE))
      public void exchangeData(List<
          LineData> borders);
}
```

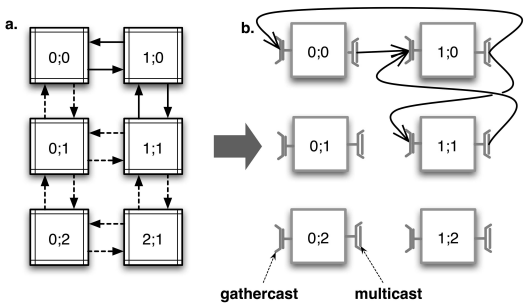

Figure 4. Jacobi computation: borders exchanges through multicast and gathercast interfaces (communications are represented only for component (1;0))

As we can see, a list of `LineData` elements is given as a parameter to the `exchangeData` method. This list is built from the values of the borders of the current matrix. The distribution of the `LineData` elements among the connected server interfaces is specified by the annotation: a *block* distribution, which means that the server interface of rank *i* automatically receives the `LineData` element of rank *i* in the list.

This communication pattern is represented in Fig. 4.

We implement the Jacobi algorithm in one method, that triggers the computation and the communication with neighbors. There is no need for explicit synchronization code:

```
public void exchangeData(List<LineData>
    borders) {
   replaceOldBordersWithNewData(borders);
   // compute matrix including borders
   matrixComputation();
   // exchange new borders with neighbors
   // (triggers next iteration)
   neighbors.exchangeData(newBorders);
}
```

We also bench an alternative way of sending borders data: sequential invocations on a collection of interfaces. This allows a comparison between sequential invocations and multicast invocations.

5.2 Benchmarks

For comparison purposes, the benchmarks are performed on 3 versions of the Jacobi application: an OOSPMD version (using explicit synchronization barriers), a component version without multicast (using collection interfaces, and gathercast interfaces), and a component version with multicast (using multicast and gathercast interfaces).

The component based version uses the same computation algorithm than the reference OOSPMD version, the

only differences are a different process for object creation and binding, and the absence of explicit synchronization barriers.

We fix the size and content of the matrix, and measure the execution time for a fixed number of iterations. The application is deployed on the *azur* cluster of our lab, INRIA Sophia Antipolis, which consists of 105 processors of type AMD Opteron, CPU clock of 2GHz, and 2GB of RAM. The Java runtime is Sun's 1.5.0_04 version, configured with a maximum heap size of 1.5GB.

The global matrix is decomposed into submatrixes, and each distributed active object or component works with one submatrix.

The parameters which vary are the number of machines used for the computation and the version of the implementation. The results of these experiments are represented in Fig. 5.

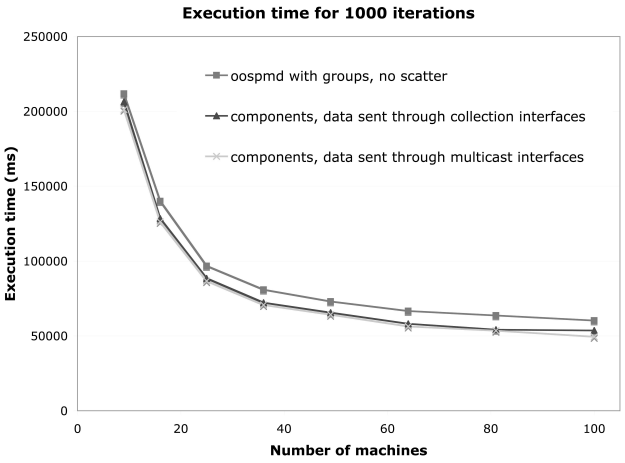

Figure 5. Jacobi computation benchmarks

5.3 Analysis

We draw three conclusions from this example, regarding respectively the applicability of collective interface, the performance, and the generalization of the SPMD approach.

First, this benchmark is an excellent *proof on concept of collective interfaces* in the GCM component model: we managed to deploy and run an SPMD application on up to 100 machines and collective communications were extensively used.

Second, this benchmark demonstrates that our approach for SPMD programming using components and collective interfaces is *valid*, and even *performant compared to the existing OOSPMD-based implementation*. The benchmark results in Fig. 5 highlight three characteristics of the collective interfaces. The first characteristic is an almost linear performance speedup when increasing the number of processors: a correct behavior for a clustered application. The second characteristic is a faster execution for the component based versions compared to the OOSPMD-based version. We explain this behavior by the explicit synchronization barriers used in the OOSPMD version: setting barriers implies sending extra requests to the SPMD group members; these additional requests harm performance because sending a request requires a complex and costly reification process, . On the contrary, there is no explicit barrier in the component version, therefore no extra request. The third characteristic is a slightly better performance when using multicast interfaces than when making successive invocations on members of collection interfaces. We explain this gain by the multithreading mechanisms involved when using multicast interfaces.

Third, from a qualitative point of view, this example is much *easier to implement* than SPMD solutions with explicit synchronization barriers, for instance OOSPMD solutions. By comparison, the equivalent MPI program for this benchmark would require explicit and error-prone synchronization, either through collective primitives or by matching send-receive statements.

6 Conclusion

In this paper we presented a proposal for addressing collective communications through collective interfaces of software components. Collective interfaces offer the following advantages:

- they expose collective behaviors in the definition of components;
- they provide automatic data distribution, parallelism, and synchronization;
- they enforce a separation between the communication logic and the business logic;
- they guarantee structured programming;
- no intermediate component is required.

We implemented collective interfaces in the ProActive Grid middleware, and we demonstrated the performance and applicability of two kinds of collective interfaces, multicast and gathercast interfaces. As an illustrative example, we showed how collective interfaces enable SPMD programming with components in a non-intrusive style with respect to the applicative logic. However, the expressiveness and the capabilities of the collective interfaces are not restricted to SPMD programming; they allow to express any form of collective communication in a structured way.

We plan to apply collective interfaces for facilitating the coupling of parallel codes (also referred to as the "MxN problem" [6]). We intend to develop more complex and varied applications, in particular the NAS Grid benchmarks [11], for which we can take advantage of both the hierarchical nature of the Fractal component model and the collective interfaces.

We are currently extending our implementation of collective interfaces with dispatch and reduction capabilities (cf 3.4). The model can be extended so that these capabilities are generalized: aggregation can be extended to any transformation from a list to any type of data; and dispatch mode can be extended to take into account any way to split, transform, and dispatch data. Finally, the component type system can be extended to consider collective interfaces as a subtype of simple interfaces. This allows a parallel component to safely replace a sequential one, and thus improve parallelization of an application without refactoring it.

Acknowledgements

This work was partially supported by France Telecom under external research contract 46127879. The authors also thank the CoreGRID community for its feedback since the inception of this work.

References

[1] B. Allan, R. Armstrong, A. Wolfe, J. Ray, D. Bernholdt, and J. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14:323–345, 2002.

[2] 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. ACM Press, 2002.

[3] L. Baduel, F. Baude, and D. Caromel. Object-Oriented SPMD. In *IEEE International Symposium on Cluster Computing and Grid, CCGrid 2005*, volume 2, pages 824– 831, May 2005.

[4] L. Baduel, F. Baude, N. Ranaldo, and E. Zimeo. Effective and Efficient Communication in Grid Computing with an Extension of ProActive Groups. In *JPDC, 7th International Worshop on Java for Parallel and Distributed Computing at IPDPS*, Apr. 2005.

[5] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim. mpiJava: An Object-Oriented Java interface to MPI. In *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*, San Juan, Puerto Rico, Apr. 1999.

[6] F. Bertrand, R. Bramley, K. B. Damevski, J. A. Kohl, D. E. Bernholdt, J. W. Larson, and A. Sussman. Data Redistribution and Remote Method Invocation in Parallel Component Architectures. In *19th International Parallel and Distributed Processing Symposium: IPDPS*, 2005.

[7] E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model Specification, 2004.

[8] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12:1019–1038, 2000.

[9] CoreGRID Network of Excellence. http://www.coregrid.net.

[10] P. Felber, B. Garbinato, and R. Guerraoui. The Design of a CORBA Group Communication Service. In *Symposium on Reliable Distributed Systems*, pages 150–159. IEEE Computer Society, October 1996.

[11] M. Frumkin and R. F. V. der Wijngaart. NAS Grid Benchmarks: A Tool for Grid Space Exploration. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC '01)*, volume 00, page 0315. IEEE Computer Society, 2001.

[12] S. Gorlatch. Send-Receive Considered Harmful: Myths and Realities of Message Passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, January 2004.

[13] K. Keahey and D. Gannon. PARDIS: A Parallel Approach to CORBA. In *6th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1997.

[14] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs. In *3rd Workshop on Reflective and Adaptive Middleware (RM'2004)*, Toronto, Canada, October 2004.

[15] J. Maassen, T. Kielmann, and H. E. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'02)*, Mar. 2002.

[16] A. Mayer, S. Mcough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington. Meaning and Behaviour in Grid Oriented Components. In *Third International Workshop on Grid Computing, GRID*, volume 2536 of *LNCS*, pages 100–111, 2002.

[17] MPI: A Message-Passing Interface Standard. Technical report, MPI Forum, University of Tennessee, Knoxville, Tennessee, June 1994.

[18] A. Nelisse, T. Kielmann, H. E. Bal, and J. Maassen. Object-based Collective Communication in Java. In *Joint ACM Java Grande - ISCOPE Conference*, pages 11–20, Palo Alto, California, USA, 2001. ACM Press.

[19] N. Parlavantzas, V. Getov, M. Morel, F. Baude, F. Huet, and D. Caromel. Componentising a scientific application for the grid. Technical Report TR-0031, Institute on Grid Systems, Tools and Environments, CoreGRID, April 2006.

[20] C. Pérez, T. Priol, and A. Ribes. A Parallel CORBA Component Model for Numerical Code Coupling. *International Journal of High Performance Computing Applications (IJHPCA)*, 17(4):417–429, 2003.

[21] C. René and T. Priol. MPI code encapsulating using parallel CORBA object. *Cluster Computing*, 3(4):255–263, 2000.