HAL
archives-ouvertes.fr

Generation of Distributed Parallel Java Programs

Pascale Launay, Jean-Louis Pazat

To cite this version:

Pascale Launay, Jean-Louis Pazat. Generation of Distributed Parallel Java Programs. [Research Report] RR-3358, INRIA. 1998. <inria-00073331>

HAL Id: inria-00073331

https://hal.inria.fr/inria-00073331

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Generation of distributed parallel Java programs

Pascale Launay and Jean-Louis Pazat

No 3358

Février 1998

THÈME 1

Rapport de recherche

ISSN 0249-6399

Generation of distributed parallel Java programs

Pascale Launay* and Jean-Louis Pazat†

Thème 1 — Réseaux et systèmes
Projet CAPS

Rapport de recherche n ° 3358 — Février 1998 — 14 pages

Abstract: The aim of the *Do!* project is to ease the standard task of programming distributed applications using Java. This paper gives an overview of the parallel and distributed frameworks and describes the mechanisms developed to distribute programs with *Do!*.

Key-words: Java, framework, parallel programming, programs transformations, distribution

(Résumé : tsvp)

* Pascale.Launay@irisa.fr
† Jean-Louis.Pazat@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Répartition de programmes Java parallèles

Résumé : Le but du projet *Do!* est de faciliter la programmation d'applications distribuées, et utilise le langage Java. Ce papier présente un framework parallèle et un framework distribué et décrit les mécanismes développés pour la répartition de programmes avec l'environnement *Do!*.

Mots-clé : Java, framework, programmation parallèle, transformations de programmes, répartition

1 Introduction

The aim of the *Do!* project is to ease the task of programming distributed applications using object-oriented languages (namely Java). The *Do!* programming model is not distributed, but is explicitly parallel. It relies on structured constructs (PAR) and shared objects. This programming model is embedded in a framework described in section 3.1, without any extension to the Java language. Programs distribution is expressed through distribution of COLLECTIONS. The *Do!* preprocessor transforms the parallel program into a distributed program that uses a distributed framework described in section 3.2. The generated codes rely on a run-time managing remote creation of objects and remote methods invocations. These mechanisms are described in section 2.

2 Programs distribution

During the execution of an object-oriented program, the control flow runs successively in the distinct objects of the program, through methods invocations. We distribute the programs control flow by distributing their objects on distinct processors. This requires:

- to map the objects of the program on distinct processors: when an object is located on a processor, its attributes are managed by the local memory and its methods run on this processor;
- to have a mechanism allowing objects to access objects located on distinct processors (remote method invocations).

Programs distribution is obtained by objects transformations to allow transparent remote accesses to objects (figure 1) and relies on a run-time allowing remote creations of objects.

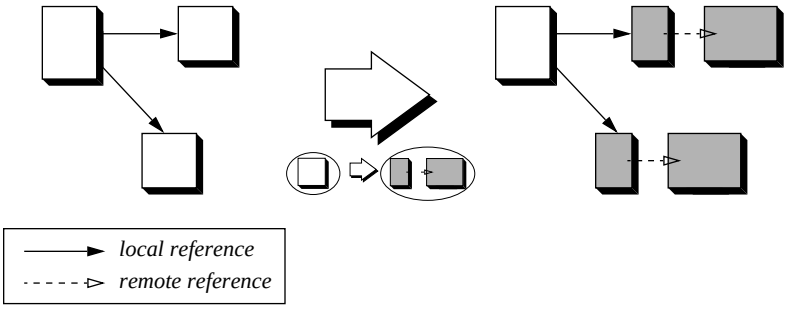

Figure 1: Program distribution

To map objects on distinct processors, the object creation semantics has been extended to allow remote creation of objects. This extension is implemented through classes using Java standard features. It is described in section 2.1.

To allow objects to invoke remote objects methods, we use code transformations: a preprocessor transforms objects in a way allowing transparent methods invocations wherever they are located (section 2.2).

2.1 Remote creation of objects

The mapping of an object on a processor occurs at the object creation. We have implemented an object creation with a semantics allowing the user to enforce the location of the new instance he creates. Thus, the object that initializes the creation and the one that is created may be located on distinct hosts. We have developed a runtime that supplies remote object creation through a method `remoteNew`, instead of the standard `new` statement. This runtime masks the problems of managing distinct Java Virtual Machines (JVMs) and locating objects maintained on distinct hosts. It is composed of classes, using Java standard features without any change to the JVM. It relies on creation servers running on each processor, that manage object creations using the reflection mechanism.

2.1.1 Creation servers.

To avoid synchronizations between processors involved in an object creation, a *creation server* runs on each host. Each server is responsible for local creations of objects and manages invocations to the remote servers: a server handles remote references to all other servers. Each server has its own thread, distinct from the application computing thread.

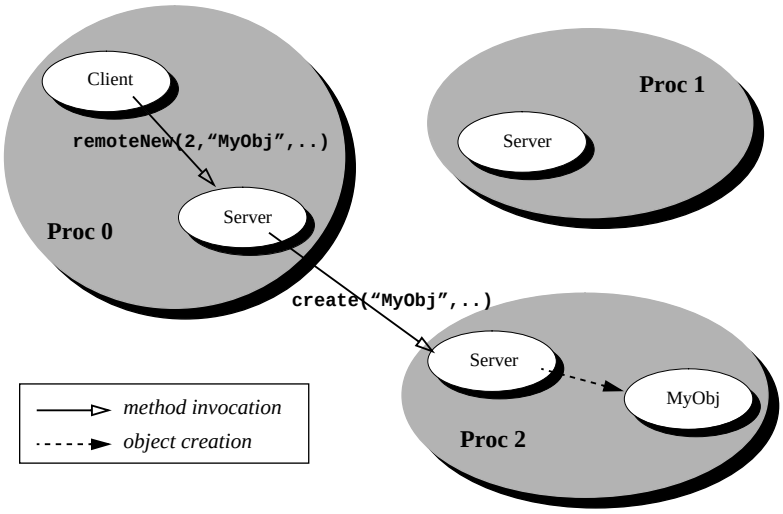

Figure 2: Creation servers

Clients only deal with their local server. Given an identification of the class to instantiate and its location, the server returns to the client a reference to the new object, possibly mapped on a remote host: the server creates the new object if it is local, or invokes the remote server responsible for the creation through its remote reference (figure 2).

2.1.2 Reflection.

To extend the object creation semantics, we have to manipulate language abstractions (such as *classes*, *constructors*, *instantiation*...) at run-time: the reflection mechanism[1] allow us to represent abstractions by first-class objects accessible in the program, proceeding in two steps (figure 3):

- *reification:* abstractions are reified into *meta-objects* (e.g. a class is represented by an object of type `Class`), that can be used like a basic object (e.g. retrieve the class name, its constructors, ...)
- *reflection* occurs when these meta-objects are transformed into first-class objects (e.g. the `Class` object is transformed into a basic object relevant to the application).

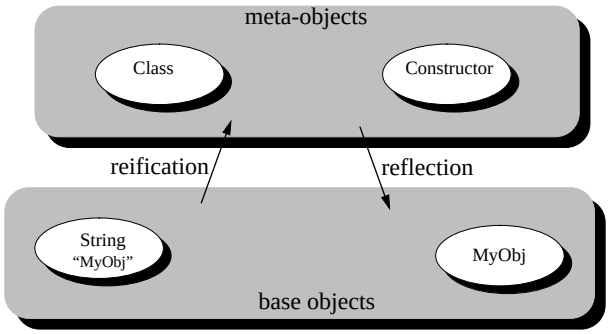

Figure 3: Reflection

We process by first reifying the type identifier (the class name) into an object `Class`, that we use to find the constructor (represented by an object of type `Constructor`), implementing a `newInstance` method used to create and initialize the new object.

2.2 Transparent remote invocations

To distribute programs in a transparent way, we have to offer mechanisms masking the locations of objects: accesses to local and remote objects must be expressed in the same way in the source code. Objects that may be accessed from remote hosts must be transformed, but

[1] Java offers reflection utilities as a *core API* (part of the standard JDK) [7].

all the objects of the program should not be transformed, either because this is not possible (when the source code is not available) or because the programmer does not wish to make them accessible (when the object is never accessed from remote hosts). The programmer guides the preprocessor by labeling the classes to transform *accessible* (implementing the interface `Accessible`). We developed a preprocessor (with the JavaCC [15] parser generator) that transforms classes to be able to invoke the objects methods wherever they are located. We split the source class in two classes (figure 4), the *proxy class* and the *implementation class*:

- the *proxy class* has the same name and methods signatures as the source class, but the methods bodies consist in remote invocations of the corresponding implementation in the *implementation class*. The proxy object handles a reference on the *implementation object*; it catches the invocations to the source object and redirects them to the right host. The proxy object state is never modified, so it can be replicated on all processors getting a reference on the source object.
- the *implementation object* contains the source methods implementations. It is not replicated and is located where the source object has been mapped. It is shared between all *proxy objects*.

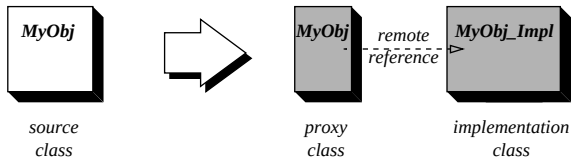

Figure 4: Object transformation

At object creation, the *proxy object* is instantiated locally with the basic creation mechanism, and then uses the remote creation mechanism described above to instantiate the *implementation object* on the target processor. Additional constructors are generated in the *proxy object* with an additional parameter (the processor identification) to enforce the *implementation object* location. The default constructors are used to create local objects.

2.3 Run-time – the Java RMI

The remote creation and invocation are implemented using the Java RMI [8], that offers run-time mechanisms to access remote objects, but is not sufficient for our purpose (it does not offer an easy programming model):

- clients of *remote objects*[2] interact with remote interfaces, never with the implementation classes of those interfaces and they must deal with additional exceptions that can occur during a remote method invocation. So, the RMI is not transparent for the user;

[2] a *remote object* is one whose methods can be invoked from another JVM, potentially on a different host

```
MyClass anInstance = new MyClass(); /* object creation */
anInstance.aMethod(); /* local method invocation */
```

Non distributed code

```
   Client code:
try { /* getting a reference to the remote object */
   MyClass_Int anInstance = (MyClass_Int)
      java.rmi.Naming.lookup("//"+remoteHostId+"/anInstanceId"); }
   anInstance.aMethod(); /* remote method invocation */
catch (NotBoundException e) { ... }
catch (MalformedURLException e) { ... }
catch (UnknownHostException e) { ... }
catch (RemoteException e) { ... }

   Server code:
MyClass_Impl anInstance = new MyClass_Impl(); /* object creation */
try { /* object registration */
   java.rmi.Naming.bind("//"+myHostId+"/anInstanceId", anInstance); }
catch (AlreadyBoundException e) { ... }
catch (MalformedURLException e) { ... }
catch (UnknownHostException e) { ... }
catch (RemoteException e) { ... }
```

Distributed code with RMI

```
MyClass anInstance = /* remote creation */
   (MyClass)DoRuntime.remoteNew(1,"MyClass",new Object[0]);
anInstance.aMethod(); /* transparent remote method invocation */
```

Do! distributed code

Figure 5: Object creation and invocation

- to locate remote objects, a name server stores named references to remote objects and provides methods to access thoses references. The programmer has to manage the naming of objects and the synchronizations between the client and the owner of an object;
- the parameter passing semantics depends on whether the argument is a *remote object* or not: *nonremote objects* are passed by copy, whereas *remote objects* are passed by reference. Moreover, the RMI parameter passing semantics may be different when used locally (when the two objects involved in the method invocation are in the same JVM).

We address this problem by transforming all objects that may be accessed from remote hosts: the *implementation object* is never copied, so the local invocations semantics is preserved (parameters are passed *by reference*).

Figure 5 shows the distribution of a code consisting in an object creation and invocation by using the Java RMI or the Do! facilities.

3 Parallel programs distribution

Distributing sequential programs may be interesting for example to exploit the location of specific resources (e.g. data collected in geographically distributed sites, expensive hardware resources), but the sequential programming model do not benefit from the distributed environment advantages: only one processor is used at a time. The parallel programming model allows to run control flows concurrently; in a multithreaded environment, the parallel activities may run concurrently, but sharing a global memory; in a distributed environment, activities run concurrently on distinct computers.

In the Java language, parallelism is expressed by using `Thread` objects, that allow to start asynchronous activities. The reification of activities into objects allow us to distribute concurrent activities in the same way as we distribute objects of a program.

Java Threads provide us with asynchronism, but not with a structured parallel programming model. To ease parallel programming, we have defined a parallel framework (section 3.1), restricting the expressiveness of the Java parallel programming features. Using this framework allows the programmer to write parallel programs by providing implementation for some objects (tasks and their arguments), the control and synchronizations being managed by the framework.

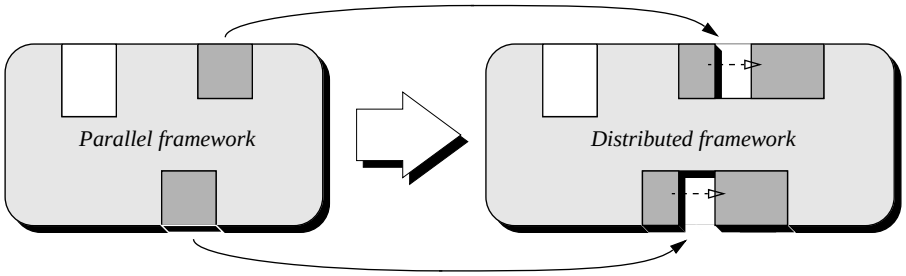

Figure 6: Parallel program distribution

This framework provides us with structured parallel programs. We also use this structure for program distribution: we have defined a distributed framework (section 3.2), with the same interface as the parallel framework. To distribute a program expressed with our parallel framework, we use the distributed framework instead of the parallel framework,

with the objects defined by programmer, some of those objects being transformed to allow transparent remote accesses using the mechanisms described above (section 2). Figure 6 represents a parallel program distribution by using the distributed framework and by objects transformations.

The parallel framework is based on COLLECTIONs of tasks and data. Parallel programs are distributed by using the distributed framework, based on the parallel framework, but using distributed COLLECTIONs to store distributed tasks and data.

3.1 Parallel framework

In this section, we give an overview of the parallel framework described in [13]. The aim of this framework is to separate computations from control and synchronizations between parallel tasks allowing the programmer to concentrate on the definition of tasks. This framework provides a parallel programming model without any extension to the Java language. It is based on active objects (TASKs) and on a parallel construct (PAR) that allows to execute COLLECTIONs of tasks in parallel.

A task is an object extending the class TASK implemented using the Java THREAD class; its behavior is inherited from the *run* method of TASK, that can be re-defined to implement the task specific behavior. A task is activated by invoking its *run* method and is active during its whole execution. The asynchronous invocation of *run* methods is used for parallel execution of tasks.

We have extended the operators design pattern [9] designed to express regular operations over COLLECTIONs through OPERATORs; COLLECTIONs manage the storage and the accesses to elements. OPERATORs represent autonomous agents processing elements. Including the concept of active objects, we offer a parallel programming model: active and passive objects are stored by COLLECTIONs; task parallelism is a processing over a TASKS COLLECTION, tasks parameters being grouped in a DATA COLLECTION.

The class PAR implements the parallel activation of tasks grouped in a COLLECTION with their parameters stored in another COLLECTION. A synchronization occurs at the end of tasks execution. Nested parallelism can be expressed by including an object PAR in a COLLECTION of tasks (the class PAR extends the class TASK).

Figure 7 shows an example of a simple parallel program, using ARRAY collections; the class MY_TASK represents the program specific tasks; it extends TASK, and takes an object of type PARAM as parameter.

3.2 Distributed framework

The distribution of parallel programs must preserve the program semantics: we consider that running two parallel independent tasks in a shared multi-threaded environment should be equivalent to running the same tasks on two distinct processors. If the tasks modify shared objects, we consider that the semantics is preserved if the shared objects are not replicated and have the same concurrent access management policy.

```
import DO.SHARED.*;

public class SIMPLE_PARALLEL {
  public static void main (String argv[ ]) {
    ARRAY tasks = new ARRAY(N);
    ARRAY data = new ARRAY(N);
    for (int i=0; i<N; i++) {
      tasks.add (new MY_TASK(), i);
      data.add (new PARAM(), i); }

    PAR par = new PAR (tasks,data);
    par.call();
  }
}
```

Figure 7: A simple parallel program

In the parallel framework, we use collections to manage the storage and accesses to active and passive objects. The distributed framework is obtained by the distribution of collections in the parallel framework. A distributed collection is a collection that manages distributed elements (objects mapped on distinct processors), the location of the elements being masked to the user. When a client retrieves a remote element through a distributed collection, it gets a remote reference to the element, that can be invoked transparently. Distributed tasks are activated by remote invocation of their run methods.

The physical implementation of COLLECTIONS is redefined without changing their interface through inheritance: COLLECTION defines the collection interface and is implemented by two classes (figure 8):

- ND_COLLECTION implements a non distributed collection
- D_COLLECTION implements a distributed collection.

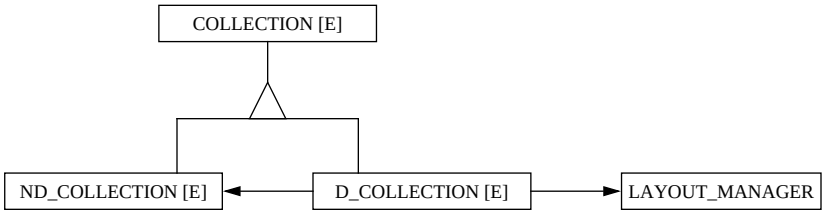

Figure 8: Non distributed and distributed collections

A distributed collection is composed of fragments, mapped on distinct processors, each fragment managing the local subset of the collection elements. A local fragment is a non distributed collection (figure 8: D_COLLECTION uses ND_COLLECTION); the distributed collection processes an access to a distributed element by remote invocation to the fragment *owning* this element (local to this element).

```
abstract public class LAYOUT_MANAGER {

  /* transforms a global key into a local key */
  abstract public KEY gl2l (KEY k);

  /* returns the owner of the element identified by k */
  abstract public int owner (KEY k);

  /* returns the list of processors owning at least one element */
  abstract public int[ ] owners();

}
```

Figure 9: The LAYOUT_MANAGER class

To access an element of a distributed collection, a client identifies this element with a global identifier (relative to the whole set of elements). The distributed collection has to identify the fragment owning the element and transform the global identifier into a local identifier relevant to the local collection. The task of converting a global identifier into the corresponding local identifier and the owner identifier devolves on a LAYOUT_MANAGER object (figure 9). Different LAYOUT_MANAGER implementations provide different distribution policies. The user guides the collection distribution by choosing a LAYOUT_MANAGER implementation. The figure 10 represents an example of class hierarchy of distribution layout

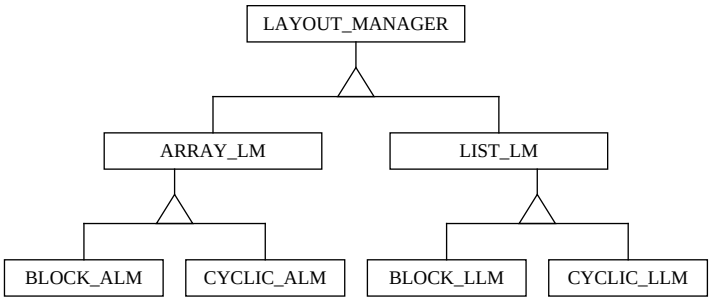

Figure 10: Distribution layout managers

managers; for example, the distribution of arrays by blocks is obtained through the use of the BLOCK_ALM class.

4 Related work

Tools produce Java parallel (multi-threaded) programs, relying on a standard Java runtime system; they do not generate distributed programs. The High Performance Java project at Indiana University comprises the development of JAVAR [2], a restructuring compiler to generate parallel programs from sequential Java programs with annotations, and JAVAB [1], a tool to automatically detect and exploit implicit loop parallelism in bytecode. Roudier and Ichisugi [5] have defined Tiny Data-Parallel Java as an example application of their extensible Java preprocessor EPP: methods identified as data-parallel methods are translated by EPP in Java multi-threaded codes.

Other projects are based on distributed objects and remote method invocations. Philippsen and al [14] have extended the Java language by adding *remote* objects at the language level. A runtime manages creation, distribution and migration of objects. The programming model is explicitly distributed but remote method invocation is transparent from the programmer point of view. Kalé and al [12] have defined a parallel extension to Java, providing dynamic creation of remote objects with load balancing, and object groups. It is implemented using the Converse [11] interoperability framework, which makes it possible to integrate parallel libraries written in Java with modules in other parallel languages in a single application.

Some environments rely on a data-parallel programming model and a SPMD execution model, without any extension to the Java language. Ivannikov and al [6] have defined a class library, containing a set of Java classes and interfaces for the development of data-parallel programs using a run-time based on MPI. EPEE (Eiffel Parallel Execution Environment) [10] is an object oriented design framework developed in our team. It proposes a programming environment where data and control parallelism are totally encapsulated in regular Eiffel classes, without any extension to the language nor modification of its semantics. This research is very close to ours but uses no program transformation and the execution model is limited to the SPMD model.

Like in the *Do!* project, parallelism may be introduced through the notion of active objects. Caromel and al are developing Java// which is based on active objects and uses a library that is itself extensible by the programmers. This work is based on Eiffel//[3] and C++// [4], and uses the reflection mechanism through a Meta-Object Protocol.

5 Conclusion

In this paper, we have presented the distribution of Java parallel programs. The programming model we propose is explicitly parallel, based on a parallel framework, without any extension to the Java language. Distributed programs are generated by objects transforma-

tions (a preprocessor transforms objects to allow remote accesses) and generated programs use a distributed framework based on the parallel framework. The runtime uses the standard Java Virtual Machine, without any change in the bytecode.

This framework can be extended to handle dynamic creation of tasks, through dynamic collections (e.g. lists); distributed scheduling could also be added to the framework.

Another foreseen extension of this work is to use a more efficient run-time than the RMI.

References

[1] A. J. C. Bik and D. B. Gannon. Exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.

[2] A. J. C. Bik, J. E. Villacis, and D. B. Gannon. JAVAR : a prototype Java restructuring compiler. *Concurrency, Practice and Experience*, 1997. To appear.

[3] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[4] D. Caromel, F. Belloncle, and Y. Roudier. The C++// system. In G. Wilson and P. Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[5] Y. Ichisugi and Y. Roudier. Integrating data-parallel and reactive constructs into Java. In *Proc. of France-Japan Workshop on Object-Based Parallel and Distibuted Computation (OBPDC'97)*, France, October 1997. To appear in LNCS, Springer-Verlag.

[6] V. Ivannikov, S. Gaissaryan, M. Domrachev, V. Etch, and N. Shtaltovnaya. DPJ : Java class library for development of data-parallel programs. Institute for System Programming, Russian Academy of Sciences, 1997.

[7] Javasoft. Java core reflection – API and specification. `ftp://ftp.javasoft.com/docs/jdk1.1/java-reflection.ps`, January 1997.

[8] Javasoft. Java remote method invocation specification – revision 1.42. `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.ps`, December 1997.

[9] J. M. Jézéquel and J. L. Pacherie. Parallel operators. In P. Cointe, editor, *ECOOP'96*, number 1098 in LNCS, Springer Verlag, pages 384–405, July 1996.

[10] J. M. Jézéquel, F. Guidec, and F. Hamelin. Parallelizing object oriented software through the reuse of parallel components. In *Object-Oriented Systems*, volume 1, pages 149–170, 1994.

[11] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse : an interoperable framework for parallel programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawai, April 1996.

[12] L. V. Kalé, M. Bhandarkar, and T. Wilmarth. Design and implementation of Parallel Java with a global object space. In *Proc. Conf. on Parallel and Distributed Processing Technology and Applications*, Las Vegas, Nevada, July 1997.

[13] P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *HPCN'98*, LNCS, Springer Verlag, Amsterdam, April 1998. To appear.

[14] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. In *PPoPP*, June 1997.

[15] S. Sankar, S. Viswanadha, and R. Duncan. Java Compiler Compiler – the Java parser generator. `http://www.suntest.com/JavaCC/`, November 1997.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399