Generation of Distributed Parallel Java Programs

Pascale Launay and Jean-Louis Pazat

IRISA, Campus de Beaulieu, F35042 RENNES cedex
Pascale.Launay@irisa.fr, Jean-Louis.Pazat@irisa.fr

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!.

1 Introduction

Many applications have to cope with parallelism and distribution. The main targets of these applications are networks and clusters of workstations (NOWs and COWs) which are cheaper than supercomputers. As a consequence of the widening of parallel programming application domains, programming tools have to cope both with task and data parallelism for distributed program generation.

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 parallelism and shared objects. This programming model is embedded in a framework described in section 2, without any extension to the Java language. The distributed programs use a distributed framework described in section 3. Program distribution is expressed through the distribution of collections of tasks and data; the code generation is described in section 4.

2 Parallel Framework

In this section, we give an overview of the parallel framework described in [8]. 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 structured parallelism (through the class PAR) that allows the execution of tasks grouped in a collection in parallel.

The notion of active objects is introduced through task objects. A task is an object which type extends the TASK class, that represents a model of task, and is part of the framework class library. The task default behavior is inherited from the behavior described in the TASK class through its *run* method. This behavior

can be re-defined to implement a specific task behavior. A task can be activated synchronously or asynchronously.

We have extended the operators design pattern [6] designed to express regular operations over COLLECTIONs through OPERATORs, in order to write data-parallel SPMD programs: a COLLECTION manages the storage and the accesses to elements; an OPERATOR represents an autonomous agents processing elements. Including the concept of active objects, we offer a parallel programming model, integrating task parallelism: active and passive objects are stored by collections; a parallel program consists in a processing over task collections, task parameters being grouped in data collections. The PAR class implements the parallel activation and synchronization of tasks, providing us with structured parallelism. Nested parallelism can be expressed, the PAR class being a task.

Figure 1 shows an example of a simple parallel program, using collections of type ARRAY; the class MY_TASK represents the program specific tasks; it extends TASK, and takes an object of type PARAM as parameter.

```
import DO.SHARED.*;

/* the task definition */
public class MY_TASK extends TASK {
  public void run (Object param) {
    MY_DATA data = (MY_DATA)param;
    data.select(criterion);
    data.print(out); data.add(value); } }

/* the task parameter */
public class MY_DATA {
  public void add(...) { ... }
  public void remove(...) { ... }
  public void print(...) { ... }
  public void select(...) { ... } }

public class SIMPLE_PARALLEL {
  public static void main (String argv[ ]) {
    /* task and data array initializations */
    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); }
    /* parallel activation of tasks */
    PAR par = new PAR (tasks,data); par.call (); } }
```

Fig. 1. A simple parallel program

3 Distributed Framework

The distributed framework [9] is used as a target of our preprocessor to distribute parallel programs expressed with our parallel framework. In the parallel framework, we use collections to manage the storage and accesses to active and passive objects. The distributed framework is based on distributed collections: a distributed collection is a collection that manages elements 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 in parallel by remote asynchronous invocations. 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; the distributed collection processes an access to a distributed element by remote invocation to the fragment *owning* this element (local to this element). 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 fragment. The task of converting a global identifier into the corresponding local identifier and the owner identifier devolves on a LAYOUT_MANAGER object. Different distribution policies are provided by different types of layout_managers. The program distribution is guided by the user choice of a specific layout_manager implementation.

Nevertheless the distributed framework does not manage the remote creations and accesses. They are handled by a specific runtime, and require to transform objects of the program (section 4).

4 Distributed Code Generation

We have developed a preprocessor to transform parallel programs expressed with our parallel framework into distributed programs, using our distributed framework. The parallel and distributed frameworks have the same interface, so the framework replacement is obtained by changing the imports in the program. Despite this, we have to transform some objects of the program: the objects stored in the distributed collections are distributed upon processors and need to have access to other objects. So, we have to:

- create (map) objects of the program on processors: when an object is located on a processor, its attributes are managed by the local memory and its methods run on this processor;
- have a mechanism allowing objects to be accessed remotely without any change from the caller point of view.

We have extended the object creation semantics to take into account remote creations of objects. Servers running on each host as separate threads are responsible for remote object creations. This extension is implemented using the standard Java reflection mechanism. To allow the transparent accesses to remote objects, the *Do!* preprocessor transforms a class into two classes:

- the *implementation class* contains the source methods implementations. An implementation object is not replicated and is located where the source object has been created (mapped). It is shared between all proxy objects.
- the *proxy class* has the same name and interface as the source class, but the method bodies consist in remote invocations of the corresponding methods in the implementation class. The proxy object handles a remote 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 each processor getting a reference on the source object.

5 Related Work

Many research projects have appeared around the Java language, aiming at filling the gaps of parallelism and distribution in Java. Some of them are presented in [1]. Some projects are based on parallel extensions to the Java language: tools [2, 4] produce Java parallel (multi-threaded) programs, relying on a standard Java runtime system using thread libraries and synchronization primitives; they do not generate distributed programs; others [7, 10] are based on distributed objects and remote method invocations. Other projects use the Java language without any extension: some environments [5] rely on a data-parallel programming model and a SPMD execution model; as in the *Do!* project, parallelism may be introduced through the notion of active objects [3].

6 Conclusion

In this paper, we have presented an overview of the *Do!* project, that aims at automatic generation of distributed programs from parallel programs using the Java language. We use the Java RMI as run-time for remote accesses; a foreseen extension of this work is to use CORBA for objects communications. Ongoing work consists in extending the parallel and distributed frameworks to handle dynamic creation of tasks, through dynamic collections (e.g. lists) and distributed scheduling.

References

1. ACM 1997 Workshop on Java for Science and Engineering Computation. *Concurrency: Practice and Experience*, 9(6):413–674, June 1997.
2. A. J. C. Bik and D. B. Gannon. Exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, 1997.
3. D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.
4. Y. Ichisugi and Y. Roudier. Integrating data-parallel and reactive constructs into Java. In *OBPDC'97*, France, October 1997.
5. 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.
6. 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.
7. L. V. Kalé, M. Bhandarkar, and T. Wilmarth. Design and implementation of Parallel Java with a global object space. In *Conference on Parallel and Distributed Processing Technology and Applications*, Las Vegas, Nevada, July 1997.
8. P. Launay and J.-L. Pazat. A framework for parallel programming in Java. In *HPCN'98*, LNCS, Springer Verlag, Amsterdam, April 1998. To appear.
9. P. Launay and J.-L. Pazat. Generation of distributed parallel Java programs. Technical Report 1171, Irisa, February 1998.
10. M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. In *PPoPP*, June 1997.