A Framework for Parallel Programming in JAVA — Source link

Pascale Launay, Jean-Louis Pazat

Related papers:

- JavaParty – transparent remote objects in Java

- Java/DSM: A platform for heterogeneous computing

- Towards seamless computing and metacomputing in Java

- cJVM: a single system image of a JVM on a cluster

- The Java Language Specification

A Framework for Parallel Programming in Java

Pascale Launay, Jean-Louis Pazat

A Framework for Parallel Programming in Java

Pascale Launay and Jean-Louis Pazat

No 3319

December 1997

THÈME 1

Rapport de recherche

A Framework for Parallel Programming in Java

Pascale Launay* and Jean-Louis Pazat[†]

Thème 1 — Réseaux et systèmes
Projet Pampa

Abstract: To ease the task of programming parallel and distributed applications, the *Do!* project aims at the automatic generation of distributed code from multi-threaded Java programs. We provide a parallel programming model, embedded in a *framework* that constraints parallelism without any extension to the Java language. This framework is described here and is used as a basis to generate distributed programs.

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

(Résumé : tsvp)

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

Un framework pour la programmation parallèle en Java

Résumé : Pour faciliter la programmation d'applications parallèles et distribuées, le projet Do! concerne la génération automatique de code réparti à partir de code Java parallèle. Le modèle de programmation parallèle est exprimé par un framework, qui nous permet de limiter l'expression du parallélisme sans extension au langage Java. Ce framework est décrit ici, et nous l'utilisons pour générer des programmes distribués.

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

1 Introduction

Many applications ranging from scientific computing and data mining to interactive and virtual reality applications need powerful computer resources and have to cope with parallelism, distribution, heterogeneity and reconfigurability. The main targets of these parallel and distributed 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, we can no longer restrict the target of programming tools to Fortran.

Object-oriented languages are widely used in many areas of computing and provide a practical solution for embedding application domain programming models into *frameworks* [4] easing the work of programmers. These languages offer a convenient way to separate design from implementation decisions by efficiently encapsulating implementation details into classes but solutions used to ease the programming of efficient and reliable sequential applications do not directly fit with parallel and distributed requirements.

Our aim is to ease the task of programming parallel and distributed applications using object-oriented languages (namely Java).

2 Overview of the Do ! project

The aim of the *Do!* project consists in automatic synthesis of distributed memory programs from shared memory (multi-threaded) programs, with user annotations for distribution of objects. We do not intend to hide completely parallelism and distribution as it was the case in High Performance Fortran.

Usually the programmer has some knowledge about the parallelism that can be exploited in his application. Using a pure sequential programming style would induce the sequentialization of parallel tasks of the application, thus complicating the code and loosing information about the application. This is why our programming model is explicitly parallel. Parallelism can be expressed on tasks or on data both considered as objects, and constraints have been defined to ensure that the parallel programs will be manageable by our tool (analysis, transformations, distribution). The parallel framework is the main topic of this paper ; parallel constructs based on generic COLLECTIONs of objects are described in section 3.

The programmer may also have some hints about the distribution of some objects of his program whereas he does not want to deal with many other aspects of distribution that are not related to the behavior of his application. For example data may have a physical location in a database or in a geographically distributed system, tasks may be mapped on nodes according to specific hardware requirements (for example a user interface that needs to run on a visualization console). In some cases, the programmer may need to control the distribution of data or tasks to improve the performance of his application. In order to allow users to describe the distribution of some objects without loosing the ease of non distributed parallel programming, the *Do!* programming model is not explicitly distributed :

the distribution of objects is guided by the distribution of COLLECTIONs that the programmer controls. We give an overview of this part of the project in section 4.

In this project we use the Java environment because it offers communication APIs (socket communications, RMI), and thread and synchronization mechanisms : this allows to write portable parallel and distributed programs. Because Java does not offer an easy parallel programming model, we have defined a structured programming model and an execution model which are implemented in Java by frameworks :

- the *Do!* parallel programming model is embedded into Java by a framework allowing us to constraint the expression of parallelism. This framework provides a model for parallel programming and a library of generic classes. The *non-expert* programmer needs only to extend some classes relevant to his application (TASK for example) while the *advanced* programmer can define new framework classes to make a better tuning of his application.

- a distributed framework, using distributed COLLECTIONs, is used to express the generated distributed memory programs. Distant objects communicate using the Java RMI (Remote Method Invocation).

The transformation from a shared memory program into a distributed memory program is not only obtained by changing the framework classes used by the program but also through the transformation of some classes of the program in order to be able to use Java remote objects.

3 Parallel framework

Our parallel programming model is embedded in Java by a framework without any extension to the Java language. 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 which are the pieces of code dedicated to his application.

The parallel framework that we have defined is based on active objects (TASKs) and on a parallel construct (PAR) that allows to execute COLLECTIONs of tasks in parallel. Tasks communicate only through shared passive objects (data) passed as parameters ; in the current implementation synchronizations only occur when tasks terminate.

In the following, we first introduce active objects (TASKs), then we describe COLLECTIONs, that we use as tasks or data containers. In order to express the processing of operations over COLLECTIONs, we use the operator design pattern which is described in 3.3. The parallel framework itself is presented in the last part of this section.

```
public class TASK {

    /* the task behavior */
    protected void run (Object param) { }

    /* synchronous invocation of run */
    public final void call (Object param) { ... }

    /* asynchronous invocation of run */
    public final void start (Object param) { ... }

    /* synchronization with the task termination */
    public final void join () { ... }

}
```

Figure 1: The TASK class

3.1 Tasks

The class TASK (figure 1) provides us with a model of task. Similarly to the Java THREADs[1], the default behavior of tasks is defined in the *run* method of this class and a user task is defined by extending TASK ; its behavior is inherited from the *run* method of TASK, that can be re-defined to implement the task specific behavior.

TASKs are activated by invoking their *run* method and are active during the whole execution of their *run* method. Task activation can be synchronous if one invokes the *call* method ; in this case the caller is blocked until the TASK terminates and there is no parallelism. In order to execute tasks in parallel, one must use the asynchronous task activation through the invocation of the *start* method : in that case, the caller resumes its activity immediately after the invocation . Synchronization with the TASK termination is provided by the *join* method. This asynchronous invocation is used for parallel execution of tasks.

TASKs are implemented using Java THREADS, but contrarely to a Java THREADS, a TASK can be activated more than once.

3.2 Collections

Tasks only provide us with a basic mechanism for parallelism, but we still need a structured parallel programming model. We think that parallelism must be more constrained and structured to be usable. This is why we express our programming model with COLLECTIONs.

A COLLECTION is an object that manages a set of elements and provides methods to retrieve and insert items or sequences of items. The COLLECTION class (figure 2) is a collection abstraction. Any kind of COLLECTION (for example LIST or TREE) can be defined

---

[1] Threads are used to implement Tasks

```
abstract public class COLLECTION {

    /* provides an iterator to go across the collection */
    abstract public ITERATOR items();

    /* returns the element associated to k */
    abstract public Object item (KEY k);

    /* adds obj to the collection with the specified key */
    abstract public void add (Object obj, KEY k);

}
```

Figure 2: The COLLECTION class

by extending COLLECTION. COLLECTIONs are generic[2] : their elements are of any type. We consider two kinds of COLLECTIONs :

- tasks COLLECTIONs : these collections are defined by instantiation of their elements type with the class TASK ;

- data COLLECTIONs : elements of data collections are of any type. A data COLLECTION is used to group parameters that are passed to tasks of a COLLECTION.

Our framework implements parallelism by the traversal of two COLLECTIONs (one contains the tasks and the other contains data objects). This traversal consists in the parallel (asynchronous) activation of the TASKs of the first COLLECTION with corresponding objects in the second COLLECTION passed as parameters to tasks. This framework is based on the operators design pattern that provides a model to express a processing over a data COLLECTION as an entity independent from the COLLECTION itself.

3.3 Operators framework

Jézequel and Pacherie [10] have defined the operators design pattern (figure 3) within the Eiffel language to design regular operations over large data COLLECTIONs. COLLECTIONs manage sets of data ; OPERATORs are autonomous agents processing data. The connection between OPERATORs and COLLECTIONs is provided by ITERATORs. The class OPERATOR (figure 4) represents a model of operator ; a CROSS is a specialization of an operator, representing a computation over two COLLECTIONs.

In the operators design pattern, one can express data parallelism through the distribution of data COLLECTIONs over processors and the processing of a regular operation concurrently

---

[2]The object-oriented notion of *genericity* does not exist in the Java language, so we have decided to use *casting* to implement it. This is one of the lacks of this language, but we could also use a Java extension supporting genericity [14].
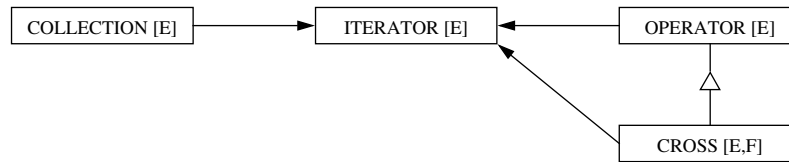
Figure 3: Operators framework

over each sub-collection. The distributed program relies on the SPMD (Single Program Multiple Data) model. The mapping of computations over processors is guided by the data distribution, following the *owner compute rule* : the processor on which an object is mapped is the only processor that can modify (write) the state of the object.

```
public class OPERATOR {

    /* constructor
       c - the collection to process */
    public OPERATOR (COLLECTION c)
    { it = c.items(); }

    /* the operation to process over the collection
       obj - an item of the collection */
    public void run (Object obj) { }

    /* runs the operation over the collection */
    public final void doAll()
    { for (it.start();!it.exhausted();it.next())
        { run(it.item()); } }

    /* the iterator used to provide items */
    protected ITERATOR it;

}
```

Figure 4: The OPERATOR class

3.4 Parallel framework

The operators design pattern has been used to implement parallel computations over large data structures (data parallelism). Including the concept of active objects, we have extended this framework to offer a parallel programming model including control (task) parallelism : task parallelism is a processing over a tasks COLLECTION (section 3.2), tasks parameters being grouped in a data COLLECTION.

```
public class START extends CROSS {

    /* constructor
       c1 - tasks collection ; c2 - data collection */
    public START (COLLECTION c1, COLLECTION c2)
    { super(c1,c2); }

    /* operation to process over tasks and data collections
       obj1 - the task to activate ; obj2 - the argument for the task */
    public void run (Object obj1, Object obj2)
    { ((TASK)obj1).start(obj2); }

}
```

Figure 5: The START class

The parallel activation of TASKs is realized by a CROSS object processing asynchronous invocations (section 3.1) of the TASKs *run* methods, with data corresponding as arguments. This is defined in the START class (figure 5) : this class extends the CROSS class, overriding its *run* method to describe the START specific operation.
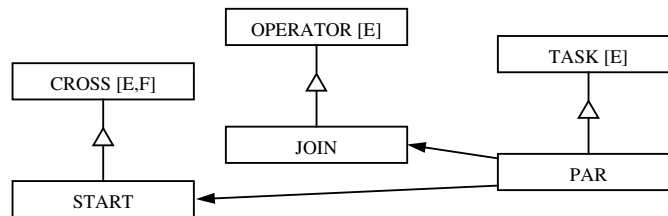

Figure 6: Parallel framework

To synchronize the TASKs at their termination, another operation is processed over the tasks COLLECTION, invoking the TASKs *join* method : this is defined in the JOIN class, and implements a global synchronization when all TASKs *run* methods are finished.

The class PAR implements those computations over tasks and data COLLECTIONs : this class is the only one the programmer has to deal with ; given two COLLECTIONs (of tasks and of data), it creates the START and JOIN objects, and realizes the parallel activation of TASKs with their corresponding arguments, and the final synchronization.

The class PAR extends the class TASK, so it is an active object, and can be included in a tasks COLLECTION. This provides a way to express **nested parallelism** : a parallel task (a instance of PAR) can be activated in parallel with other TASKs. The parallel framework is represented in the figure 6.

The 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 argument.

```
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 parallel program

4   Using the parallel framework for distribution

The parallel framework not only allows to built shared memory multi-threaded programs but is also used to generate distributed memory programs.

The distribution features are embedded in classes of the framework, without changing their interface, so the shared memory and the distributed memory programs are nearly the same. We only need to modify the imports in the program to use the distributed framework classes instead of the parallel one, and to transform some classes of the program to use the Java RMI. The source and generated programs have the same semantics : executing a parallel program in a shared memory or a distributed memory environment gives the same results.

The distributed framework relies on distributed COLLECTIONs that manage distributed elements transparently. A distributed collection is composed of sub-collections mapped on distinct processors, each sub-collection managing the local items of the COLLECTION. The distribution of a COLLECTION is defined by a distribution layout manager, that associates a processor to an element.

The execution model is based on remote creation of objects and remote method invocation. The remote creation of objects have been implemented using the reflection mechanism [8] and servers running on each processor. Communication between distant objects is implemented with the Java RMI (Remote Method Invocation) package.

To preserve the local parameter passing semantics (in local method invocations, arguments are always passed by reference), we transform objects into *remote objects* (because only *remote objects* are passed by reference in the Java RMI). The programmer annotates the objects that may be accessed remotely : classes that implement the interface ACCESSIBLE are automatically transformed into *remote objects* by the *Do!* preprocessor, developed with the parser generator Javacc [16] and the tree building preprocessor JJTree. The transformation is transparent : the generated objects have the same interface, especially concerning the exceptions handling[3].

5 Related work

Extensions to the language :

some projects are based on parallel extensions to the Java language.

Tools 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 :

- the High Performance Java project at Indiana University comprises the development of two different tools : JAVAR [2] and JAVAB [1]. JAVAR is a restructuring compiler that relies on annotations on a sequential Java program for generating a parallel program. JAVAB is a tool that automatically detects and exploits implicit loop parallelism in bytecode. JAVAR techniques could be used to build a preprocessor for our tool to help programmers to write parallel programs.

- Roudier and Ichisugi [6] have defined Tiny Data-Parallel Java as an example application of their extensible Java preprocessor EPP. In the Tiny Data-Parallel Java language, some methods are identified as data-parallel methods and translated by EPP in Java multi-threaded codes, executed on a large number of virtual processors. Their approach is designed to provide a general extension mechanism for Java.

Other are based on distributed objects, and remote method invocations :

- Philippsen and al [15] have extended the Java language by adding *remote* objects at the language level (*remote* is a class modifier). Calls to a specific runtime allows to create, distribute and move (migrate) objects. The programming model of JavaParty is explicitly distributed but remote method invocation is transparent from the programmer point of view. Compared to our approach, JavaParty could probably be used as a higher level runtime for *Do!* programs than the Java RMI and Thread packages.

- 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

---

[3]The attributes of a remote object are not directly accessible from other objects. We have not addressed this problem, because we think that the internal state of an object should not be seen nor modified without using an object's method. This approach is followed by many object oriented languages (for example Eiffel)

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.

Libraries and frameworks :

other projects use the Java language without any extension.

Some environments rely on a data-parallel programming model and a SPMD execution model :

- Ivannikov and al [7] 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. Contrarily to our tool, the distributed code uses a SPMD model.

- EPEE (Eiffel Parallel Execution Environment) [9] 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. We have also built a prototype using Java and the JGL (Java Generic Library) based on the ideas of EPEE.

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]. The synchronization model is less constraint than in the *Do!* project.

Doug Lea [13] provides design principles to build concurrent applications using the Java parallel programming facilities.

Also note that there exists PVM, MPI and other communication libraries interfaces for the Java language. These interfaces may be convenient to build alternate efficient run-times instead of using Java RMI.

6 Conclusion

In this paper we have presented an expressive parallel programming model, implemented by a *framework* in the Java language. We have not made any extension to the language. The synchronization model is very simple and will be extended in order to enlarge the application domain of our programming model. Compared to more traditional approaches (e.g. High Performance Fortran), this way to build a programming model and to implement it is much more efficient.

The distribution of *Do!* programs on possibly heterogeneous NOWS can be derived from user annotations for the distribution of COLLECTIONs. The distribution is obtained by changing the framework classes used by the program and transforming classes to use the Java RMI transparently ; no special compiler is needed. A simple preprocessor has been developed to transform objects into *remote objects* using Javacc and Javatree, and a mechanism allows remote creations of objects. Our parallel and distributed framework currently includes ARRAYS, and classical HPF-like [5] distribution indications. We plan to study other types of COLLECTIONs, and their convenience to express applications.

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] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[5] High Performance Fortran Forum. High Performance Fortran language specification. version 2.0. Technical report, Rice University, January 1997.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[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] D. Lea. *Concurrent Programming in Java. Design principles and patterns.* The Java Series. Addison-Wesley, 1996. ISBN 0-201-69581-2.

[14] Martin Odersky and Philip Wadler. Pizza into Java : translating theory into practice. In *24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

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

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