A new estimation method for distributed Java object activity

A. BOUCHI, R. OLEJNIK, B. TOURSEL*
LIFL - UPRESA CNRS 802
Laboratoire d’Informatique Fondamentale de Lille
* Ecole Universitaire D’Ingénieurs de Lille
Université des Sciences et Technologies de Lille
59655 Villeneuve d’Ascq Cédex
France
Phone : 33 (0)3 20 43 45 39 - Fax: 33 (0)3 20 43 65 66
http://www.lifl.fr/
bouchi@lifl.fr, olejnik@lifl.fr, toursel@lifl.fr

Abstract

We introduce a new method to estimate the Java object activity in a distributed context of irregular applications. This method uses an observation mechanism which is itself a part of a more global load balancing system. It predicts the tendencies of the communication between the objects of the distributed application. To illustrate the behaviour of this mechanism we applied it to the “Travelling Salesman Problem” TSP which is solved by means of a genetic algorithm. Finally we present the overhead measurements we have done.

1 Introduction

The distributed object platform and the use of wide-area network for intensive computing become a very important research and development area at international level. The arrival of large-scale oriented object systems allows many studies in that field: JavaNow, Globus project [6], Do! [7] for the load balancing and distribution aspects, JavaParty [9] and ACADA [10] for the mobility and adaptativity aspects.

The project ADAJ (Adaptive Distributed Application in Java) attempts to provide some answers for designing applications and efficient executions concerning distributed treatments on the network. ADAJ works in the context of distributed object systems built around Java/RMI. The Java language gives some interesting features to the distribution management, heterogeneity and security. The Java compiler provides an intermediate code handled by the virtual machine (Java Virtual Machine or JVM). The JVM runs on the majority of workstations and PCs. The pair formed by the language Java and the virtual machine JVM can be considered as one of the key platforms for large scale applications. This approach presents two major advantages. The first advantage is the large diffusion of this platform. The second advantage lies in the fact that all the Java virtual machines consist of a homogeneous base for the development of such applications.

A permanent well balanced object distributions requires to get continuously informations on load evolution and on dynamic relations between the objects. Therefore, ADAJ introduces a load balancing mechanism at the middleware level.

We have introduced in [3] a first approach of an observation mechanism of distributed object relations. This approach which we develop here allows one to acquire a knowledge of treatment behaviour on the execution platform.

The obtained predictions are used to compute the work of each object of the application and thus are used to determine a dynamic distribution of these objects on the computing platform. We don’t describe here the distribution mechanism. Finally, we rather interest us on the overhead which our observation mechanism introduces in the computing time.

2 Objects characterization

Therefore, ADAJ is an execution environment for distributed applications. ADAJ is composed of a set of JVMs: Java virtual machines. This environment enables to use the concept of distributed class introduced by JavaParty [9]; this
concept is based on the instanciation of the objects of the same class on different virtual machines. As these objects interact on each other, they must be declared “remote”. For load balancing reasons, and communication times optimization, this kind of object must be able to migrate. These objects are marked global. We have developed a precompiler, similar to that of JavaParty, which does this marking.

The applications treat also traditional java objects. We mark them as local, and they are neither remote, nor migratable.

The choice of the marked objects like global, (i.e. likely to migrate) is a delicate conception problem which is to be studied. For the moment, the tools necessary for the implementation of this marking are provided to the programmer.

3 The observation process

The execution of an irregular application 1 can generate an imbalanced load harming the effectiveness of the system. Moreover, in a multi-user context, the potential of calculation available for a given computer varies during the execution. Because of this problem, a strategy of adequate observation will be set up. This observation is based on two parts:

• Observation of the relations between the objects.

• Observation of the load on the various nodes and the network.

The mechanism of load balancing uses the information provided by these observations. The load measurement mechanism of the nodes is still under development and is described in [2]. We present in this paper the observation of the relations between objects.

In our particular context of irregular applications, the static analysis of the code is inefficient because of the dynamic relations of objects. So only the observation during the execution can provide relevant information to the redistribution.

The information is collected by an observer. A new distribution, which does not generate a significant overhead of communications, have to be defined from this information. Only the access to the global objects generates distant communications. Therefore, only these objects will be observed. The communications between objects in Java are realized through method invocations. Thus, the method invocations must be observed. Counting these calls is sufficient. A finer observation such as the measurement of the execution method time will generate a prohibitory cost of observation. However, the invocations on the global objects, potentially distant, and the invocations on the local objects (local activations) are to be distinguished. Doing that, we can balance the importance of the distant communications with the local activity of an object. Each global object holds three counters of invocations: the first is a table containing a counter for each global object related to it; the second counter is for the local invocations (called local objects); and the third counter is for invocations of its own methods.

The counting of the communications cannot be a simple accumulation. To fix a new distribution, we need more information from the recent past. So we introduced a smoothing information mechanism. Moreover, to avoid too abrupt variations and micro-phenomena, this mechanism must absolutely be implemented. The observer periodically reads, resets to zero the counters of invocations, and balances the recent information with older informations.

4 The object observation mechanism

ADAJ observes the global objects by counting the number of method invocations on these objects. This information will be used to predict the tendencies of the relations between the global objects in the future, i.e. to be able to estimate if the communication between two objects is in an increasing, decreasing or stagnant phase.

Counting is made by specific instructions inserted in the bytecode by a post-compiler process. These instructions enable one to count the invocations between the global objects. More precisely, for each global object $o_j$, for a given period, several attributes are added to this object:

• A table of $(o_j, outputGlobalInvocations_{j})$ stores the number $outputGlobalInvocations_{j}$ of method invocations of global object $o_j$.

• A counter $outputLocalInvocations$ stores the number of invocations carry out towards local objects.

• A counter $inputInvocations$ stores the total number of method invocations of the observed object $o_i$.

The observer inspects regularly the counters of activations, reads their values and resets them to zero. The measurement smoothing is done during the inspection by balancing the far past and the recent past. More precisely, the operations carried out for the counter $outputGlobalInvocations$ of the relation of $o_i$ to $o_j$ are:

• for each method invocation of $o_j$ :

\[
outputGlobalInvocations_{j} + +
\]

• for each inspection $k$, this information is smoothed:

\[
outputGlobalInvocations_{j} = \frac{outputGlobalInvocations_{j}^{k-1} + \sum_{i=0}^{k} \Delta outputGlobalInvocations_{j}}{k}
\]
\[ \text{scan}_k = \alpha \times \text{scan}_{k-1} + (1-\alpha) \times \text{outputGlobalInvocations}_j \]  
(2)

\[ \text{outputGlobalInvocations}_j = 0 \]  
(3)

where:

- \( \alpha \) is chosen in the range \([0, 1]\).
- \( \text{outputGlobalInvocations} \) is the number of method invocations between two inspections.
- \( \text{scan} \) is the smoothed value of the counter.

The observer will smooth in the same way, the other counters \( \text{outputLocalInvocations} \) and \( \text{inputInvocations} \).

5 The Java bytecode transformation

The marking of the migratable objects, therefore observable, is done implicitly; the marked objects are those which inherit from the \( \text{RemoteObject} \) class of JavaParty [9]. This heritage makes these objects remote.

The implementation of counters and instructions needed to increment these counters, is made by a post-compiler. This is done by using a transformation bytecode tool: JavaClass [5] which was supplied by the Free University of Berlin. We have used many kinds of transformations on the bytecode allowed by JavaClass: addition of new attributes and methods, method modifications by instruction adding, etc...

In the post-compilation of an application ADAJ, the bytecodes of all the classes are examined. The classes which inherit from \( \text{RemoteObject} \) are implicitly the observable classes.

For these particular classes three transformations are carried out by adding to them: the attributes for invocation counters, the instructions to increment these counters and the management methods of these counters which are:

1. For the counters, three attributes are added, two integer attributes, \( \text{outputLocalInvocations} \) and \( \text{inputInvocations} \), and a hashtable implements the table of \( (o_j, \text{outputGlobalInvocations}_j) \).
2. All the methods are analyzed. At the beginning of each method, the instructions to increment the attribute \( \text{inputInvocations} \) are added. For example the following instructions can be used:

\[
\begin{align*}
\text{aload}_0 & \quad \text{push } \text{this} \text{ onto the operand stack} \\
\text{dup} & \quad \text{duplicate } \text{this} \\
\text{getfield} & \quad \text{fetch the field } \text{inputInvocations} \\
\text{iconst}_1 & \quad \text{push } 1 \text{ onto the operand stack} \\
\text{iadd} & \quad \text{add } 1 \text{ to } \text{inputInvocations} \\
\text{putfield} & \quad \text{set the field } \text{inputInvocations}
\end{align*}
\]

The instructions of each method are examined to find the method invocation instructions, \( \text{invokevirtual}, \text{invokeinterface} \). For each found instruction two cases can arise: either it is a method invocation on a local object, or it is a method invocation on a global object. The signature of the called method, present in the bytecodes, allows one to choose between the two cases by using the method name and the method class.

In the case of a method invocation on a local object, inserting the instructions to increment the attribute \( \text{outputLocalInvocations} \) just before the invocation instruction is necessary. These instructions are similar to those which increment the attribute \( \text{inputInvocations} \).

In the opposite case, the increment of the couple \( (o_j, \text{outputglobalinvocations}_j) \) is necessary. So \( o_j \) has to be known. When the bytecode of the method invocation instruction is carried out, the stack contains the method parameters and the object on which the method is called, (i.e. \( o_j \)). Unfortunately these parameters are pushed on the stack before the object, making the access to these parameters difficult. In this case the signature of the called method is used, too. This enable the knowledge of the number, the type and thus the size of the pushed parameters. Inserting several series of instructions just before the invocation instruction is necessary by this way:

(a) instructions to pop from the stack all the parameters which are on the top and store them in local variables.
(b) an instruction to duplicate \( o_j \), which is now at the top of the stack; this duplication is made by an instruction \( \text{dup} \).
(c) instructions to call a management method of the hashtable: the method increments the suitable counter and is a method of the current object; thus we push the current object \( \text{this} \) with an instruction \( \text{aload}_0 \); the objects at the top of the stack are not in good order. So, they should be swapped with an instruction \( \text{swap} \); the management method is then called with an instruction \( \text{invokevirtual} \); this method has a parameter \( o_j \) which is popped from the stack;
(d) instructions to push the local variables, where the parameters were stored, with the aim of restoring the stack.

An example of the transformation of a method invocation on a global object could be:

The instruction Java:

\[
... o_j.\text{method1}(\text{"abc"},1.0); ...
\]

is compiled into the bytecode:

\[
\begin{align*}
\text{aload}_0 & \quad \text{push } \text{this} \text{ onto the operand stack} \\
\text{dup} & \quad \text{duplicate } \text{this} \\
\text{getfield} & \quad \text{fetch the field } \text{inputInvocations} \\
\text{iconst}_1 & \quad \text{push } 1 \text{ onto the operand stack} \\
\text{iadd} & \quad \text{add } 1 \text{ to } \text{inputInvocations} \\
\text{putfield} & \quad \text{set the field } \text{inputInvocations}
\end{align*}
\]
getfield fetch the field $o_j$
ldc push “abc” onto the stack
dconst_1 push 1.0 onto the stack
invokevirtual invoke method

which gives, after transformation (the added instructions are in bold):

getfield fetch the field $o_j$
ldc push “abc” reference onto the stack
dconst_1 push 1.0 onto the operand stack
dstore store 1.0 into local variable
astore store “abc” reference into local variable
dup duplicate $o_j$
aload_0 push this onto the operand stack
swap swapping
invokevirtual invoke a manage method with 1 para
aload push “abc” reference onto the stack
dload push 1.0 onto the operand stack
invokevirtual invoke method

With our assumption of an implementation in a hashtable, the management method verifies if its parameter $o_j$ is already in the table. In this case, the associated counter ($outputGlobalInvocations_j$) is incremented. In the opposite case, the couple ($o_j$, 1) is added to the table.

3. The last transformation consists of adding a management method for the hash table and possibly the necessary methods which allows the observer to access the counters.

6 Genetic algorithm observation

The evaluation of our mechanism is made through the observation of an evolutionary genetic algorithm [11] applied to the Travelling Saleman Problem [8]. A genetic algorithm is a heuristic algorithm in which an initial population evolves by pairwise crossing of the individuals according to the selection criterion of private individuals. Doing this repeatedly improves the quality of the individuals.

In the traveling salesman problem (TSP) a number of cities with distances between them is given and the task is to find the minimum-length closed tour that visits each city once and returns to its starting point.

6.1 Sequential algorithm

We first consider the Sequential version of the genetic algorithm to determine what object we can observed.

Forty cities and only one population of two hundred circuits are taken into account. In this application developed in Java, we have three types of classes:

- the class City : represents a city. The relations between the objects of this class form a graph of connections between the cities.
- the class Circuit : represents a Hamiltonian cycle of the graph which is a cycle that crosses every city of the graph exactly once. The cycle can be realizable. There exists in this case an effective connection between the crossed cities. In the opposite case, it is nonrealizable.
- the class Population : contains the circuits to be evolved

To carry out an observation of TSP algorithm, we choose to observe all of these three classes. Therefore we mark them like global classes. The observer periodically inspects these objects consulting the counters which we have assigned to each object. These counters represent the smoothed numbers of the communications between the objects.

![Figure 1. Relations between the objects of the sequential genetic algorithm.](image-url)

The results of the observation are illustrated in fig. 1. From the results of the observation, we can classify the objects of the application in three groups:

- the group of the objects of the class City : in our case, it contains 40 objects of this class. The state of all the 40 objects of this class doesn’t change during all the execution time of the program. They do not change during their life cycle. We thus mark them as local objects.
- the group of objects of the class Circuit : this group contains the objects of the type Circuit. At each evaluation, the algorithm creates 200 new circuits. They could be very numerous and their lifecycle is short. So, their observation costs very expensive. This Circuit class should thus not be observed, and should be retained as a local class.
the group of the objects of the class Population: this last group contains only one object of the Population type. It evolves during the whole execution and is in relation to all the objects of the group of the Circuit class. We may observe it in order to divide it in various subpopulations. We can consider it as a global object.

So the objects of the class Population are the only one which could be observed and marked as global object.

### 6.2 Distributed algorithm

We apply now the observation mechanism to a distributed version of the genetic algorithm described before. This version implements the “parallel model in islands” [4]. In this model, the subpopulations of circuits (colony$_i$, $i = 0, 1, 2, 3$, in fig. 2) evolve independently of each other. Sometimes, some circuits migrate towards other subpopulations, thus mixing the genetic features of these subpopulations.

As for the sequential case the $a$ parameter described in the fourth section is set to 0.5 and the inspection period to 20 seconds. And then we observed the relations between the objects.

This time, we considered 51 cities and 4 subpopulations containing 50 circuits each. Only two circuits at the same time can be transferred. The application contains 4 types of classes: the first three classes are exactly the same as in the sequential algorithm. The fourth one, the Colony class, contains a subpopulation and communicates with its neighbors by transferring two circuits.

We choose the Colony class as a global class. So, we only observe it. Four objects of the Colony class communicate in a ring manner with its neighbors. They are placed on four machines which have different powers.

The results obtained are presented in fig. 3. The number of communications is represented near the links between the objects.

![Figure 3. Relations between the objects of a parallel genetic algorithm.](image)

The communications of the colony colony$_0$ towards the colony colony$_1$ are clearly the most significant and the communication of the colony colony$_2$ towards the colony colony$_3$ are the weakest. This is due to a higher power and on a smaller load of the machine containing the colony colony$_0$, compared to the other machine containing the colony colony$_2$.

### 7 Overhead estimation

The system used for the first experiments described in this paper was composed by a cluster of Sun Sparc Workstation networked through Ethernet 100 Mega bits. The workstations are clocking at 300 MHz (RISC) and running Solaris.

One JVM per node is running and contains a global object colony and some other local objects.

First we launch the tests (the parallel genetic algorithm) 10 times. We compute the evolution time of each population then we compute the average time. Because the workstations we had used were not isolated from the LAN (some people could connected on the computers during the tests), we had to suppressed the worst cases from this average.

The fig. 4 presents the measurements which we have done in the conditions described below and after suppressed the worst cases. The first column of the figure represents the execution time with an inspection time fixed at 20 seconds. The inspection time for the second column is 10 seconds. And the last is relative to the execution time without observation.
We have computed the observation mechanism overhead as following:

\[
\text{overhead} = \frac{t_{\text{obs}} - t_{\text{wo}}}{t_{\text{wo}}}
\]  

(4)

where:

- \(t_{\text{obs}}\) is the time of the both genetic and observation algorithm execution.
- \(t_{\text{wo}}\) is the genetic algorithm execution time (without observation).

These experiments show a reasonable cost of about 5%.

\[
\begin{array}{cccccc}
\text{naïade} & \text{harpoon} & \text{lotus} & \text{ringnes} & \text{average} \\
\hline
\text{inspection time = 20 sec} & 800 & 600 & 700 & 900 & 800 \\
\text{inspection time = 10 sec} & 600 & 400 & 500 & 700 & 600 \\
\text{without observation} & 500 & 300 & 400 & 600 & 500 \\
\end{array}
\]

Figure 4. Observation overhead

8 Conclusion

In this article we have presented an observation mechanism of distributed objects in the context of irregular applications developed in distributed Java.

To realize the distribution and to rectify nonsatisfactory situations of execution, our execution environment ADAJ needs two types of information:

- observations inside the applications which is based on a method invocation counting which we have described here.
- load measurement of stations and network with a special instrument [1].

All the experiments we have made show the good behaviour of the observation mechanism presented in this paper. We have found that the observation tool depends on the period between two inspections and the weight given to a smoothing parameter (\(\alpha\)). The token solution will be improved by the introduction of adaptive periods of inspection according to the intensity of the communications.

The first measurements show only a weak overhead which we could still improved by using more specific code.

9 Acknowledgements

We would like to thank other members of the Paloma team for their collaboration.

References