

Sharing of variables among threads in heterogeneous grid systems by Aspect-Oriented Programming

Laurent Broto
UMR 5505 IRIT
118 route de Narbonne
F-31062 TOULOUSE
CEDEX4
broto@irit.fr

Jean-Paul Bahsoun
UMR 5505 IRIT
118 route de Narbonne
F-31062 TOULOUSE
CEDEX4
bahsoun@irit.fr

Robert Basmadjian
UMR 5505 IRIT
118 route de Narbonne
F-31062 TOULOUSE
CEDEX4
basmadji@irit.fr

ABSTRACT

Nowadays, the need for grid systems is becoming more and more relevant to applications demanding either large amount of data storage or computation. In this paper, we propose an approach that permits thread's distribution in heterogeneous grid systems without any middleware layer or thread's API modifications so that the programmer feels comfortable and shares variables in complete transparency. In order to achieve this task, we will use the Aspect-Oriented Programming and Java's mechanism such as introspection and RMI.

Keywords

grid, thread, aspect, AspectWerkz, java, scalability, introspection

1. INTRODUCTION

A grid system is the collection of either homogeneous or heterogeneous computers called nodes which are interconnected to each other in order to enhance the overall system capabilities. Amongst these capabilities, the followings are the three principal ones[1]: enhanced data storage by aggregating the storage spaces of each node, rapid execution of computation-demanding applications by taking advantage of the computing power of every node and the aggregation of the bandwidths of the nodes in order to obtain a larger bandwidth for every connection destined for the outside of the grid's domain.

On the other hand, the applications of such grid systems must be either compiled at every node of the grid or written in a language portable to every node of the grid.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

The realization of the first case is a tedious and costly task. Therefore, we have concentrated our efforts on the second case by taking Java as the programming language because of its multiplatform characteristic.

Nevertheless, in order that the applications written in Java benefit from the computing grids, they must be decomposable into pieces that can be migrated from one node to another. Java's threads cannot accomplish this task, so a more convenient solution must be found.

1.1 Contribution

We have implemented Java facilities that permit migration of the Java threads. This permits the application programmer to use grid systems in order to write simple, highly scalable and efficient programs. In this paper, we describe an approach called *MigThread* that permits the migration and execution of Java threads on different nodes of the grid and corresponds to the following specifications: sharing of variables among the threads, no modification of the JVM (Java Virtual Machine), Java's syntax and the program interface to which the programmer is accustomed. Finally, no deployment of additional application to the nodes other than the JVM of course.

In order to satisfy these specifications, we used the following mechanisms: adding of *advices* to certain portions of the user code thanks to the Aspect-Oriented Programming, Java's introspection mechanism which allows the program to discover the variables and the methods of a class from its name, Remote Method Invocation (RMI), grid file system's facilities which permit to access a Java application from any node and distant process execution by SSH.

We were forced to go back to the notion of process in order to realize efficient migration and then to return to the notion of threads (starting from these processes) in order to permit sharing of variables.

Our approach does not include the following functionalities: checkpointing, serialization, scheduling, fault tolerance and synchronization of Java threads. However, these functionalities will be included in upcoming releases.

1.2 The Plan

The rest of this paper is organized in the following manner: section 2 gives a quick overview of the approaches used to distribute threads in the grid system whereas section 3 presents Aspect-Oriented Programming as well as AspectWerkz. Section 4 presents our approach and its mechanism. Finally, the obtained results as well as the perspectives are discussed in section 5 and the conclusion is given in section 6.

2. STATE OF ART

Much work has been concentrated on migrating threads in grid systems. These are based on a middleware layer which provides grid services; examples of such a middleware layer are GTPE[6] (Grid Thread Programming Environment) of the University of Melbourne which is based on GridBus Broker[2] and PROACTIVE of INRIA[5]. Other works are based on distributing Java's virtual machine like the cJVM[3] approach. It is important to note that the middleware approach is based on modifying the syntax of the Java language whereas the second approach is based on modifying the virtual machine.

3. THE ASPECT ORIENTED PROGRAMMING

The computer programs are generally composed of two important perspectives: a functional perspective which permits the program to realize the things that it was devised for and the non-functional perspective which permits the program to be integrated into its environment or to respect certain specifications. Examples of non-functional perspectives are authentication or the event logging procedures. However, such perspectives are often blended in with the functional code, what makes their evolution and separation a very difficult task. These scattered pieces of code of the program are called *cross cutting concern*.

The aspect programming[4] permits to separate this non-functional code from the functional one when writing the program. There exists different approaches to achieve this task but we will quickly present only *AspectWerkz*[7].

3.1 AspectWerkz

An aspect contains a code called *advice* that will be executed at *pointcuts*. AspectWerkz proposes to define the pointcuts in an XML file and the advices in Java. The pointcuts are described by a logical language which permits to specify particular points in the code where the advices can be applied. These points can be, for example, method calls or reading and writing of variables.

For instance, the following files permit to display "the foo method will be called" every time the method foo of the application is called.

- the file that contains the advice

```
public class aspect_foo
{
    public void call_foo_method()
```

```
{
    System.out.println("the foo
        method will be called");
}
}
```

- and the file that describes the pointcut:

```
<aspectwerkz>
  <aspect class="aspect_foo">
    <pointcut
      name="foo_call"
      expression="execution(* *.foo
        (...))"
    />
    <advice
      name="before_foo_call"
      type="before"
      bind-to="call_foo_method"
    />
  </aspect>
</aspectwerkz>
```

AspectWerkz possesses a static and a dynamic weaver. In our approach, we used only the static one.

3.2 The Aspect Programming of our Approach

We are going to use the aspect programming in order to apply an advice before reading or writing the variables of the *run()* method. Thus, in our approach, we will be using a *get* or *set* pointcut types. Unlike AspectJ, AspectWerkz's API will allow us to discover the read or written variable. Then the advices will permit to update this variable before carrying out any operation related to it.

4. OUR APPROACH

Our approach is made up of two parts: migration of threads towards a distant node and the sharing of variables among the migrated threads.

4.1 Migration Towards a Distant Node

To achieve this task, a *MigProcess* class is created. Its main purpose is to realize a migration towards a distant node but does not allow any communication among the migrated tasks. Hence, we can consider these migrated tasks as processes. This class retrieves the API of JAVA threads but on the other hand modifies the signature of the *join()* method and makes it *final* so that neither it can be overridden nor it can be overloaded by the classes that inherit it. When the *start()* method is called, the task peels off in order to be executed on another node. When the *join()* method is called, the father process remains blocked until the execution of the distant process terminates. This method is used in order to synchronize the father with the child processes. We can notice in this class the existence of a *main()* method which allows it to become an executable class. This method will permit to resume the migrated process at the distant node.

In order to realize a migration, the following steps must be taken: separation of the migrating code from the principle application, migration towards the distant node, distant execution and synchronization between father and child processes.

In this paper, we will not delve into the details of these different steps. On the contrary, we consider that the threads are already distributed among the nodes and see how the variables can be shared among the threads.

4.2 The Principle of Variable Sharing Among Local Objects and Migrating Threads

4.2.1 Highlights of the Proposition

During the application's compiling phase, the programmer must call the AspectWerkz weaver which will weave the predefined aspects to his code. These aspects will play the role of capturing all read and write operations of the variables of the migrating thread's *run()* method. For each read and write operation, a call to a variable's address server is performed in order to know the address of the machine (thread) that holds the most recent value of the concerned variable. If this updated value is not found in the local migrating thread, it will be obtained by an RMI call to the thread that has the most recent value.

Hence the table of this variable's address server is updated. It is important to note the presence of a distributed thread context: not all updated values of the variables are found in the same migrating thread. Each migrating thread possesses a portion of the application's context and works locally as long as this portion is neither read nor written by another node of the grid. These mechanisms are included in the *MigThread* class which is the basic class of our approach and all migrating threads must inherit it.

4.2.2 Mechanism

As explained above, each migrating thread must inherit the *MigThread* class. Thus, each will behave like a Java thread but with a context distributed among several nodes of the grid. The *MigThread* class extends JVM's *UnicastRemoteObject* class which makes it an RMI component. Likewise, every instance of the *MigThread* class is associated with an instance of the *MigProcess* class. Finally, this class uses the *start()* and *join()* methods of the standard thread API.

When the *start()* method of the migrating thread is called, the *start()* method of the ancestor class *MigThread* is executed.

This method performs the following operations:

1. if it is the first migrating thread of the application, an RMI register is created at the local node of the application and an empty static hashtable is reserved (belonging to the *MigThread* class),
2. the *MigProcess* class associated with this instance of the class is started.

The migrating thread migrates to the distant node and the *main()* method of the *MigProcess* is called. This method executes the *pre_run()* method which creates an RMI register at the distant node and then executes the *run()* method of the migrating thread.

4.2.3 The Address Server

The server of the variable's address is encapsulated within the *MigThread* class. It consists of a static hashtable and an RMI register using the *CommWithBase* interface. The RMI register and the hashtable are created during the first execution of the distant thread. They remain on the machine that executes the Java application and the hashtable's *static* modifier permits it to remain unmodified even if all the migrating threads terminate their execution.

The *CommWithBase* interface defines the following method:

```
public String getAddress(String var,
                        String host, String name);
```

A call to this method is included in the advices which are bind to the code of the application in order to know the migrating thread that has the most recent value of the variable *var*.

The *host* and *name* parameters are used by the server in order to update the hashtable. In fact, we consider that a migrating thread that asks for the address of a variable, will obtain the most updated value. For each call to the *getAddress()* method, the table is updated.

This table is initially empty and as, during the migration, the context of the migrating threads is serialized, we consider that the first migrating thread that asks for a variable is also the one that possesses the most updated value.

4.3 The Migrating Threads

Every migrating thread carries with it an RMI server included in the superclass *MigThread*. This RMI server permits a thread to provide another thread a variable that will be updated in its context. Likewise, each thread is able to ask another thread for the value of a variable.

Therefore, the architecture for the exchange of variables is not centralized but is totally "point to point". On the other hand, the server of the variable can be a real bottleneck which imposes a client/server architecture to retrieve the addresses of the variables.

At the compilation time, the *run()* method of the migrating threads is modified by the AspectWerkz's preprocessor which captures all the read and write operations of the variables. When a read or write operation is called, the methods of *ExtendedInformation* interface are called by the advices.

4.3.1 The ExtendedInformation Interface

This interface is composed of the following two methods:

```
public void getVar(String variable);
public void setVar(String variable);
```

It permits the advices to update the variable that looks to be read or written.

The `getVar()` method, called during each read operation, sends a query to the address server in order to retrieve the address of the variable *variable*. If this variable is local, it stops at this point. If the variable is not local, its type is determined. A second query is sent to the thread possessing the updated value and then an update is performed locally. Thus the thread can perform the read operation.

The `setVar()` method, called during each write operation, sends only one query to the address server in order that this latter updates its address table. Thus the thread can perform the write operation.

4.3.2 The RMI Server of the Migrating Threads

This server works with the *Information* interface. This interface permits the threads to ask for the variables and possesses the methods of the following type:

```
// A method to retrieve an object called
// variable
public Object getObject(String variable)
    throws RemoteException,
    NoSuchFieldException,
    IllegalAccessException;

// A method to retrieve an integer
// called variable
public int getInt(String variable)
    throws RemoteException,
    NoSuchFieldException,
    IllegalAccessException

// The same for every type
```

When one of these methods is called, the asked variable is searched by the introspection mechanism using its name. When it is found, its name is sent to the thread that asked for it or an exception of *NoSuchFieldException* type is raised.

4.3.3 The Used Aspects

4.3.3.1 The advices.

The advices that we used are of two types: *before_reading* and *before_writing*. They recognize the name of the variable which will be read or written and then they call the `getVar()` or `setVar()` methods according to the case.

4.3.3.2 The pointcuts.

The pointcuts are dynamically generated before the weaving phase of the user. In fact, AspectWerkz can not determine the class from its superclass. Hence, we must find all the classes that extended the *MigThread* class and then for each of these classes the following type of pointcut must be created:

```
<pointcut name="write_AMigratingThread"
    expression="set(* *.AMigratingThread
        .*) AND withincode(* *.
        AMigratingThread.run())" />
<pointcut name="read_AMigratingThread"
    expression="get(* *.AMigratingThread
        .*) AND withincode(* *.
        AMigratingThread.run())" />
<!-- and so forth for all the classes
    that extend MigThread -->
```

for the *AMigratingThread* class.

The advices are attached to these pointcuts thanks to the following definitions:

```
<advice name="before\_reading" type="
    before" bind-to="
    reading_AMigratingThread" />
<advice name="before\_writing" type="
    before" bind-to="
    writing_AMigratingThread" />
<!-- and so forth for all defined
    pointcuts -->
```

If the inheritance manager of the AspectWerkz's pointcut language would have been integrated, only two pointcuts and two pointcuts-advice associations will have to be necessary.

4.4 Synthesis

Below is the flowchart of the read and write operations of a variable.

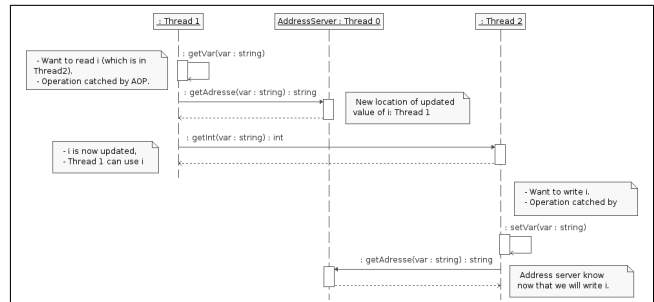


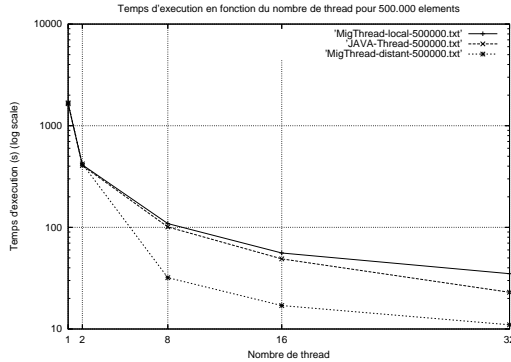
Figure 1: Flowchart of the read and write operations of a variable

5. RESULTS AND PERFORMANCES

We tested our approach by using a simple bubble-sorting algorithm on a grid composed of 4 machines each possessing dual processors. This sorting method is not the optimum one and this is acceptable because our main objective was to prove the validity of our approach. It executes on an array whose elements are inversed and functions in its worst case at $O(n^2)$.

We present the results in the form of a curve graph where the horizontal axis represents the number of used threads to sort an array of 500000 elements and the vertical axis represents the execution time to sort this array. On the diagram, we will find the curves for sorting the

array using Java threads, local MigThreads and distant MigThreads.



Here we can notice that the three curves overlap for a number of threads of 1 and 2. This is normal because the machines on which these tests were performed possess a double processor. We can equally notice the absence of any significant overhead by the MigThread approach using 1 or 2 processes. For a thread number of 8, the curves start to diverge: the portion of the curves of the Java thread and the local MigThread approaches have a similar variation whereas the distant MigThread approach gets benefit from distribution. The results of the distant MigThread approach are completely in conformity with the theoretical results of parallelism with 8 threads executing on 8 different processes. For 16 threads, both the distant MigThread and Java thread approaches obtain the same gain in performance. This is due to the progressive drop off of the complexity because the original array is partitioned into several sub-arrays. The local MigThread approach begins to become less efficient than the Java thread one. In fact, it is clear that these two approaches are both theoretically efficient but the overhead which is due to the network connection of the local loop starts to be a dominant factor in performance degradation. Finally, with 32 threads, local MigThread approach is less efficient than the Java thread approach. The distant MigThread approach varies in performance as the Java thread approach because the use of 8 processors masks away the problem of overhead due to the migration of the MigThreads.

We performed another test with 5000 elements. The results weren't good enough because of the migration overhead.

6. CONCLUSION

We have succeeded in distributing an application throughout a heterogeneous grid by respecting our constraints. Indeed, by using Java, the execution of applications in a heterogeneous grid becomes plausible. On the other hand, neither Java's parallel API nor its virtual machine are

modified and the application can be distributed among several nodes of the grid without deploying any application other than the JVM.

The obtained performances, by not taking into account the problem related to the migration time of the MigThreads through the network, are almost the same compared to the ones that could have been obtained if the machine on which threads were executed possess the same number of processors as the grid to which the MigThreads are migrated.

Nevertheless, the absence of synchronization makes coding the program a difficult task. Hence it is important that this synchronization be done in an explicit manner for example by means of a semaphore server. That is the reason why we are actually using AOP approach in order to synchronize the threads by using classical Java threads for synchronization.

Hence we will have a complete solution of distributing Java applications in a heterogeneous grids.

7. REFERENCES

- [1] V. Berstis. Fundamentals of grid computing. In *Redbooks*, 2002.
- [2] R. Buyya and S. Venugopal. The gridbus toolkit for service oriented grid and utility computing: An overview and status report. In *First IEEE International Workshop on Grid Economics and Business Models (GECON 2004, April 23, 2004, Seoul, Korea)*, 2004.
- [3] IBM. A scalable single system image vm for java on a cluster, 2000. <http://www.haifa.il.ibm.com/projects/systems/cjvm/overview.html>.
- [4] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [5] ProActive. Inria, 1999. <http://www-sop.inria.fr/oasis/ProActive>.
- [6] H. Soh, S. Haque, W. Liao, K. Nadiminti, and R. Buyya. Gtpe (grid thread programming environment). In *13th International Conference on Advanced Computing and Communications (ADCOM 2005)*, 2005.
- [7] WebSite. Aspectwerkz - plain java aop - overview, 2005. <http://aspectwerkz.codehaus.org/>.