.NET as a Platform for Implementing Concurrent Objects

Antonio J. Nebro, Enrique Alba, Francisco Luna, José M. Troya

Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga (SPAIN)
antonio,eat,troya@lcc.uma.es

Abstract. *JACO* is a Java-based runtime system designed to study techniques for implementing concurrent objects in distributed systems. The use of Java has allowed us to build a system that permits to combine heterogeneous networks of workstations and multiprocessors as a unique metacomputing system. An alternative to Java is Microsoft's .NET platform, that offers a software layer to execute programs written in different languages, including Java and C#, a new language specifically designed to exploit the full advantages of .NET. In this paper, we present our experiences in porting *JACO* to .NET. Our goal is to analyze how Java parallel code can be re-used in .NET. We study two alternatives. The first one is to use J#, the implementation of Java offered by .NET. The second one is to rewrite the Java code in C#, using the native .NET services. We conclude that porting JACO from Java to C# is not difficult, and that our sequential programs run faster in .NET than in Java, while internode communications have a higher cost in .NET.

1 Introduction

Concurrent object-oriented languages are characterized by combining concurrent programming and object-oriented programming. However, there is not a unique way to combine these two paradigms [1]. An alternative is to consider programs as collections of concurrent objects that communicate and synchronize by invoking the operations they define in their interfaces.

In the past, we have investigated implementing concurrent objects in parallel and distributed systems [2][3] in an efficient way. As a result, we have developed *JACO*, a runtime implemented in Java. With *JACO* (JAva based Concurrent Object System), we can use Java to write programs according to a concurrent object model. *JACO* offers services to concurrent object creation, object communication, and object replication plus migration.

The choice of Java to implement *JACO* is justified by the suitability of some Java features, such as object-orientation, multithreading support, socket based communication, heterogeneity, reflexion, and XML support. These features can be also found in the new language C# [4], which is implemented on top of Microsoft's .NET platform [5]. The resulting runtime system allows to combine

heterogeneous networks of workstations and multiprocessors as a unique metacomputing system. Both together, .NET and C#, appear as an alternative to Java. One of the most interesting features of .NET is that it is a multi-language platform. Whereas the Java Virtual Machine is bounded to only one language, the .NET runtime allows to execute programs written in C++, Visual Basic, C#, and even Java. The Java implementation on .NET is called J#, although it only provides the functionality of certain JDK classes, while some features, such as applets or RMI, are not supported. A drawback of .NET is that it is bound only to the Windows family of operating systems, although there are currently some initiatives to port .NET to other platforms [6]. Given that C# is similar to Java in many aspects, it is interesting to study whether parallels programs written in Java can be ported easily to this new environment. Furthermore, the availability of J# should allow us to execute Java programs directly on .NET.

In this paper, we present our experiences in porting *JACO* to .NET. We have developed a C# version of *JACO*, which is named $JACO^{C\#}$. The original Java version will be referred as $JACO^{J}$. After recompiling $JACO^{J}$ using J#, we have got a version named $JACO^{J\#}$.

The paper is organized as follows. In Section 2, we give an overview of the *JACO* runtime system. In Section 3, we compare the implementation of *JACO* in Java, J# and C#. In Section 4, we present results from preliminary experiments. Finally, we provide some conclusions in Section 5.

2 The *JACO* Runtime System

The object model assumed by *JACO* considers a concurrent object as an active entity, with an internal state and a public interface. Objects can also have synchronization constraints, which disable some operations when they are not allowed. There are two kinds of operations: *commands*, which are asynchronous operations, and *queries*, which are read-only synchronous operations. To ensure that a number of operations are executed in mutual exclusion, objects need to be acquired in shared or exclusive mode before being accessed.

Internally, *JACO* runs a process by node, containing the runtime system and the concurrent objects. The main components of the system are the object table, the object scheduler, and a communication agent for controlling internode communication. The *JACO* scheduler manages a pool of threads, and its mission is to assign an object ready to run to a thread. The object table contains references to object handlers, which are proxies used to access *JACO* objects.

3 Java versus .NET Implementations

In this section, we compare the three implementations of *JACO*. We begin with the description of $JACO^{J}$, and later we discuss $JACO^{J\#}$ and $JACO^{C\#}$.

As discussed in last section, $JACO^{J}$ uses threads to execute objets. Internode communication is carried out using TCP sockets. Each concurrent object in $JACO^{J}$ has an object identifier, which is a data structure containing an

identifier of the Java class of the object, among other information. Thus, given an object identifier, the runtime can create an instance of the object handler using the Java's reflexion mechanism. $JACO^J$ uses configuration files which contain network information. These are XML files, which are processed using the JAXP API of Java. Apart from these features, $JACO^J$ programs are pure Java programs. We do not use graphics, applets, nor RMI.

The simplest way to port $JACO^J$ to .NET is to use J#. In theory, we only have to recompile the Java code with the J# compiler. However, J# does not implement the JAXP API for XML processing, because this functionality is offered by the underlying .NET platform. So, a solution is to use the .NET XML services from J#. For this work, we took the simple approach of removing the XML code and including the information contained in the configuration files as constants objects in header files. After removing the XML code, $JACO^{J\#}$ compiled without problems. When running some $JACO$ applications (see Section 4), we found some problems. For example, the Java random objects (`java.util.Random`) did not work well in J#, but they can be due to the fact that we have used a beta version of J# (Visual J# .NET Beta 1). Anyway, the problem was solved easily by invoking the equivalent service (`System.Random`) offered by .NET.

The implementation of $JACO$ in C#, as well as the applications developed on top of it, required to rewrite all the Java code in C#. Syntactically, the two languages are similar: their object-orientation model is basically the same, and the .NET services to manage threads, sockets, XML, and object serialization are almost equivalent in C# and Java. Furthermore, we were able of maintaining the same package structure of the original Java code, by simply replacing packages by namespaces, so the translation was not a complicated task.

4 Performance Comparison

In this section, we present a performance comparison of the three $JACO$ implementations. The experiments we have carried out must be considered as preliminary ones, because we have used a beta version of .NET. Nevertheless, the results obtained show the current differences between Java and .NET, and they can give us an insight of what we can expect from the upcoming releases of .NET.

To measure performance, we have computed the cost of invoking object operations and analyzed two distributed applications, a branch and bound algorithm and a genetic algorithm [7]. The experiments were executed on a network of 6 PCs, each one having an Intel Pentium III 550MHz processor, 128MB of real memory, and a Fast Ethernet 100 Mbps adapter. They run Windows 2000 (SP2). We have used JDK 1.3.1-b24, and the Java programs were compiled with the `-O` optimization flag. The J# programs were compiled using Visual J# .NET Beta 1, on top of Visual Studio .NET Beta 2. We run the release version of J# programs on top of the CLR V1.0.2914. Finally, we compiled and run the C# programs using the Framework SDK .NET, CLR V1.0.3705. C# programs were compiled using the `/o` optimization flag.

Table 1. Costs of local and remote object communication (in ms)

JACO version	Java		J#		C#	
	Local	Remote	Local	Remote	Local	Remote
Command	0.007	3.6	0.005	40.3	0.005	13.3
Query	0.209	11.7	0.076	89.2	0.256	29.2

Table 2. Times (in sec) and speed-ups obtained with the branch and bound program

JACO version	Java	J#	C#
Sequential	756	722	737
Distributed (6 nodes)	134	384	202
Speed-up	5.6	1.8	3.6

Let us begin by measuring basic communication costs in *JACO*. In Table 1 we include the cost of invoking a command and a query operation on a concurrent `double` object. In the case of local communication, we observe that commands take a similar time, while queries in Java perform slightly better than in C#, but roughly tree times worst than in J#. However, remote communications in Java are significantly better.

In Table 2 we report the times and speed-ups obtained when running the distributed branch and bound algorithm to solve a 100-city instance of the Traveling Saleman Problem. This algorithm is characterized by a high degree of communication, needed to enhance the load balancing. The execution of the sequential program shows that the three versions yield comparable times, being faster the J# version. The speed-ups obtained in the parallel executions using 6 nodes are the consequence of the high cost of remote communication in J# and C#, which work out 1.8 and 3.6, respectively, while the speed-up in Java is 5.6.

The second application is a distributed genetic algorithm (DGA) that tries to optimize the following ONEMAX function: $f_{ONEMAX}(\boldsymbol{x}) = \sum_{i=1}^{n} x_i$. This algorithm is the *JACO* version of the one used in [8], which was implemented in Java using sockets and threads. The DGA program is characterized by a low computation/communication ratio, and its results are strongly dependent of the random number generator, because it uses stochastic operators. In Table 3 we report the results of running each version of the program on our 6 node network. Here, the execution time does not provide a complete measure of performance, because we observed that the optimum was found rapidly by the Java version, while it was hard to find with the J# and C# versions. Considering that the

Table 3. Results obtained with the distributed genetic algorithm

JACO version	Java		J#		C#	
	Local	Distributed	Local	Distributed	Local	Distributed
Time (in ms)	74.02	13.27	158.90	66.91	153.00	19.46
Evaluations	2,451	14,557	3,066	11,075	5,421	29,942

DGA is exactly the same in all the tests, the explanation has to do with the random number generator of .NET. However, an insight of performance can be obtained if we analyze the mean number of evaluations per second of the three programs: the C# version is roughly twice faster than the Java and J# versions.

5 Conclusions

In this paper we have presented our experiences in porting *JACO*, a Java-based runtime system for implementing concurrent objects, to .NET. We have used two .NET languages, J# and C#. We can conclude that Java parallel programs that use standard mechanisms, such as threads and sockets, can compile and run on .NET using J# with few problems, and that the similarities between Java and C# allow to rewrite Java programs in C# with little effort.

Preliminary performance results show that our .NET-based programs perform slightly better than the Java versions of the same programs in sequential execution, while there is a significant advantage for Java concerning remote communications. However, our tests using C# reveal a reduction in the communication time when comparing with the same tests using J#. Since C# programs use a more recent version of the Framework SDK .NET than J# programs (V.1.0.3705 versus V.1.0.2914), we can conclude that this issue is being improved by Microsoft. Despite C# and J# run over .NET, there are differences in the execution time of the programs. This can be due to the fact that the compilers are different, and probably they do not generate the same code.

.NET is recent, while Java JDK has been continuously improved for many years, so we can expect that future releases of .NET will allow distributed programs to run efficiently. A more exhaustive evaluation of *JACO*, including more applications on top of Java and .NET is a matter of future work.

References

1. Philippsen, M.: A survey of concurrent object-oriented languages. Concurrency: Practice and Experience **12** (2000) 917–980
2. Nebro, A.J., Pimentel, E., Troya, J.M.: Distributed objects: An approach based on replication and migration. The Journal of Object-Oriented Programming (JOOP) **12** (1999) 22–27
3. Nebro, A.J., Pimentel, E., Troya, J.M.: Integrating an entry consistency memory model and concurrent object-oriented programming. In: Third International Euro-Par Conference. (1997) Passau, Alemania.
4. Liberty, J.: Programming C#. O'Reilly (2001)
5. Platt, D.S., Ballinger, K.: Introducing Microsoft .NET. Microsoft Press (2001)
6. de Icaza, M.: The Mono Project: An Overview (2001) http://www.ximian.com/devzone/tech/mono.html.
7. UEA Calma Group: Parallelism in combinatorial optimisation. Technical report, School of Information Systems, University of East Anglia, Norwich, UK (1995)
8. Alba, E., Nebro, A.J., Troya, J.M.: Heterogeneous computing and parallel genetic algorithms. Accepted for publication in the Journal of Parallel and Distributed Computing (2002)