An Approach to Asynchronous Object-Oriented Parallel and Distributed Computing on Wide-Area Systems*

M. Di Santo[1], F. Frattolillo[1], W. Russo[2] and E. Zimeo[1]

[1] University of Sannio, School of Engineering, Benevento, Italy
[2] University of Calabria, DEIS, Rende (CS), Italy

Abstract. This paper presents a flexible and effective model for object-oriented parallel programming in both local and wide area contexts and its implementation as a Java package. Blending *remote evaluation* and *active messages*, our model permits programmers to express asynchronous, complex interactions, so overcoming some of the limitations of the models based on message passing and RPC and reducing communication costs.

1 Introduction

Exploiting geographically distributed systems as high-performance platforms for large-scale problem solving is becoming more and more attractive, owing to the high number of workstations and clusters of computers accessible via Internet and to the spreading in the scientific community of platform-independent languages, such as Java [14]. Unfortunately, the development of efficient, flexible and transparently usable distributed environments is difficult due to the necessity of satisfying new requirements and constraints. (1) *Host heterogeneity*: wide-area systems solve large-scale problems by using large and variable pools of computational resources, where hosts often run different operating systems on different hardware. (2) *Network heterogeneity*: wide-area systems are characterized by the presence of heterogeneous networks that often use a unifying protocol layer, such as TCP/IP. While this allows hosts in different networks to interoperate, it may limit the performance of specialized high-speed networking hardware, such as Myrinet [1]. (3) *Distributed code management*: a great number of hosts complicates the distributed management of both source and binary application code. (4) *Use of non-dedicated resources*: traditional message-passing models are too static in order to support the intrinsic variability of the pool of hosts used in wide-area systems. In this context, the interaction schemes based on one-sided communications are more apt, even if some of them, by adopting synchronous client/server models, do not ensure an efficient parallelization of programs.

Starting from these considerations, we propose a flexible and effective model for object-oriented parallel programming in both local- and wide-area contexts.

* Work carried out under the financial support of the M.U.R.S.T. in the framework of the project "Design Methodologies and Tools of High Performance Systems for Distributed Applications" (MOSAICO)

It is based on the *remote evaluation* [13] and *active messages* [3] models and overcomes some of the limitations of the models based on message passing and RPC, thanks to its completely asynchronous communications and to the capability of expressing complex interactions, which permit applications to reduce communication costs. Moreover, its ability of migrating application code on demand avoids the static distribution and management of application software.

The model has been integrated into a minimal, portable, efficient and flexible middleware infrastructure, called *Moka*, implemented as a Java library. *Moka* allows us both to directly write object-oriented, parallel and distributed applications and to implement higher-level programming systems. *Moka* applications are executed by a parallel abstract machine (*PAM*) built on top of a variable collection of heterogeneous computers communicating by means of a transparent, multi-protocol transport layer, able to exploit high-speed, local-area networks. The *PAM* appears as a logically fully-interconnected set of abstract nodes (*AN*), each one wrapping a Java Virtual Machine. Each physical computer may host more than one *AN*.

2 Related work

De facto standard environments for parallel programming on clusters of workstations are doubtless PVM [7] and MPI [12]. Both use an execution model based on processes that communicate by way of message passing, which offers good performances but supports only static communication patterns. Moreover both PVM and MPI are rather complex to be used by non-specialists and, on the wide-area scale, present the distributed code management problem. Java implementations of PVM [4] and MPI [8] simplify a little the use of message passing, especially for object-oriented parallel programming.

A different and more attractive approach to distributed and parallel computing is the one proposed by Nexus [5] and NexusJava [6]. These systems support fully asynchronous communications, multithreading and dynamic management of distributed resources in heterogeneous environments. The communication scheme is based on the use of global pointers (one-sided communication), which allow software to refer memory areas (Nexus) or objects (NexusJava) allocated in different address spaces. NexusJava is rather similar to *Moka* both in the programming model and in the architecture, but it has a too low-level and verbose programming interface and limits distributed interactions to the invocations of methods explicitly registered as handlers.

Commonly used middlewares based on Java RMI [14], generally, do not directly provide asynchronous mechanisms on the client site. Some recent systems, such as ARMI [11], transform synchronous RMI interactions into asynchronous ones by using either an explicit or an implicit multithreading approach. However, in both cases, inefficiencies due to the scheduling of fine grain threads are introduced, especially on commodity hardware. Instead, *Moka* provides more efficient, fully asynchronous communications at system level.

3 The *Moka* programming model

The model is based on the following three main concepts. (1) *Threads*, which represent the active entities of a computation. (2) *Global objects*, which, through a global naming mechanism, allow the applications to build a global space of objects used by the threads to communicate and synchronize themselves. (3) *Active objects* (auto-executable objects), which allow threads to interact with global objects, by using a one-sided asynchronous communication mechanism.

A *Moka* computation is fully asynchronous and evolves through subsequent changes of global objects' states and through their influence on the control of application threads. These have to be explicitly created and managed by the program, which must pay attention to protect objects against simultaneous accesses from the threads running on the same AN.

An active object (AO) can be asynchronously and reliably communicated, as a message, to an AN where it may arrive with an unlimited delay and without preserving the sending order. When an AO reaches its destination node, the execution of its handler (a function connected to the AO) is automatically carried out. The handler code, when not already present on the node, is dynamically retrieved and loaded by *Moka*. So it is possible to program according to a pure MIMD model, where applications are organized as collections of components, each one implementing a class used to build active or global objects, loaded on the nodes only when necessary (code on demand promoted by servers).

The automatic execution of handlers implies that a message is automatically and asynchronously received without using any explicit primitive. This semantics requires the existence of a specific activity (*network consumer*) devoted to the tasks of extracting active objects from the network and of executing their handlers. A solution is based on the *single-threaded upcall* model, in which a single network consumer serially runs in its execution environment the handlers of the received objects. On the other hand, when a handler may suspend on a condition to be satisfied only by the execution of another AO, in order to avoid the deadlock of the system, the program must use the *popup-threads* model and so explicitly ask for the handler to be executed by a separate, dedicated thread (*network consumer assistant*) [9].

Application threads and handlers interact by using a space of global objects (GOS). A global object (GO) is abstractly defined as a pair (GN, GO_IMPL). GN is the global name of the GO and univocally identifies it in the *Moka* system. GO_IMPL is the concrete GO representation, an instance of a user defined class physically allocated on an AN. A thread can access a GO_IMPL by making a query to the GOS, which is organized as a distributed collection of *name spaces*, each one (NS_i) allocated on a different AN. Therefore, in order to access a GO_IMPL, it is necessary to know its location (node i). *Moka* offers two ways in order to do this: (1) by means of a static, immutable association (GN, i), established at the GN creation time; (2) by using a dynamic approach, where the GO_IMPL location is explicitly specified at access time. In this latter case, one different implementation per node may be bound to a given GN; so, it

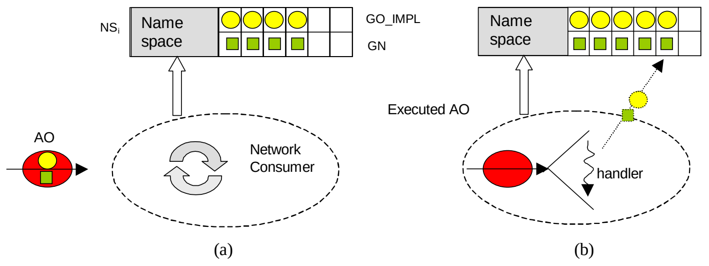


Fig. 1. Creation of a new GO. (a) An AO, containing a GO_IMPL and a GN, is arriving on an node. (b) The network consumer runs the AO handler, which creates the association (GN, GO_IMPL) in the NS_i

is possible to realize a replicated implementation of a GO, even if the program must explicitly ensure the consistency of replicas.

At the start of computation, only AO handlers can refer the local *namespace* (NS_i). Subsequently, the NS_i reference may be passed to other activities running on the same node. When GO_IMPLs become shared resources, they must be explicitly protected from simultaneous accesses. The creation of a GO on a node i requires three operations executed on the node where the request starts: (1) generate a GN; (2) create a specific AO containing both the GN and the GO_IMPL; (3) send the AO to the node i (see fig. 1), where its handler execution binds the GN to the GO_IMPL in the local namespace NS_i. An existent GO can be remotely accessed through the following operations: (1) reclaim from the GN the identifier of the node where the GO resides; (2) send to this node an AO which looks up NS_i with GN as key, in order to get the GO_IMPL; (3) execute on the GO_IMPL the operations abstractly requested on the GO.

Through a special form of active objects (AOc), *Moka* provides a deferred synchronous send primitive that calls for a result produced by the execution of operations tied to a dispatched AOc. Sending an AOc does not suspend the caller which immediately receives a *promise*, an object that will get the result in the future. A suspension will occur only if the result is reclaimed before it is really available. An AOc can be modified and forwarded many times before returning the result (agent like model); so the caller may receive the result of a complex distributed interaction.

4 The Java API of *Moka*

A Java package implements the proposed model, by offering an API that allows programs to create and dynamically configure the PAM, to create and send active objects, and to create global objects. Thread management and synchronization on global objects, are instead committed to the Java language default mechanisms. In fact, differently from the proposal in [2], we do not provide

Moka-level synchronization mechanisms, because we want *Moka* to be a minimal and extensible middleware.

The *Moka* package contains the following classes and interfaces: `Moka`, `ActiveObject`, `ActiveObjectCall`, `Promise`, `GlobalUid` and `LocalNameSpace`. The `Moka` class allows us to dynamically configure the *PAM* and provides programmers with primitives for either asynchronously or deferred synchronously sending active objects to nodes, by using either point-to-point or point-to-multi-point communication mechanisms, with the possibility to specify the single threaded upcall model (`Moka.SINGLE`) or the popup threads one (`Moka.POPUP`). It is worth noting that, when a synchronous send primitive is used, the result is to be caught by an instance of `Promise`. An active object is an instance of a user-defined class implementing one of the Java interfaces `ActiveObject` and `ActiveObjectCall`, to be respectively used for asynchronous and deferred synchronous interactions. Instances of `GlobalUid` and `LocalNameSpace` classes respectively implement the global name (GN) of a GO and the namespace of a node i (NS_i). In particular, *Moka* creates one `LocalNameSpace` instance per node, which is automatically passed to all the handlers. Moreover, the *Moka* API provides three classes of active objects: `Create`, `InvokeVoidMethod` and `InvokeMethod`, that, using the reflection Java package, allow programs to respectively create a global object, invoke on a GO a void method or a method returning a value.

For the sake of clarity, in the following, we present a simple program that multiplies in parallel two square matrices, A and B.

```
public class Main implements ActiveObject {
 public void handler(LocalNameSpace ns) {
  float[] a, b; int dim;
  <read matrices a and b as mono-dimensional arrays of dim*dim>;
  GlobalUid gn = new GlobalUid(); ns.bind(gn, new Matrix(b));
  Moka.broadcast(new Create(gn, Matrix.class, b), Moka.SINGLE);
  int numNodes = Moka.size(); int rfn = dim/numNodes;
  float[] subM = new float[dim*rfn];
  Promise[] result = new Promise[numNodes];
  for(int node = 0; node < numNodes; node++) {
   System.arraycopy(a, node*dim*rfn, subM, 0,rfn*dim);
   result[node] = Moka.call(new SubMatrix(gid,subM),node,Moka.SINGLE); }
  for(int node = 0; node < numNodes; node++) {
   float[] r = (float[])result[node].getValue(); <print r>; }
 }
}
public class Matrix implements Serializable {
 private float mat[];
 public Matrix(float[] m) { mat = m; }
 public float[] multiply(float[] a) { return < a x mat >; }
}
public class SubMatrix implements ActiveObjectCall {
 private float[] part; private GlobalUid gn;
 public SubMatrix(GlobalUid g, float[] a) {gid = g; part = a;}
 public Object handler(LocalNameSpace ns) {
  return ((Matrix)ns.lookUp(gn)).multiply(part);
 }
}
```

The algorithm is organized as follows: B is replicated on each node of the PAM, whereas A is dissevered into submatrices, each one formed by an equal number of rows and assigned to a different node of the PAM. The computation evolves through the parallel multiplication of each submatrix with B. In more detail, the program creates the PAM and sends a first (`Main`) AO to one of its nodes, where the handler replicates B as a global object by generating a global uid (`gn`) and wrapping B in an instance of the class `Matrix` (`og_impl`); locally, this is obtained by directly binding, in the local name space, `gn` to a local instance of `Matrix`; remotely, by broadcasting an AO of the class `Create`, which takes charge to create a remote `og_impl` and bind it to `gn` in the name space of the remote node. Moreover, the handler divides A into `dim/numNodes` submatrices, where `dim` is the dimension of A and `numNodes` is the size of the PAM, obtained by invoking the primitive `Moka.size`. Each submatrix is wrapped into an instance of the class `SubMatrix`, which implements the `ActiveObjectCall` interface, and sent to each node by using the deferred synchronous send primitive `Moka.call`; all these invocations immediately return a promise. Remotely, the handler of `SubMatrix` gets the local instance of `Matrix` and invokes its method `multiply`. The resulting values are caught by *Moka* and implicitly sent to the node that executed the main AO; here, they are extracted from the promise by using the blocking `Promise.getValue` primitive.

5 Transparent vs. non-transparent distributed interactions

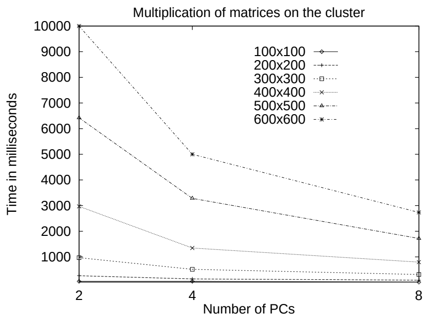
The *Moka* model is based on non-transparent, distributed interactions among threads and global objects, whereas the middlewares based on the RMI model make possible the transparent invocation of methods on remote objects. Unfortunately, in order to realize transparency, these systems use IDLs (I*nterface* D*escription* L*anguage*) and stub generators, which complicate the development of distributed applications, especially when many remote objects are used. In addition, using RMI, the interactions between different address spaces are limited to the invocations of remote object methods, explicitly exported by an interface. So, it is not possible to remotely invoke class methods, to directly access the public data members of an object and to dynamically create remote objects. *Moka* instead, thanks to the possibility of remotely executing all the locally executable operations, permits us to efficiently realize any kind of interactions among address spaces, without using stub generators and preserving a non transparent polymorphism between local and remote (global) objects. Moreover, our model does not require global objects to be instances of particular classes, as Java RMI does; instead any object can become a global one after its binding in a NS_i. In addition, the use of distributed interactions as first class objects allows

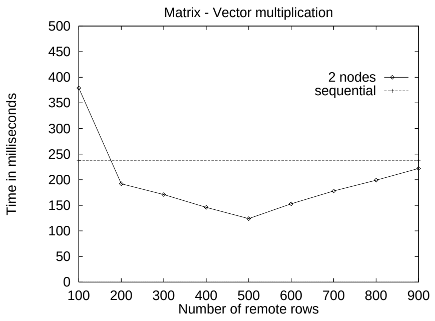
Moka to minimize communication costs when a distributed interaction implies the execution of methods whose results are used as arguments of other methods on the same *AN*. Therefore, we argue that while transparent distributed interactions allow programs to easily express remote invocations of methods, a non-transparent lower-level approach seems more efficient when object-oriented programming is used for high-performance parallel computing or when the interest is in the development of higher level systems.

6 Performance evaluation

At the present, *Moka* implements three transport modules, two respectively based on TCP and reliable UDP with multicast support, to be used on the Internet and local-area Ethernet networks, and the third, based on Illinois Fast Messages [10], to be used on Myrinet networks. All these protocols can be contemporary used by the *PAM*, because it is possible to specify a different protocol for each pair of nodes.

In the following we present two graphs. The first graph shows the performances of the presented matrix multiplication example on a cluster composed of bi-processor PCs (PentiumII 350MHz) interconnected by a multi-port Fast Ethernet repeater and using the TCP as communication transport. We can observe that, without taking into account the time necessary to transfer the right matrix, a good speedup is achieved for matrices over 400×400.





The second graph shows the time needed to compute the product between a 1000×1000 matrix and a vector on two Sun UltraSparcs interconnected by Ethernet, without taking into account the time to transfer the matrix. We can observe that the best performance is reached for a fifty-fifty splitting of the matrix rows on the two machines. Anywhere, the graph shows an absolute speedup even for the other, less favorable splittings of the matrix. The anomaly in the case of 100 remote rows is due to the need of initial remote code loading.

A third experiment was realized on a small wide-area system composed of a PentiumII 400MHz PC interconnected to 2 Sun UltraSparcs by means of a frame relay network (512 Kbps CIR, 2 Mbps CBIR). On this system, the product of two matrices shows a little speedup only for the dimension of 500×500 floats (21 sec. parallel, 33 sec. sequential). In fact, for smaller dimensions the grain is too small, while for larger ones the available bandwith is saturated.

7 Conclusions

We have shown how the integration of the *remote evaluation* model and the *active messages* one permits Java programs to set-up global objects spaces and, with the use of promises, to realize distributed asynchronous interactions that overcome some limitations of message passing and RPC. The proposed model has been integrated into the middleware *Moka* which, thanks to the use of a multi-protocol transport, ensures acceptable performances both in local- and wide-area networks. At this end, we will provide a better integration of default Java serialization with the FM libray in order to improve performances on Myrinet clusters, which currently are only slightly better than the ones provided by TCP clusters.

References

1. N. J. Boden et al.: Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29-36, 1995.
2. D. Caromel, W. Klauser, and J. Vayssiere: Towards Seamless Computing and Meta-computing in Java. *Concurrency: Pract.&Exp.*, 10(11-13):1043-1061, 1998.
3. T. von Eicken et al.: Active Messages: A Mechanism for Integrated Communication and Computation. *19th Ann. Int'l Symp. Computer Architectures*, *ACM Press*, NY, 256-266, 1992.
4. A. Ferrari: JPVM: Network Parallel Computing in Java. *ACM Workshop on Java for High-Performance Network Computing.* Palo Alto, 1998.
5. I. Foster, C. Kesselmann, and S. Tuecke: The Nexus Approach to Integrating Multithreading and Communication. *J. of Par. and Distr. Computing*, 37:70-82, 1996.
6. I. Foster, G. K. Thiruvathukal, and S. Tuecke. Technologies for Ubiquitous Supercomputing: A Java Interface to the Nexus Communication System. *Concurrency: Pract.&Exp.*, June 1997.
7. A. Geist et al.: *PVM: Parallel Virtual Machine.* The MIT Press, 1994.
8. V. Getov and S. Mintchev: *Towards a Portable Message Passing in Java.* http://perm.scsise.vmin.ac.uk/Publications/javaMPI.abstract.
9. K. Langendoen, R. Bhoedjang, an H. Bal: Models for Asynchronous Message handling. *IEEE Concurrency*, 28-38, April-June 1997.
10. S. Pakin, V. Karamcheti, and A. A. Chien: Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs. *IEEE Concurrency*, 60-73, April-June 1997.
11. R. R. Raje, J. I. William, and M. Boyles: An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. *Concurrency: Pract.&Exp.*. 9(11):1207-1211, 1997.
12. M. Snir et al.: *MPI: The Complete Reference.* The MIT Press, 1996.
13. J. W. Stamos, and D. K. Gifford: Remote Evaluation. *ACM Transactions on Computer Systems*, 12(4):537-565, October 1990.
14. http://www.javasoft.com.