The AL++ project: object-oriented parallel programming on multicomputers

M. Di Santo, F. Frattolillo, W. Russo and E. Zimeo[*]
*University of Salerno - *University of Napoli, Italy*
Fax: +39 89 964574 - E-mail: disanto@dia.unisa.it

Abstract
AL++ is a software system which combines high-level object-oriented facilities with the simplicity, flexibility and power of the Actor computational model. AL++ lets programmers develop C++ parallel applications and run them on multicomputer platforms.

Keywords
Parallelism, Multicomputers, Actors, C++, Object-oriented parallel programming

1 PROJECT AIMS

Object-oriented programming models provide an attractive base for developing parallel programming systems. In fact, they promote the effective application of modern software engineering techniques, which have already proven to be successful in developing complex and large-scale sequential applications. Moreover, thanks to the dynamic creation and reconfiguration of objects, they also support applications whose computational structures can not be statically determined and facilitate decisions about object placement and migration, by aggregating data and code into single semantic units. In short, object-oriented parallel models seem to offer the expressiveness and the efficiency which are needed to effectively harness the computational power of modern, distributed-memory multicomputers.

Among the object-based models of parallel computation, *Actors* (Agha, 1986) is the best known. It can be classified as a partly abstract model based on process nets (Skillicorn, 1993) which allows computations to be specified without restricting their form. The Actor model has recently become the basis for a number of parallel object-oriented programming languages, such as ABCL (Yonezawa, 1990), CA (Chen, 1993) and HAL (Agha, 1992), even though it still has to establish itself as a practical tool for the development of parallel software. This is due to the difficulties encountered in turning the model into a truly general-purpose, object-oriented parallel programming language, to the scarcity of efficient implementations and to the limited experience for significant applications. In conclusion, the object-oriented approach, particularly if based on the Actor model, is well-suited for structuring parallel activities, but many further research and implementation efforts are needed in order to provide parallel programmers with elegant language ideas efficiently implemented on existing hardware.

These considerations have motivated the AL++ project which began in 1990 as one of the research proposals to be developed within the national project "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, sottoprogetto Architetture Parallele", sponsored by the *National Research Council* (CNR) of Italy. Part of the research activities were also developed within the project "Architetture Convenzionali e Non Convenzionali per Sistemi Distribuiti" sponsored by the *MURST* (Ministero dell'Universit e della Ricerca Scientifica e Tecnologica).

The AL++ project aims at developing a programming system characterized by the following requirements: (a) to provide programmers with elegant and simple mechanisms to develop object-oriented parallel applications on distributed-memory architectures; (b) to enable application code to be independent of underlying hardware/software platforms; (c) to achieve a modular implementation of the programming system so that it can be ported on new hardware with a reasonable effort.

2 KEY RESEARCH ISSUES AND ACHIEVEMENTS

Since the design and implementation of a new language is an expensive activity, the simpler approach of embedding Actor concepts and primitives into a widespread sequential programming language has been followed. In particular, the main achievement of the project has been *AL++*, a semantic extension of C++, implemented through a class library which provides an object-oriented interface for actor programming. The choice of C++ has been motivated by its availability and popularity with programmers. Another motivation is that C++ is efficiently implemented with a minimum of run-time support on all the architectures of major interest.

2.1 The actor model

Actors are objects which manifest a pure reactive nature and interact with other actors only via *message passing*. They unify both data and code in local states, called *behaviors*, and are dynamically created and referred through system-wide identifiers, called *mail addresses*

The communication mechanism is point-to-point, asynchronous and one-directional. Because mail addresses may be transmitted via messages, the actor-net which shows the potential flow of information may dynamically change. Messages are guaranteed to be delivered to their destinations, but transmission order is not necessarily preserved at delivery. Incoming messages are buffered into unbounded queues associated to receiving actors, before being serially processed. Functional interactions among actors are modeled with the use of *continuations*; that is an actor, instead of returning a result, sends it to a continuation actor that it knows about.

The processing of a message triggers the execution of the actor *script*, the code in the behavior of the receiver. During this processing, new actors can be created, messages asynchronously sent and the current behavior substituted by a new one (*replacement behavior*). In practice, replacements implement local state changes which can span from simple updates in the values of state variables to radical changes in the set of state variables and in the script.

2 The AL++ interface

AL++ enables programmers to exploit software engineering techniques in modeling parallel applications. In fact, AL++ joins C++ object-oriented powerful facilities, such as data abstraction, multiple inheritance, overloading and dynamic binding, with the clear, simple and flexible mechanisms provided by the Actor model. Moreover, thanks to the support for automatic and dynamic resource management, programmers can design AL++ programs as *ideal* algorithms, without having to specify allocation strategies or other programming details which make them depending on specific hardware platforms and network topologies.

AL++ supports the *SPMD* (Single Program Multiple Data) computational model; therefore, each node in the system stores and runs the same program; data, on the contrary, is distributed among all the nodes. The library makes available all the basic abstractions and primitives of the Actor model. Messages and behaviors are dynamically created instances of user classes respectively derived by the library classes *Message* and *Behavior*, while mail addresses are instances of the library class *MailAddress*. Actors are dynamically created by invoking the member function *MailAddress::create*. The behavior of the new actor can be specified at the creation time or later, by invoking the member function *MailAddress::init*. In the latter case it is possible to create actors whose behaviors mutually refer.

The *Message* class defines all the communication and message management primitives as its member functions. In particular, *Message::send* sends a message to a target actor, while *Message::request* associates to the sent message the identity of a *continuation* actor, which will be used as the implicit destination of the result when the target actor executes *Message::reply*.

Each user class derived from *Behavior* must include the local data as its data members and define the pure virtual member function *script*, which accepts the message to be processed as an argument. In many cases the script selects, on the basis of the tag associated to the message and returned by *Message::type*, the appropriate method and invokes it. *Behavior::become* permits to specify the actor replacement behavior, while *Behavior::self* returns the mail address of the current actor.

AL++ enables to control the dynamic placement of actors in two ways: (1) automatically, by employing one of the dynamic load balancing strategies integrated into the runtime support: *random*, *ACWN* (Shu, 1989) and *PWFA* (Di Santo, 1995, in preparation); (2) in a programmed way, by utilizing some primitives which allow both to specify the node on which an actor is to be created and to *migrate* actors according to the computation load at run-time. Moreover, immutable actors may be *duplicated* and "garbage" actors explicitly *deallocated*.

2.3 Implementation issues

The AL++ interface is built on top of a runtime support, called *ASK* (*Actor System Kernel*), which has been designed so as to fully exploit the power of the underlying hardware, and to be flexible enough to represent a stable basis for further enhancements. A working prototype of the kernel has been developed for Transputer networks.

To make the kernel portable to different hardware/software platforms and independent of network characteristics, it is built on top of a low-level interface which consists of two components: an *abstract node environment*, providing each node with facilities for running concurrent threads which interact through some shared-memory mechanism (semaphores or equivalent), and an *abstract network environment*, providing node-to-node asynchronous communication primitives and taking charge of performing routing between non-adjacent nodes and of buffering incoming messages.

An instance of the kernel, consisting of a few threads implementing system processes, is present on each node. One of these threads is the *scheduler* which cyclically schedules a local actor and processes messages in its mail queue; it is worth noting that the processing of a message can not be suspended and, therefore, once started, proceeds till its completion. Another thread is the *server* which carries out the remote requests as though they were issued locally.

Mail addresses are represented with global identifiers generated according to a completely distributed scheme that does not introduce overhead. The identifiers are then translated into physical addresses by a *lookup table* that returns either a local memory address, or a node identifier, according to the physical allocation of actors. In the latter case, a system message is sent to that node, and a new access to the lookup table is performed upon arrival.

Migration can be implemented quite cheaply in an actor based system. In ASK all the steps needed are fully asynchronous and so, while the actor migration proceeds on a node, other activities allocated on the same node have not to wait, but they are allowed to do useful work. Migration times are therefore masked by the resulting parallel execution of system threads, and they only affect the response time of the messages in the queue of the migrating actor. The migration procedure has been adopted as a basis to implement the *remote creation* primitive. In fact, an actor is always locally created and only then asynchronously migrated to its remote destination. This mechanism permits to minimize the time spent for an actor creation and to maximize the locality of data references in the first phase of actor existence.

2.4 Performances

The prototype implementation of ASK has been developed in the 3L Parallel C programming environment, and runs on a network of sixteen T800, clocked at 20 MHz, with links at 20 Mbits/s. Two versions of the network environment (NE) are available, respectively for ring-connected and 2D-torus networks.

Table 1 shows execution times of the four basic AL++ primitives (creation of a new actor, assignment of an initial behavior to a new actor, sending of a void message and replacement of the current behavior) in the case of purely local execution.

Table 1 Local execution of some AL++ primitives (μs)

create	*init*	*send*	*become*
39	54	223	65

Table 2 shows execution times of the *send* primitive as a function of the distance in *hops* of the target node. The table also reports the overall amount of time spent in the NE. The execution time of a remote *create* is constantly equal to 71 μs, in that ASK always performs a local creation asynchronously followed by a migration of the actor.

Table 2 Remote execution of the *send* primitive (μs)

	1 hop	*2 hops*	*3 hops*	*4 hops*	*5 hops*	*6 hops*	*7 hops*	*8 hops*
send(NE)	382(113)	466(177)	535(240)	604(305)	690(368)	771(432)	854(496)	920(583)

2.5 Bibliography

In the following we provide a list of the AL++ key publications written in English:

Arcelli F., De Santo M., Di Santo M. and Picariello A. (1993) Computer Vision Applications Experience with Actors, *PARLE '93*, 14-18 June 1993, Munich (Germany), *LNCS 694*, Springer-Verlag, Berlin (Germany).

Di Santo M. and Iannello G. (1990) ASK: A Kernel for Programming Actor Systems, *Procs. of the 1990 ACM SigSmall/PC Symposium on Small Systems*, ACM Press.

Di Santo M. and Iannello G. (1991) Implementing actor-based primitives on distributed memory architectures, *Procs. ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming*, 21-22 Oct. 1990, Ottawa (Canada), *OOPS Messenger* 2(2), ACM Press.

Di Santo M. and Iannello G. (1992) Implementation of dynamic languages on multicomputer architectures, in *Parallel Computing: Problems, Methods and Applications* (eds. Messina P. and Murli A.), Elsevier, Amsterdam (Nederland), selection of papers presented at the *Conference on Parallel Computing: Achievements, Problems and Prospects*, 3-7 June 1990, Capri (Italy).

Di Santo M., Iannello G. and Russo W. (1992) ASK: a Transputer implementation of the Actor model, *Int'l Conf. on Parallel Computing and Transputers Applications*, 21-25 Sept. 1992, Barcelona (Spain), IOS Press, Amsterdam (Nederland).

Di Santo M., Frattolillo F. and Iannello G. (1992) *Actor System Kernel (ASK) 4.0. Introduction and User Guide*. Tech. Rep. n. 3/108, CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo.

Di Santo M., Frattolillo F. and Iannello G. (1994) *Run-time support for highly parallel algorithms on multicomputer architectures*, Tech. Rep. n. 3/139, CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo.

Di Santo M., Frattolillo F. and Iannello G. (1995) Experiences in Dynamic Placement of Actors on Multicomputer Systems, *Proceedings Euromicro Workshop on Parallel and Distributed Processing*, San Remo 25-27 Jan. 1995 (Italy), IEEE Computer Society Press.

Di Santo M. and Iannello G. (1995) Actor Models, in *General Purpose Parallel Computers: Architectures, Programming Environments and Tools* (eds. Balbo G. and Vanneschi M.), Edizioni ETS, Pisa (Italy).

Di Santo M., Frattolillo F., Russo W. and Zimeo E. (1995) *A Dynamic Load Balancing for Object-Based Computations on Multicomputers*, in preparation.

3 FUTURE DIRECTIONS

The AL++ project is still alive and we want to utilize the experiences accumulated since its start in order to globally redesign both its interface and implementation. Precisely, at the interface level, we will proceed to substitute C++ with the new object-oriented language *Java* (Gosling, 1995) which will offer the advantages of being, according to its authors: (i) simple and familiar; (ii) architecture neutral, portable and robust; (iii) interpreted, dynamic, secure and multi-threaded; (iv) efficient and equipped with extensive and well developed class libraries. Moreover, we will complete the programming interface with mechanisms for expressing *local synchronization constraints*, which permit to delay the processing of messages until they are "serviceable", and *grouping of actors*, which permit to express data parallelism, to support broadcast communication and to implement distributed objects.

On the other hand, at the implementation level, we will move our environment to a *network of workstations* equipped with *PVM* (Geist, 1992), which nowadays have proven to offer viable and cost-effective platforms for parallel computing in many application domains. Moreover, we will design and implement new mechanisms for explicit and automatic resource management at runtime: among these we will include new algorithms for dynamic placement and distributed garbage collection of actors.

4 ACKNOWLEDGMENTS

The AL++ project has been partially supported by CNR, under funds of "Progetto Finalizzato

Sistemi Informatici e Calcolo Parallelo", and by MURST, under funds 60% and 40% "Architetture Convenzionali e Non Convenzionali per Sistemi Distribuiti".

In addition to the authors, Giulio Iannello of the University of Napoli "Federico II" has contributed to the project.

We gratefully acknowledge Gul Agha of the University of Illinois at Urbana-Champaign for inspiring our work and providing insights into the Actor model and its implementation.

5 REFERENCES

Agha G. (1986) *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press.

Agha G. and Houck C. (1992) HAL: A High-level Actor Language and Its Distributed Implementation, *Proceedings of the 21st International Conference on Parallel Processing* (ICPP '92), Aug. 1992, St. Charles (IL - USA).

Chien A. (1993) *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. The MIT Press.

Geist G. A. and Sunderam V. S. (1992) Network-Based Concurrent Computing on the PVM System, *Concurrency: Practice and Experience*, 4(4).

Gosling J. and McGilton H. (1995) *The Java Language Environment: a white paper*, available at http://java.sun.com

Shu W. and Kalè L. V. (1989) *Dynamic scheduling of medium-grained processes on multicomputers*, Tech. Rep., Dep. of Computer Science, Univ. of Illinois at Urbana-Champaign.

Skillicorn D. B. (1993) Models for parallel computation, in *Advanced workshop on Programming tools for parallel machines*, 21-25 June 1993, Otranto (Italy).

Yonezawa A. (ed.) (1990) *ABCL: An Object-Oriented Concurrent System*. The MIT Press.

6 BIOGRAPHIES

Michele Di Santo is a professor of computer engineering at the University of Salerno, Italy. He received the degree in electronic engineering, cum laude, from the University of Napoli and worked at the University of Napoli and the University of Calabria. His scientific interests include programming languages and environments for parallel and distributed systems. He is a member of ACM and IEEE Computer Society.

Franco Frattolillo is a faculty member at the "Dipartimento di Ingegneria dell'Informazione e Ingegneria Elettrica" of the University of Salerno, Italy. He received the degree in electronic engineering, cum laude, from the University of Napoli. His research interests include parallel and distributed architectures and programming environments for parallelism.

Wilma Russo is an associate professor of computer engineering at the University of Salerno, Italy. She received the degree in physics, cum laude, from the University of Napoli and worked at the University of Calabria. Her scientific interests include programming languages and environments for parallel and distributed systems.

Eugenio Zimeo holds a scholarship from CNR at the University of Napoli, Italy. He received the degree in electronic engineering, cum laude, from the University of Salerno. His research interests include parallel and distributed architectures and programming environments for parallelism.