High-Level Management of Allocation in a Parallel Objects Environment

Antonio Corradi, Letizia Leonardi, Franco Zambonelli

DEIS Technical Report No. DEIS-LIA-96-003, LIA Series No.10, June 1996

High-Level Management of Allocation in a Parallel Objects Environment

Antonio Corradi[1], Letizia Leonardi[2], Franco Zambonelli[1]

1) Dipartimento di Elettronica Informatica e Sistemistica - Università di Bologna
2, Viale Risorgimento
40136 Bologna - ITALY
Ph.: +39-51-6443083
Fax: +39-51-6443073
{acorradi, fzambonelli}@deis.unibo.it

2) Dipartimento di Scienze dell'Ingegneria - Università di Modena
214, Via Campi
41100 Modena - ITALY
Ph.: +39-59-378517
Fax: +39-39-370040
leonardi@dsi.unimo.it

DEIS Technical Report No. DEIS-LIA-96-003, LIA Series No. 10, June 1996

Keywords: Parallelism, Object-Orientation, Programming Tools, Load Balancing

Abstract

Widespread diffusion of parallel architectures is currently limited by the lack of tools that allow to efficiently exploit the available resources with few programming efforts, in particular with regard to the allocation problem. The paper presents a configuration language (ACL) implemented within an object-oriented parallel programming environment. By the means of ACL, the user can specify its applications allocation needs without being aware of the architectural details. The main point is that ACL directives guide the run-time support, by adapting its general-purpose behaviour to follows applications peculiar allocation needs. The ACL approach is validated by testbed applications.

High-Level Management of Allocation in a Parallel Objects Environment

Antonio Corradi, Letizia Leonardi, Franco Zambonelli

1. Introduction

The software development process includes different phases at different abstractions level: the analysis of the problem, the algorithm definition, the programming language implementation. All these phases are abstract, since they do not usually need to face any architectural issues. Concrete phases, such as loading, executing and debugging, instead, require a large knowledge of the architectural details of the target computing system. A clear isolation criterion is usually adopted: abstract phases are separated from concrete ones. The duty of abstract ones is given to the programmer, of concrete ones to automatic tools of the programming environment. In the parallel programming field, in particular, an application is decomposed in several cooperating entities, in a complete abstract way. Only during the loading and the executing phases the allocation of the application onto the resources of the target architecture is decided by the means of automated tools.

When dealing with the allocation problem in parallel and distributed environment, this solution is not satisfactory. Automatic allocation tools, moving toward general purpose goals such as the load balancing one [ShiKS92], cannot act in respect of the peculiar needs of the applications. In the effort toward the equalisation of the execution load, for example, frequent communicating entities can be moved far with a great increase in communication costs.

We think the separation between abstract and concrete phases, though important and to be maintained, must be thinned: in particular, it is necessary to allow the user to specify in an abstract way its application allocation needs; then, the run-time support of the programming environment must be able to work by taking into account the user allocation directives, concretise it for the specific target architecture and let them produce feedback on the automated allocation policies. That way, parallel applications can be implemented without worrying about the physical architecture and efficiently executed.

The Abstract Configuration Language (ACL for short), presented in the paper and integrated within the Parallel Objects (PO for short) programming environment [CorL91], achieves this goal. The PO programming model follows the guidelines of the object-oriented paradigm [Weg90], and enriches it by the means of parallelism: several object can execute at the same time and the same object can host several concurrent activities. Such a scenario, highly dynamic because of run-time object creations and passing of object references, makes it impossible to statically foresee the application execution and communication needs of resources. Thus, dynamic allocation tools are needed [CorLZ94].

In PO, the dynamic automatic allocation policies integrated within the run-time support adapt their behaviour to the ACL allocation directives, so to follows their goal in respect of the application allocation needs. ACL, however, remains at an abstract level since its directive does not refer to any particular architectures: it is the support that translate them so to let them assume concrete meaning.

ACL directives are defined at the class level, so to define the allocation needs of the instance objects. That allows the user to characterise its applications by acting not at the level of the single application components, but at the level of the class property. This choice is even more important in an object-oriented environment, such as the PO one, characterised by inheritance: ACL directives, defined at the class level, can be subject to inheritance, allowing the unskilled user not to worry about them and let them to be inherited from super classes.
The paper is organised as follows. The allocation problem is analysed in general in section 2. The PO environment is briefly described in section 3, both with its allocation issues. The ACL language is presented in section 4 and an application examples, showing the effectiveness of the approach, is described in section 5.

2. The Allocation Problem

The allocation of a parallel application onto a specific architecture is the assignment to its components of memory, communication and execution resources. The allocation is said to be **static** when the decision about the allocation of an application are taken before the execution of the application itself [NorT93], **dynamic** allocation, when allocation decisions are taken at run-time. Dynamic allocation is necessary whenever a parallel application is subject to different execution phases, with different needs in terms of resources, or whenever its behaviour is not a priori determinable [ShiKS92].
A parallel programming environment should be able to face the allocation problem, both in static and dynamic case, in the respect of two main requirements

- **transparency**, i.e. the programmer should be made unaware of the allocation onto a given architecture, by having him concentrate only on the logical structure of its applications;
- **efficiency**, i.e. the allocation must both produce a good exploitation of those resources and minimise the execution time of the application.

To provide full transparency, a parallel programming environment must integrate **automatic tools** for dealing with the allocation problem. During the static allocation phase, it is the job of the compiler to extract from the application code all the information needed to build the application graph, to assign the application a given number of processor and to map it on the basis of a mapping algorithm [NorT93]. Dynamic allocation tools are substantially different. The application behaviour must be monitored at run-time and then, on the basis of the information obtained, decisions must be taken about the allocation of the new entities that enter the system (remote creation). If already allocated entities are allowed to **migrate** from one node to another one [Nut94], the allocation tools must evaluate whether the situation requires it or not [CorLZ94].
Static allocation tools, due to their off-line nature, can take into account (depending on the available a priori information) both the execution and the communication needs of an application. Dynamic allocation decision, because of their real-time nature, cannot deal with high order complexities related to communication: they only goal they can effectively pursue is the equalisation of the execution load (**load balancing**).
Several distributed systems and programming environments have been developed in the last few years that integrate automatic tools for dynamic allocation. Among the others, Demos/MP [PowM83], one of the first system that integrated in itself process migration mechanisms and load balancing policies, and Charlotte [ArtF91], both providing full transparency; even object

oriented systems exists with an automated approach to the allocation problem [Mot94, Nut94]: in Orca [BalK93], for example, not only remote creation and migration is automatically managed by the system, but replication too.
The limit of automatic allocation tools is their inability to manage the peculiar needs of the applications. In general, within a system or programming environment, the same allocation policy is used whatever the application is: it is clear that the same goal pursued with the same strategy cannot fits the needs of all possible applications, introducing inefficiencies in their execution. In the static case, for example, an incorrect allocation of the entities situated on the application critical path [ShiWP90] or the incorrect identification of the application bottleneck [Hac89] can produce a bad exploitation of the system resources and slow down the application execution. In the dynamic case, migration decisions can move far, with the aim of load balancing, heavily communicating entities so to increase their communication costs and, finally, slowing them down; on the other side, the policy could decide to allocate on the same node entities that would have profited from parallel (not simply concurrent) execution.

When the allocation of automated tools is not satisfactory, one might decide to sacrifice the transparency requirement in order to obtain better efficiency. The user, in this case, is given the duty of studying its applications behaviour, by the means of static code analysis, profiling and post-mortem debugging tools. That allows him to detect where inefficiencies reside and what allocation could best fit its application. Such a solution, adopted for example in the PARADE environment [Bru93], can surely increase the execution efficiency, but require great tuning efforts for any application. Moreover, such kind of approach, is not viable for dynamic application.
The capability of dynamically influencing the allocation of a parallel application can be provided in two ways:

1. by using libraries of allocation policies among which to choose the one that best fits the peculiar application needs;
2. by inserting within the application code the allocation directives.

The first method is mainly limited by the fact that often a single allocation policy is not the best for the entire execution of the applications: many kind of applications are characterised by different execution phases. In addition, the choice of a policy is not trivial: neither theoretical basis nor well established experimental results are available to help in the choice.
The second solution is more viable, but the semantics of the allocation directives to be inserted in the code and their interaction with the run-time support has to be carefully evaluated. It would be important that the allocation directives be independent of the target architecture. That makes it unnecessary for the user to deal with architectural details and makes the application code more portable. Many systems do not respect that requirement [Hel89]. A bigger effort toward abstraction is attempted in modern object-based environment, such as Emerald [Jul88] and its relatives, Amber [Cha89] and DOWL [Ach93]. In these systems, coresidence directives on related objects and migration directives allow the user to specify in the application code the mutual locality needs of interacting objects in abstract terms, without need of knowing the physical details of the target architectures. However, in these systems, the run-time support is not capable of any "intelligent" interaction with the user directives: all of them are considered as commands to be executed, and actions are not influenced at all by the current system state but in case of physical impossibility of executing them;

We claim the role of the run-time support role is important, even in presence of user-defined allocation directives. It can filter user directives by evaluating them in consideration of the current system state. For examples, the conditions under which migration is beneficial are strictly dependent on the system state [CorLZ94]: the run-time support, before being blindly commanded by the user to move some entity, could verify whether such conditions exist or not. Directives are not commands but "hints". Moreover, automatic allocation policies integrated within the support can be implemented to work accordingly with the user directives: the load balancing goal can be followed in respect of the application allocation needs.

We think the best solution to the allocation problem is the integration of:

- user-defined abstract allocation directives and
- automatic allocation tools.

While the first allows users to specify the application needs in abstract terms, the second complete the allocation by taking into account the current system state and its characteristics.
This approach provide a compromise between a transparent but inefficient allocation and an efficient but architecture-dependent one. Moreover, a hybrid solution makes it easy to move from one extreme to the another: on the one hand, the presence of automatic tools makes an application work even in absence of any user-defined directives; on the other hand, user-defined directives can be so constrainable to completely inhibit automatic allocation tools.
In the static case, it is easy to make automatic tools cooperate with the user: their off-line nature, as previously stated, allows such kind on interaction. In the dynamic case, instead, few systems implement a form of cooperation between the user abstract level and the one of the run-time support. In the AL-1D object-based reflective system [OkaI94] each object is associated with a meta-level object manager, that defines the object allocation needs and is in charge of its allocation decisions. The meta-level object, inserted at the run-time support level, is allowed to dynamically exploit, in its allocation decision, information about the state of the system. In that system, however, the meta-object definition is still strictly architecture dependent.
In the PO environment, whose presentation follows, the ACL language is capable of both maintaining its directives at an abstract, architecture independent level, and let them cooperate with the automatic policies integrated in the run-time support.

3. The PO Environment

The object-oriented programming paradigm [Weg90] is particularly suitable for the implementation of parallel programming environments. This section describe the PO programming model and its allocation issues. PO is currently implemented on a massively parallel architecture, a Meiko CS-1 [Mei89].

3.1. The PO Model

The PO environment is based on the **active object model** [ChiC91]. At least one execution thread is associated with each object: these threads are the only ones allowed to execute the object code and to access the state of the object.
PO can express parallelism with two dimensions of concurrency, by associating independent execution capacity with different objects and also by allowing multiple threads of execution within the same object. We distinguish between **inter-object** and **intra-object parallelism**.

In parallel objects languages, computation results from message passing between objects. When one object requires an external service, it sends a message to another object: the message specifies the service the sender needs. The receiving object verifies whether the operation is correct and, then, executes the requested operation. Asynchronous communication modes enforce real inter-object parallelism: both the sender and the receiving objects execute at the same time.
The second form of parallelism, the intra-object one, arises because of **multiple activities** (similar to threads) within the same object. Since a parallel object can receive more than one request, each parallel object can execute several internal activities, each for a service of one request. Activities are created on request and terminated when the service is completed. The PO model does not constrain the number of internal activities that run within the same object, but the servicing of a request can be delayed on the basis of synchronisation constraints with the already executing activities [CorL91].

In a more detailed perspective, each PO object is composed of two logically separated parts: the **parallel part** and the **non-parallel** one. The structure of the non-parallel part is composed of a **state** and a set of operations, called **methods**, defining the object interface. The parallel part is composed by a **management** part, whose primarily role if of interfacing the object itself with the external and of synchronising the other parallel parts, the **activities**, each one is a process devoted to the execution of a method corresponding to a service request.
PO objects are always instances of a given class. PO classes describe both the non-parallel and the parallel part of an object. In particular, a class describes the interface (i.e. all the operations that can be requested to its instances), the state variables of an object, and the synchronisation constraints. A **tight connection** between class and its instances forces instances to share the description maintained in the class: the behaviour of an instance is contained in its class and any updating of a class shows automatically its effects to all its children/instances. This approach becomes ineffective in a distributed environment, if classes and instances can be non coresident. This is why the PO environment adopts a solution with multiple class copies, one on each node where is executing an instance of the class. The distribution of the replicas of classes does not incur in any consistency problem: up to now classes cannot change at run-time.

3.2. Allocation Issues in PO

The scenario of a PO application is highly dynamic.
A first aspect of dynamicity, a intra-object one, comes from the servicing of the requests by a server object: every request imply the dynamic creation of a new thread of execution within the object.
Objects themselves have dynamic nature: when a PO application starts, it is composed of some initial objects, called **starters**; every object but the starters is created at run-time, in an explicit way. Starters objects are in charge of booting the application, by exchanging the first service requests and eventually by creating new objects and request them some service. Moreover, every newly created object can create new objects in its turn and send them requests. As objects can be dynamically created they can be dynamically destroyed.
In PO, objects and activities not only are dynamically created, but their allocation too must be dynamically decided at the creation time.

The dynamic creation of objects requires the support to be able to manage it, by implementing **remote creation** policies and mechanisms, able to decide the allocation in dependence of the system state (i.e. the load of the system nodes) at the moment of creation. The PO environment also provides the possibility for an object to **migrate** [CorLZ94], by moving it from a node (the sender one) to another (the receiver one), during its execution life. Migration can be useful to make objects follow the system evolution and it is suited to produce better load balancing than simple remote creation policies.

When a PO object is completely allocated onto a single system node, it is said to be a **concentrated object**: all its executing activities will exploit concurrency. The potential parallelism of the intra-object level can be transformed in real parallelism by distributing the activities internal to an objects onto several nodes. In this case the object it is said to be **distributed**. Moreover, object distribution is allowed not only at the activity level but at the state level too: it is possible to partition the object state and distribute it onto several nodes. In this case, we talk of **partitioned object**. It should be clear that a distributed object can get advantage from having its state partitioned, so to allow its distributed activities to exploit locality in the access to the state.

4. The PO Abstract Configuration Language

The separation between automatic allocation and user-guided one is solved, in PO, by the definition of the **A**bstract **C**onfiguration **L**anguage (**ACL**). This section describes first the main characteristics of the ACL language, then it present some examples of ACL directive. Finally, it analyses how the ACL directives can influence the behaviour of the run-time support allocation policies.

4.1. ACL Overview

We can point out three main characteristics that make the ACL language suitable for solving the problem, analysed in section 2, that arises in the allocation of parallel application onto parallel and distributed architectures:

- **abstraction**: ACL directive are not tied to peculiar characteristics of one kind of architecture; instead, they are abstract properties tied to the semantic of the applications;
- **definition at type (class) level**: this allows users to integrate the allocation needs as a behavioural characteristic of the type;
- **interaction with the run-time support** of the environment: this makes it possible for the ACL directives to produce different effects in dependence of the current system execution state.

The capability of maintaining at an abstract level, independent of the architecture, the allocation directives in very important: it is in charge of the run-time support, and depending on the target architecture, to translate this abstraction into concrete "physical" terms. This abstraction level allows both the user to be unaware of any architectural details and applications to be portable across different architectures.

The definition of the directives at a class level allows users to exploit the inheritance mechanism: the allocation behaviour of a class can be inherited from super classes. That way, unskilled users can disregard, in the definition of a class, its allocation behaviour, by leaving the system adopt the one inherited from super classes.

ACL has not been conceived to be the only allocation tools within the PO environment. By the means of the ACL the user can define the allocation needs of its allocation components but if integrated in an environment that provides automatic allocation tools then ACL becomes a way of influencing the system allocation policy. The automated policy, conceived for general purposes goals such as load balancing, must act in agreement with the ACL directives, so to adapt its behaviour to the peculiar allocation needs of applications. It is clear that, depending on the user skill with the allocation problem, the ACL directive could be more or less constrainable. At one extreme, the user can totally ignore ACL and let the allocation of the applications to be completely decided by the system policy; since the system policy can not always act in respect of the application needs this might result in a lot of inefficiency. At the other extreme, the user can specify the application allocation needs in such a complete way as to almost (or fully) disable the automatic allocation policy. If the directives specified by the user are clever, they will realise "ad hoc" allocation policy for its application.

4.2. ACL Directives

ACL directives, specified in the definition of a PO class, describe the allocation behaviour of the instances of the class in abstract allocation terms. Abstraction, and so the capability of those directives to maintain their validity over a large spectrum of architectures, come from the fact that instance allocation needs are expressed by concepts, such as migratability and locality, that do not refer to any particular architecture. Only at the support level, do these concepts assume concrete meaning, depending on the particular characteristics of the implementation architecture.

As an example, the locality concept (that in ACL is expressed by the "close_to" directive), can assume very different meanings. On a transputer based architecture, two entities can be considered close if they reside on the same node or on nodes that are directly connected by a communication link. In a network of workstations, because of the large difference in cost between inter-node and intra-node communication, closeness must be considered equivalent to coresidence. In a shared memory architecture with uniform memory access, the locality concept loss its meaning and it is equivalent to "anywhere". In any case, independently of the particular meaning given to that concept, directives remains abstract and **portable**.

In the PO model, we have identified two level of parallelism, the intra-object and the inter-object one. Their presence influence the allocation needs of a PO application and, consequently, ACL directives. In particular, we distinguish between:

- **intra-object directives**, that refer to the allocation of the components internal to an object, activities and state, eventually in relation to each other;
- **inter-object directives**, that refer to the allocation of objects and their parts in relation with the allocation of other objects.

We are not interested here in laying a complete list of all the ACL directives. Rather, we want to clarify, by the means of several examples, their semantics and their interaction with the run-time support.

4.2.1. Intra-Object Directives

In the previous section we have seen that, whenever it is beneficial, the internal concurrency within a PO object can become real parallelism by distributing the object activities onto several nodes.

By default, a PO object is concentrated, so to be an atomic unity of allocation. It is possible to specify, in the definition of a class, the **distributed()** directive. In this case, the activities of the instances of that class are allowed to distribute themselves in a local environment. That means, for example, that in a transputer-based architecture objects could be distributed among five nodes (a central one and its four direct neighbours). We claim that the distribution directive does not automatically imply that all the instances of the class in which the directive is defined will be distributed objects. In particular, the run-time support provides object distribution only whenever it detect that such a distribution would provide benefits, in particular with regard to the load balancing purposes of the automatic allocation policy integrated within it. The definition of that simple ACL directive allows us to make us clarify some previous consideration:

– the directive does not represent an imperative command. What the user specify is that the objects of a given class could get benefits from real intra-object parallelism, but only at run-time, by the means of the dynamic allocation policy and the information it holds about the system state, it is decided whether the distribution is beneficial or not;
– the directive apply to any architecture. However, on different systems, the concrete meaning it assumes could make it produce different effects.

It is always possible to make to ACL directives assume imperative meaning, with a simple syntactic notation (**distributed()!**). In this case, the support interprets them as a command, whose execution is not related to the current system state. Just in case of physical impossibility (for example an object that cannot be distributed because of the lack of the necessary resources), the imperative directive is not issued. Imperative directives should be used sparingly, since reducing the interaction with the current system state could lead to a bad exploitation of system resources.

Another directive that can be specified at the intra-object level is the one that allow the migration (**migratable()**). Object can be subject of migrations during their life, and a default object behaviour could either allow or forbid objects to migrate. In the PO case the default is to forbid object migration: objects of given classes must be explicitly allowed to migrate by the means of the migratable directive.

The migratable directive, again, represents a hint given to the support rather than a command. The allocation policy can load balance the system by taking into account the possibility of migrating only those objects that have been defined as migratable. Even if it is syntactically possible to give imperative character to the migratable directive, this is meaningless: object are migrated when in need of the load balancing policy.

The presence of the parenthesis in the syntax of the directives lets suppose the possibility of parametrise them. In absence of any parameter, a default is assumed. Thus, we could have specified, in the distribution directive, a number of nodes onto which to distribute one object different from the default one (that is, the number of neighbour nodes). In the case of the migratable directive, the parameter, infinite by default, set the maximum number of times an object is allowed to migrate.

4.2.2. Inter-object Directives

The other class of ACL directives that can be specified in PO is the one that relates the allocation of the instances of a class (or of their parts) with the allocation of other application components. They represents mutual allocation property.

We distinguish several different mutual allocation needs that can occur among two components of an application:

- coresidence (**coresident()** directive), when two components needs to share the same physical resources;
- closeness (**close_to()** directive), when two components, because of intensive reciprocal communication, need to be allocated onto neighbour nodes;

- neighbourhood (**neighbour()** directive), when two components with a high degree of communication need to be allocated close but not on the same execution nodes, so to allow them to execute in parallel;

Since the inter-object directives specify mutual property, one must identify the involved entities. One is clearly found as being an instance of the class in which the directive is defined. The other (partner), in relation with the first, must be specified. The problem is that, since directives are defined at the class level, the partner must be identified before the creation of the instance itself.

In an object-oriented environment every reference to other objects can be contained either in a state variable or in the parameters passed to a service request: those references can be used to specify the partner. As an example, suppose that, within the instances of a class C, the state variable varX represents a reference to an object. A directive close_to(varX), specified in the class C, tells that any instance of the class C should preferably be allocated close to the object that is currently referenced by varX. Depending on the actual value that varX assumes, objects of the class C will be tied to given partners in a locality relationship. The same can be specified for the parameters of the operations.

It is even possible, for the partner, to be identified not by a reference to it but on the basis of its role with respect to the object that specify the ACL directive. As an example, every object is created by an activity of another object. The creator object could become the partner of a relationship specified in the class definition of the objects it creates. The directive coresident_to_creator(), for example, suggest the allocation of instance in the same node of the object that issues the creation request. Truly, every inter-object directive, when no parameter is specified, set as default parameter the creator object.

While the neighbour() and

ERROR: syntaxerror
OFFENDING COMMAND: bin obj seq, type=128, elements=1, size=36097, non-zero unused field

STACK: