How to Control the Allocation of Parallel Applications: a Survey on Tools and Language Constructs

Franco Zambonelli

Dipartimento di Scienze dell'Ingegneria - Università di Modena
Via Campi 213/b - 41100 Modena - ITALY
Ph.: +39-59-378517 - Fax: +39-59-370040
E-mail: zambonel@dsi.unimo.it

Ciaran McHale

Distributed Systems Group - Department of Computer Science - Trinity College
Dublin 2 - IRELAND
Ph.: +353-1-6081539 - Fax: +353-1-6082204
E-mail: cjmchale@dsg.cs.tcd.ie

Key-Words: Allocation, Dynamicity, Tools, Language Constructs

Abstract

The paper focuses on the allocation area, with the aim of providing a survey on systems that provides facilities to codify the behaviour of an application with regard to its allocation onto a target architecture. The paper analyses the design issues that arise in the definition of these facilities and presents the solutions adopted in several systems and programming environments. Open issues and future directions of research are outlined.

How to Control the Allocation of Parallel Applications: a Survey on Tools and Language Constructs

Franco Zambonelli, Ciaran McHale

1. Introduction

Parallel programming introduces complexities not present in traditional sequential programming, such as non-determinacy, synchronisation, distribution [AlmG89]. To achieve large user acceptance, parallel programming environments must integrate tools to help users in managing those complexities and to permits them to achieve high-performance with a limited effort. The implementation of tools for dealing with the allocation problem, i.e., the one of assigning the components of a parallel application to the available computing resources, is one of the key points to be faced.

To provide full transparency and let the user disregard any issue related to allocation, fully automated tools can be implemented and integrated within an environment. They are in charge of controlling the execution of a parallel application and deciding, on the basis of an automated allocation policy, the allocation of both newly created entities and the migration of existing ones. A drawback of an automated approach is that it is usually able to move only toward general purpose goals and disregards the peculiar application needs: different application can have different requirements in terms of their resource needs, and the same allocation policy may have different impact on different applications.

When the performances achieved by automated allocation tools are not satisfactory, one can sacrifice transparency and furnish the user high-level and user-friendly tools that give him/her the capability of driving the allocation of the components of his/her parallel applications. However, the allocation of a parallel application cannot always be effectively decided before its execution: dynamic applications exist whose behaviour - in terms of resource needs - cannot be predicted and whose allocation can be effectively decided only at run-time. In this cases, because a dynamic intervention of the user is highly intrusive and, then, not tolerable, a customised dynamic allocation policy must be somehow codified within the application. More precisely, the programming environment can provide facilities, in the form of either **system libraries calls** or **language constructs**, to provide access to the allocation mechanisms from the application level and with which to dynamically influence the allocation of the components of an application.

Over the last decade, several systems have provided facilities that enable programmers to implement application-specific dynamic allocation policies. A variety of solutions have been adopted in different and heterogeneous systems, such as process-based system, object-based systems and distributed shared memory ones. Obviously, different systems define different computational models and the entities of which the allocation must be decided are different and can be of different granularity. Nevertheless, a few design issues can be outlined that permits to characterise those different approaches. In particular:

- the degree of **architecture independence** with respect to the target architecture and, consequently, their capability of not mining the portability of the application;

- the **expressive power** of the available facilities, i.e., their capability of giving the user full power over the allocation of a parallel application onto a given architecture;
- the capability of the facilities of **enforcing modularity** and encapsulation in the definition of the allocation behaviour of a parallel application;
- the degree of the integration of the offered facilities with the **run-time system** of the environment and, if present, of the automated allocation policy.

The paper surveys and analyses, with respect to the above issues, several systems providing some kind of facilities: without the ambition of coming to a precise taxonomy, the paper outlines important similarities and differences among the facilities offered in different systems. In addition, the paper highlights the major trends in the stated research area and points out open problems.

The paper is organised as follows. Section 2 analyses the allocation problem in its general terms. Section 3 describes the automated approach to allocation and outlines its limits. Section 4 introduces the user-driven approach to allocation and discuss its main design issues. Section 5 surveys the existing environments that follows a user-driven approach to allocation. Finally, section 5 provides a discussion and outlines open issues and the future directions of research.

2. Background Definitions

A parallel application defines a set of logical components in need of execution and of interact/communicate each other. To execute on a parallel architecture the application must allocate its components onto the available physical resources of the systems, i.e., its nodes. In its general terms, the allocation problem consists in finding a wise physical resource - logical component assignment to achieve [EagLZ89]:

- **efficiency,** i.e., a wise exploitation of the available system resources.
- **high-performance**, i.e., the best speed-up that application characteristics and available resources allow.

The first goal is a systems oriented one and may tends either to avoid the presence of unexploited resources within the system (**load sharing**) or, more ambitiously - to equalise their exploitation (**load balancing**) [ShiKS92]. The second goal is more application-oriented: the allocation must take into account the resources needs - both the execution and the communication needs - of the application components.

Depending on the time at which allocation decision are taken, one can distinguish between static and dynamic approaches [CasK88].

In the **static approach**, the allocation of the components of a parallel application onto a target architecture is decided before the execution of the application itself. It is clear that a static approach to allocation is suitable only for those applications whose behaviour can be accurately predicted at compile time [NorT93, CPE95a]. Whenever the behaviour of an application is dynamic, i.e., its behaviour cannot be predicted at run-time, no effective allocation decision can be taken before the execution. In those cases, a **dynamic approach** to allocation must be adopted [CPE95b]: the allocation of the components of a parallel application can be changed at run-time, depending on the current state of the execution. One the one hand, the allocation of each component is decided only when it begins its active life within the system (i.e., when it is dynamically created). On the other hand, if the **migration** mechanism is available [Smi88, Nut94], the allocation of already allocated entities during their life, by moving them from node to node. In addition, **dynamic replication** mechanisms may allow multiple copies of the components of a parallel application to be generated on several nodes of the systems [Bal92].

3. Automated versus User-Driven Allocation Tools

In the parallel computing area **transparency** is the property that makes users unaware of concrete - architectural - issues and it has always been considered as a main requirement to diminish the intrinsic complexities of parallel programming. With regard to the allocation problem, transparency is to means that users should only concentrate on the logical structure of his/her applications, letting the system automatically decide about the allocation of its applications onto a target architecture.

3.1 Automated Allocation Tools

To provide full transparency, a parallel programming environment can integrate **automated tools** for dealing with both the static and dynamic allocation problem. In an automated approach to static allocation, it is the job of the compiler to extract from the application code all the information needed and to map it onto the target architecture on the basis of a **mapping** algorithm [NorT93]. Static code analysis [BacDP94] and off-line monitoring [Mar90] can produce a large amount of useful information about resource exploitation of the application components. In an automated approach to dynamic allocation, the application behaviour is monitored at run-time and then, on the basis of the dynamically obtained information, the allocation of the application components is decided by an **allocation policy** in charge of dynamically issuing the needed allocation **mechanisms** [CorLZ93].

The problem of an automated approach to allocation is that it makes necessary to **fix an application independent goal** to be achieved by the allocation policy. In the past few years, many research efforts have been devoted to the study of both static and dynamic allocation policy with the goal of either load sharing or load balancing [EagLZ86, SheS92]. In addition, several systems and programming environments have integrated automated allocation tools based on a fixed load balancing/sharing policy [PowM83, BalK93]. As a consequence of the fixed goal of the policies, all of the above proposals are more system-oriented rather than application-oriented: the same policy applies to all applications by taking into account only the effects of the applications on the system state rather than application dependent information.

This system-orientation cannot grant the automated policy to grants the highest performance for all kinds of applications. Instead, its decisions could also worsen the application performances. For example, because the components of a parallel application usually needs to interact each other, an automated policy could allocate on distant nodes, with the aim of load balancing, heavily communicating entities, making them incur in high communication costs and, finally, slowing down the entire application [CorLZ94]. Moreover, assigning to the entities situated on the critical path of an application the same amount of execution resources assigned to the other application components can make the whole application execution time increase [ShiWP90].

Static allocation policy - because of their off-line and non intrusive nature - can somehow take into account, in their decision detailed information about the execution and the communication behaviour of the application components. Nevertheless, one cannot grant this information will be wisely exploited by the mapping algorithms without clashing with the peculiar needs of the application. The situation is even worse in dynamic allocation: because of the on-line and intrusive nature of dynamic allocation tools, it is too expansive to collect detailed information about the dynamic behaviour of application components and to take them into account in the allocation decisions. Thus, only a very little amount of application dependent information can be collected and exploited in allocation decisions.

3.2 User-Driven Allocation Tools

When the performance provided by the allocation of automated tools is not satisfactory, one might decide to sacrifice transparency in order to obtain higher performances. The user, in this case, is given the duty of studying its applications' behaviour, and somehow driving its allocation.

The off-line nature of static approaches to allocation can make it possible to implement tools with which the user is allowed to specify, before the execution of an application, where its components must be allocated. Not only the user can decide the allocation from scratch, but he/she could also check the decision taken by an automated mapping algorithm or constraint its behaviour, in order to tune the mapping to the peculiar needs of an application. In the last few years, several high-level visual environments have been implemented to guide users in this work [Bru93, Bro95].

Taking into account the dynamic allocation needs of an application introduces different problems. Because of the on-line nature of the decision to be taken during execution, one cannot think to leave to the user the duty of directly detecting its applications' behaviour while they are running and deciding about the allocation of their components: this interaction would lead to an intolerable overhead the user cannot be prompt enough to make his/her decision effective. Such an approach is adopted, for example, in the V System [The85] - the user can decide with a shell command about the migration of its running processes - but it is clear that it can be effective only in very few cases, such as very long running (with respect to the user reaction time) tasks and very slow changing situations.

3.3 Allocation Code

To allow the execution of a parallel application to be dynamically managed with regard to its peculiar allocation needs a different approach is necessary: a parallel application must contain in itself not only the algorithmic code but some kind of **allocation code** too. The allocation code must allow users to specify the allocation needs of its application components or, that is the same, to build a customised dynamic allocation policy.

To grant the user the capability of controlling the allocation of its application component, the run-time support to the environment must permit the user to access to the low-level allocation mechanisms available implemented within the run-time support to the environment. This capability can be exploited in the form of:

- **library calls**, with a well defined interface, that give direct access to the allocation mechanisms available in the environment [Zhou93, Tar94];
- **language constructs** whose execution has the effect of influencing the allocation of the application onto the architecture [Ach93, CorLZ96].

In both cases, the allocation code is no more transparent to the programmers that, instead, has the capability of building, by means of the allocation code, its own dynamic allocation policies tuned to the peculiar needs of their applications.

4. Design Issues in the Allocation Code

In the last few years, several systems and programming environment have appeared that provide users the capability of somehow customising the allocation of their applications onto a given architecture. However, under this common feature, very different approaches have been adopted for integrating allocation code at the application level.

The distinction given in the previous section between library calls and language constructs it is not the most significant one: in many cases a sequence of language construct is simply translated by the compiler a sequence of library calls to the run-time support. Instead, other issues can be outlined that most significantly characterise a given proposal in the area. In particular, the paper focuses on the following issues:

- **abstraction**, i.e., the degree of architecture independence the allocation code is able to furnish and, consequently, its capability of making applications architecture independent and portable;
- **expressive power**, i.e., the capability of the allocation code of granting the user full control over the allocation of the components of its applications;
- **modularity**, i.e., the capability of defining self-contained and reusable allocation code;
- **integration with the run-time support**, i.e., the degree of control the run-time support has over the of allocation code system of achieving its goals in co-operation with the presence of automated decisions with the user-defined allocation policy.

The following of this section precisely defines the above terms and to shows the advantages and the drawbacks that different solutions provide.

4.1 Abstraction

Due to the the lack of a dominant parallel or distributed architecture, a major requirement for any proposal in the parallel programming area is the capability of being easily portable across different architectures and of discarding programmer from the duty of managing low-level architecture-dependent details. Also allocation code proposals must this issues: the allocation needs of a parallel application must be expressed in abstract architecture-independent terms, that are likely to assume - transparently to the programmer - different concrete meanings on different architecture.

As a trivial example, let us suppose a component of a parallel application has the need of intensively accessing a resource allocated on a given node of the system. On the one hand, a non portable approach would specify the “physical” node onto which that component should be allocated. On the other hand, a portable approach would let simply specify to “co-locate” the interested component of the application and the accessed resource.

A possible drawback of an architecture-independent approach is that it can sometimes avoid a very efficient exploitation of the resources of target architecture, because of the introduced abstraction layer.

4.2 Expressive Power

By recalling that the reason for writing application-specific allocation policies is that a single policy in the run-time system will not be suitable for all applications, an environment that provides tools for writing application-specific policies must grant these tools be powerful enough to express a wide range of policies.

In the allocation area, it is very difficult to understand and quantify the power of either a set of language construct or of library calls. Anyway, expressive capacity is obviously proportional to:

- the amount of information available about the state of the system and of the application components;
- the available allocation mechanism, e.g, remote creation, migration and replication;
- the capacity of directly issuing the above mechanisms;

A drawback of having very high expressive power in the definition of allocation policy is the increase of complexity that can derives from leaving the user full power over the allocation of its application.

4.3 Modularity

The modularity principle enforces separation of concerns and tries to build encapsulated and reusable modules. In the analysed allocation area, modularity express the capability of:

- clearly **separating** the allocation code from the algorithmic code of an application;
- **confining** the area of influence of the allocation code for a particular application or component of an application.

With regard to first point, the code that implements the core functionality of an application serves a different purpose to the code that implements its allocation/migration policy; hence, if separation of concerns is not enforced and, instead, different kinds of code are mixed together, this is likely to increase the complexity of an application, thus making it more difficult to write and maintain. The same separation principle is encouraged, for example, in the area of synchronisation for concurrent programming [CorL91, McH94].

The second point relates to reuse. The allocation code that is associated to an application or to the components of an application somehow becomes part of the behavioural description of this components. Then, to permits reuse, encapsulation and context-independence must be a characteristic of the allocation code. In other words, the specified allocation behaviour should not be influenced by any other external entity: that would be considered as a violation of the encapsulation principle and would not more guarantee that the same expected behaviour in different context. Conversely, the allocation code associated to a given component must not be allowed to influence the allocation of any external entity.

A drawback of a modular approach it can sometimes limit the expressiveness of the allocation code, by constraining the capability of directly issuing the available allocation mechanisms. For instance, the allocation of one component cannot be influenced by any external entity the same way an instance variable of an object cannot be directly accessed by other instances. Nevertheless, we claim this limit can also permits to lower the complexity of the allocation code.

4.4 Integration of System-level and Application-level Policies

The starting point of this paper is that an application-specific allocation policy is likely to perform better than a general-purpose allocation policy in the run-time system. However, leaving to the user the duty of managing the allocation of its parallel application from scratch and without any help from the run-time system may not to be the optimal solution. An alternative solution can integrate an automated allocation policy and some kind of allocation code, conceived with the purpose of customising its automated behaviour.

A first advantage that can come from this integration is that the presence of an automated policy does not oblige the user to specify the allocation requirements of his/her application. If no directives about the needs of its applications are defined, the system policy will perform anyway some kind of dynamic management on the allocation of the parallel application on the system, even if not tailored to the exact needs of the application. A second advantage relates, again, to a clear separation of concerns that an integrated approach would allow. Policies implemented in the run-time system usually make allocation decisions based on load information about the current state of the system. Policies implemented via language-level constructs make allocation decisions based on application specific information too. By integrating the two aspects, one can allow user to disregard - unless he wants to - aspects related to low-level architectural details, such as the load of the nodes in the system, and to concentrate only on the application-specific aspects.

An apparent drawback of an integrated approach is that it breaks the modularity principle. In fact, the automated policy represent an external entity allowed to influence the allocation of all the entities in the system. Moreover, an integrated solution may introduce conflicts because the allocation decisions of the automated policy may be in contrast to the directives coming from the allocation code. In spite of that, we claim a clear separation of concerns still arises: the allocation code can be defined only on the basis of application-level information while policies implemented in the run-time system take allocation decisions based on low-level load information about the current state of the system. Only by allowing an external entity to exploit this low-level information one can grant architecture-independence and ease of use without loosing in efficiency.

5. Survey of Existing Systems

Various systems offers tools at the application level to allow the customisation of the allocation of parallel applications. While the general goal of any allocation code definition is the achievement of high-performance in application, different systems, defining different computational model, may need to face the allocation problem in different way. In particular, high-performance may be obtained by achieving:

- **load balancing**, i.e., the assignment of the execution resources to the application components in terms of their execution needs;
- **locality of references**, i.e., the minimisation of the time spent in the interactions between the components of the applications by granting their neighbourhood with respect to a given architecture.

As a consequence, the semantic of the allocation code can be very different, depending on the main goal to be achieved, either load balancing or locality. In particular, we will separately analyse the solutions presented in:

- task-based systems [ShiKS92];
- object-based systems [ChiC91];
- distributed shared memory systems [NitL91, ProTM96];

Task-based systems defines an environment of usually coarse-grained and loosely interacting - or not interacting at all - processes [EagLZ86, ShiKS92]. Then, allocation mainly focuses on execution resources and **load balancing** is usually the main goal to be followed, with respect to the peculiar execution needs of single processes.

In **object-based** systems [ChiC91] the computational units are communicating objects, that include in themselves both communication and execution needs. Then, not only load balancing but also locality of references are goals to be followed to guarantee high

In **distributed shared memory** systems [ProTM96] the interactions among the executing entities are related to the access to a logical memory space, globally shared among all the application processes but physically distributed among the systems nodes.

In those systems, a process that access intensively a given page of memory should usually be co-resident with them, so to avoid the high costs associated to remote memory accesses.

5.1 Task-based Systems

In task-based systems the offered facilities for allocation mainly on writing process allocation policy with the goal of granting executing processes all needed execution resources, disregarding communication related issues.

The key idea in those systems, is to leave to a kernel layer both the implementation of the mechanisms for dynamic load balancing (for examples remote creation and/or migration) and the duty of collecting information about the state of the system nodes. Instead, allocation policies can be defined by the user by accessing to the services provided by the kernel via system calls (see figure 1).

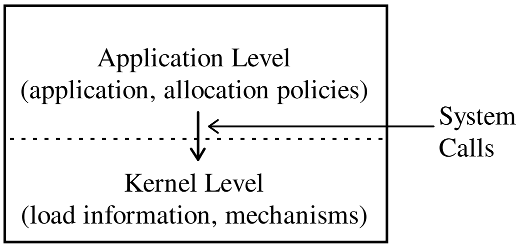

Figure 1. The general approach of task-based systems

5.1.1 Charlotte

Charlotte [ArtF89] is one of the first systems that provides the capability of writing user-level allocation policies by accessing to the services provided by the kernel layer.

For example, the Statistics() call tells the kernel to initiate collecting information about the state of a node and to place them at a given memory address, for allowing access to these information; the MigrateOut() call makes a process initiate a negotiation with another node toward the possibility of issuing a migration. Note that the above system call - as many other allocation-oriented system calls defined in Charlotte - requires to specify, as parameters, the names of the nodes of the systems onto which to migrate a process. This concreteness mines the portability of the Charlotte approach.

An allocation policy, in Charlotte is not specific to any particular application but it is system-wide: if several applications run in the system then all of them are subjected to the same allocation policy currently linked in the system (see figure 2). The background willing in the implementation of Charlotte is experience and easily test several general-purpose allocation policy, rather that writing application-specific policies. As a consequence, Charlotte does not give the user the expressive power of explicitly controlling single application components (see figure 2). In addition. The modular principle cannot apply, since there is no possibility of tying a policy to a given application and to preserve the allocation behaviour of the application.

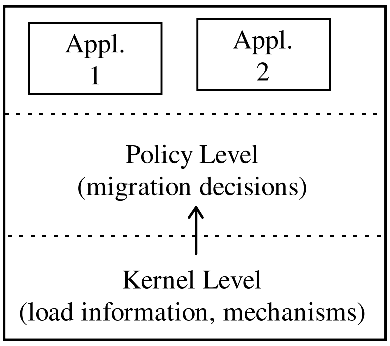

Figure 2. The Charlotte approach

More recently proposed systems have adopted Charlotte's idea of separating kernel-level mechanism from user-level policy, but with the idea of supporting application-specific rather than system-wide policies.

5.1.2 Utopia

The **Utopia** system [Zhou93] adopts an approach to writing user-level allocation policies similar to the Charlotte one: a clear separation is defined between the kernel level - devoted to the implementation of the mechanisms and to the collection of system state information - and the application level - that access to the kernel via system calls. However, Utopia presents some characteristic that makes it more application-oriented.

A system call is provided in Utopia (ls_placereq()) that makes it possible to specify the needs, in terms of execution resource, of a new task to be created[1]. A kernel-integrated load balancing policy decides the allocation node of a newly created process on the basis of its execution needs, if any is specified, and of the current load of the system nodes. The presence of the integrated load sharing policy makes it possible to completely disregard any allocation issue and let the kernel act autonomously in its decisions. In addition, the user can explicitly decide the allocation node of a given process (lsrexec(node)). As in Charlotte, the need - in using the system calls - of explicitly referring to given system nodes makes the approach not portable.

In Utopia, two types of applications can be defined that exploit the above system calls: load balancing applications and indirectly load balanced ones. Load balancing applications (called direct applications) make explicit use of the kernel system calls to implement a load balancing policy in themselves (i.e., to decide where to create a new process within the application). In this case, since load balancing application contains in themselves their own load balancing policy, the allocation code is mixed with the algorithmic code and no separation of concerns is provided.

In addition, Utopia users can write direct applications in charge of deciding the allocation of the so called indirect applications - typically simple tasks - that completely disregard any allocation related issues and, instead, use load balancing applications as their execution context For instance, the load balancing application filter all task creation commands within a given application and decides their allocation before effectively commanding their creation to the kernel. In this case, the modularity principle is enforced: the load sharing policy is tied to the indirect application that use it and can only decide about the allocation of the components of this application.

[1] Utopia rules out process migration

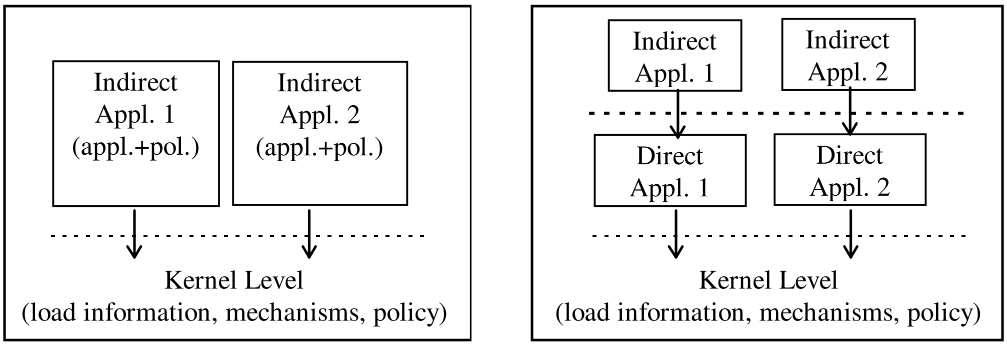

Figure 3. The Utopia approach
(left: direct applications, right, indirect applications)

5.1.3 Dynamo

The **Dynamo** [Tar95] system integrate aspects of both Utopia and Charlotte.

As in Charlotte, allocation policies are clearly separated from the applications that exploits them and may exists independently of them (the same is not true in Utopia, where a direct application has not meaning if unrelated to the indirect application that defines it). As in Utopia, allocation policies are linked to applications rather than to the whole system (see figure 4) and they can exploit application dependent knowledge. For example, the dai_MakeTask(type, loc, size, priority, cost) call command the policy to create a new task and include as parameters information about the resource requirement of the task so to allows the policy to decide the best possible allocation for these new task.

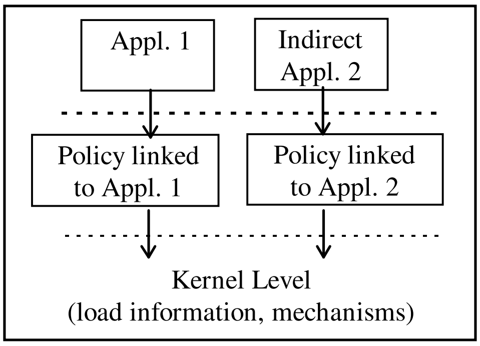

Figure 4. The Dynamo approach

The Dynamo approach is interesting since it allows to write application-specific allocation policies in a modular way, by allowing reuse of a load balancing policy in different applications and to easily replace the load balancing policy used in an application with a different policy. In addition, Dynamo provide a large number of system calls to be exploited for specifying the peculiar needs of the application components and for writing the load balancing policy, thus granting a high expressive power. However, to our opinion, that may cause the definition of a load balancing policy in Dynamo to be a complex and low-level task. Another drawback of the Dynamo approach is that it lacks of any automated load balancing policy to be integrated within the system and to co-operate with the user-defined one.

5.1.4 Charm

The Charm system [KalK93, KalS94] uses C as a base language and extends it with constructs to support distributed programming.

The Charm approach to allocation is similar to the Charlotte one, in the sense that it allow different policies for the allocation of the processes to be linked within the run-time system, by choosing them in a library of predefined allocation policies. However, in general, Charm allocation policies do not take into account application-specific allocation needs. An exception that integrates the load balancing policy with application-specific information is described in [SinK93], with regard to parallel branch and bound applications. Each process (representing one possible solution of the analysed search space) is dynamically assigned a “mark” proportional to the quality of the solution he is currently examining. This marks influence the behaviour of the integrated load balancing policy that, instead of merely balancing the execution load within the nodes of the system, assign each process an amount of execution resources proportional to its current mark. If this is not done then some nodes may end up with a scarcity of high-priority tasks and thus will spend time executing low-priority tasks (which are unlikely to contribute towards the search for a best solution). In effect, nodes with a glut of low-priority tasks end up doing “useless” work and this can limit the speedup gained by parallelising the application.

Another interesting characteristic of Charm relates to the presence of annotated shared variables, and it will be discussed later, in the section devoted to distributed shared memory systems.

5.2 Object-oriented Systems

Several proposal in the concurrent object-oriented area aims to define facilities for customising the allocation of objects in parallel/distributed object-based applications. Differently from task-based systems, these proposals mostly concentrate on communication rather that on load balancing: the implicit assumption is that in an environment of communicating entities high-performances can be achieved by exploiting locality of references.

As a general rule, in all proposal toward the user-defined allocation of parallel object-oriented application, the allocation code becomes part of the definition of a class. In fact, the application components are instances of a class that describes their behaviour (state and instances and, eventually, synchronisation of concurrent activities) and the dynamic allocation needs of the components have to be considered as part of the behaviour itself. In addition, most of the proposals introduce peculiar language constructs to specify the allocation code instead of exploiting a set of library calls provided by the kernel.

Apart from this common characteristic, very different approaches have been undertaken in the below presented systems.

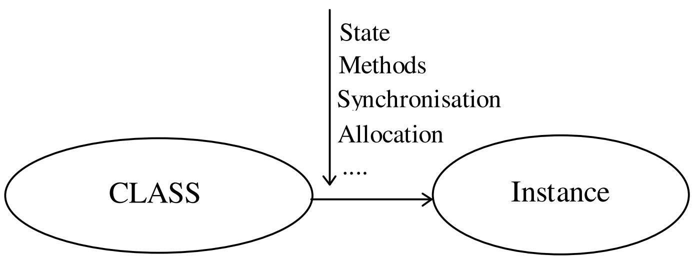

Figure 5. The allocation code as part of a class definition

5.2.1 Emerald and friends

If Charlotte firstly introduced the concept of user-defined allocation policy within a task-based system, **Emerald** [Jul88] is one of the systems that firstly introduced language-level constructs for the purpose of dynamically allocating objects of a parallel applications.

Emerald constructs for allocation can be either inserted as instructions in the code of the instance methods or as annotations in the definition of an instance variable (see figure 6).

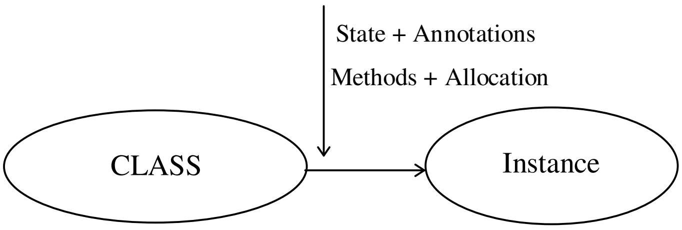

Figure 6. The Emerald approach

With regard to the former case, Emerald defines several constructs for object mobility (i.e., migration)[2], that, when executed, have the effect of making the corresponding mechanism apply. More in detail, the move construct can be used into the body of a method to migrate an object to a given location, i.e., to the node of residence of another object. For example, the instruction:

```
move X to locate Y
```

provides, when executed, to move the object referred by X to the node where the object referred by Y currently resides. The move primitive can also be used in parameter-passing when invoking an operation upon an object. For example, the following statement:

```
foo.SomeOperation(move X)
```

tells the compiler that the parameter (in this case X) should be migrated to the node where the object referred by foo is located. In Emerald parlance this is known as *call-by-move*. Another keyword, visit, can be used instead of move for parameter passing. In this case, the parameter will be migrated to the destination node for the duration of the call and moved back upon termination. Call-by-move and call-by-visit are provided in the language as both a convenience to the programmer (they avoid the need for explicit move statements before a call) and as a performance optimisation (they permit the compiler to pack the migrating objects in the same network packet as the invocation).

The last mobility concept that Emerald supports is that of *fixing* which makes an object immobile. If an object is fixed at a node then it cannot be migrated to another node, unless it is later unfixed and thus made mobile again or refixed to a different node. Thus, *move* and *visit* directives will be ignored by the run-time system if they are applied to an object which is currently fixed. One reason why an object may be fixed at a node is if it is accessing resources specific to that node. More in detail, the syntax used in Emerald for fixing, unfixing and refixing are as follows:

[2] Instead, there is no way to explicitly specify on which node an object should be initially created: newly created objects are allocated on the node where their creation has been issued and only successively migrated.

```
fix X at Y
unfix X
refix X at Y
```

Objects do not live in isolation but may contain references to other objects, e.g., an object may be the root node of a graph or a linked list. When such an object is being moved to another node, it may be appropriate to move the entire graph/linked list, so to grant locality. The *attached* primitive, used in the definition of instance variables, is provided for this purpose. For example, the following definition:

```
attach var foo: type_of_foo
```

specify that the foo instance should always follows the movements of the objects in which it is declared. Whenever the object into which this declaration is contained migrate, the object foo migrate too, along with any other object eventually attached to foo.

Emerald construct are very powerful and makes it possible to express a great variety of allocation behaviours in applications. In addition, if compared with the above presented task-based systems, they are highly portable: the allocation code of the application components is expressed in terms of the allocation of other components, without forcing (though allowing) the user to refer, say, to a physical node of the system.

However, in spite of these advantages, Emerald way of dealing with the allocation of application components presents several characteristics that we consider quite odd in an object-based system. Firstly, the allocation code is mixed with the sequential code of methods definition and with the definition of instance variable. Thus, no separation of concerns is provided and the user is forced to define its class by taking into account, contemporary, algorithmic issues and allocation issues. To our personal opinion, this is likely to greatly increase the complexity of parallel applications. In addition, the introduced allocation constructs tends to violate the encapsulation principle. In fact, the allocation annotations inserted in the code of a class does not specify the allocation behaviour of the instances of the class itself (i.e., of the object that is currently executing the primitives) but they can command the migration of other instances to which the reference is hold. This happens, for example, in Emerald, where an object can be moved around in the system being commanded in that by the clients that are currently refereeing it or by objects that want to be attached to them.

As a final remark, in Emerald all responsibility about allocation is left to the user and there is neither any system-level allocation policy to check or integrate the user-inserted allocation directives nor any way to access to information to the current system state (for example, the load of the nodes). Only a few static allocations decisions can be autonomously taken at compile time: for example, the compiler will generate code to move a parameter if it is a small and immutable object, e.g., an integer, since such objects can be copied cheaply.

The Emerald approach has influenced many other object-based systems such as Amber, Distributed Smalltalk and Trellis/DOWL.

Amber [Cha89] shapes its object mobility primitives after those of Emerald. It offers facilities to move an object via the MoveTo primitive, to Attach one object to another and to Unattach an attached object. A peculiar characteristic of Amber is the capability of marking an object as *immutable*: in this case, invoking MoveTo on it it will *replicate* the object rather than move it, thus allowing better availability without incurring in any consistency problem.

Distributed Smalltalk [Ben90] provides the move and copy instructions to *migrate* and *replicate* an object on a given node, respectively. In addition, it provides the ability to optionally state the node at which an object is to be created.

The **Trellis/DOWL** system [HeuA89, Ach93] offers the same basic object mobility constructs of the Emerald system and, again, mix them in the algorithmic code and do not integrate them with any automated allocated policy. However, the semantic of the constructs substantially differs from those of Emerald. More in particular, the Trellis way of expressing allocation directives is a step toward encapsulation: allocation constructs, in Trellis, are introduced to specify the dynamic allocation behaviour of the object onto which the constructs are specified, without influencing the allocation of other objects, say of objects to which the reference is held.

For instance, in Trellis each object has instance variables called location and fixed_at. Assigning to these variables cause an object to migrate to, or to become fixed at, a location: since instance variables can be assigned only from the methods of the objects itself, no other components within an application can decide of migrating or fixing an object to a given node. Analogously, the attachment is specified by means of an instance variable that specify to which other objects an object must be attached to (in the form of a head and tail list). However, differently from Emerald, in Trellis it is an objects that declare its attachment to another one, and so deciding of following it in its migration and not viceversa.

Trellis's move and visit primitives serve basically the same purpose as they do in Emerald but, again, they are used in a different way: they move and visit primitives do not appear in the *invocation* of an operation, as in Emerald, but rather in the operation's *declaration.* For example:

```
operation foo(a: visit someType, b: move anotherType) ...
```

It is clear that in this case the a and b parameters are influenced in their allocation by the objects that uses them as parameter of a service request. However, the allocation behaviour is somehow a bit more context-independent, because it is the same for any invocation of the same request.

5.2.2 Reflective Systems

Conventional computing system work on data structure to model something external to the system itself. Reflective systems introduce a **meta computing level** to model the computation itself [WatY88]. Reflective systems delegate any issue related to the management and the optimisation of applications' execution to the meta-level that is capable, on the basis of the current state of the computation, of influencing the behaviour of applications. Programming in a reflective system means not only to write the usual algorithmic code (the so called base-level) but also to write the meta-level code: that way, the algorithmic part of the code is completely separated from the management part.

With regard to concurrent object-based reflective systems [WatY88], each base-level object has an associated meta-level object, in charge of managing its communication and synchronisation. A natural extension is to charge meta-level objects of allocation related problems too [Mas94, OkaI94, Lux95], such as locality of reference, load balancing, management of prioritised scheduling policies (see figure 7). This allows the user to write its own allocation policies for the objects of the system.

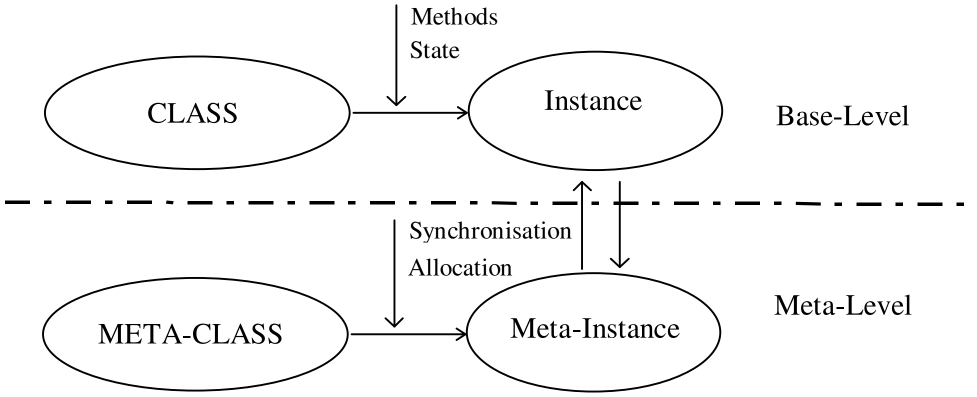

Figure 7. Allocation in a Object-Based Reflective System

Programming the meta-level the user has access to all the information usually exploited by automated allocation tools, such as the state of the hardware resources, the current load of the nodes, the location of the objects in the systems. In addition, from the meta-level, the user can directly control the location of the object systems, by commanding migrations.

In the **AL/1D** system [OkaI94], a meta-level object is associated to each object of the application. The meta-level object contains in itself, as instance variable, data to describe the computation itself. Examples of these variables can be load information about the state of the node and the location of other objects in the system. Operation are provided at the meta-level and issued by events at the base-level. Writing a meta-level object is writing an allocation policy: depending on the events occurring at the higher level, the meta-level can react by issuing specific actions such as migrating the object. For example, a method of the meta-object can be defined to react at any message send action. If the receiver of the message is the object bar then the sender is migrated toward the receiver, as shown in the below examples (with a simplified syntax):

```
meta_object Foo()
.........
instance_var localHost; /* node of residence of the base-level object */
......
method event_message_sent(message, receiver)
        local_variable receiver_location;
        ......
        if (receiver_location <> sender_location AND receiver = bar)
            migrate(sender, receiver_location)
        ......
```

The **Distributed Memory Reflective Architecture** system [Mas94] enlarge the meta-level capabilities by introducing different meta-level objects apart from the one associated with the objects of the application. In particular:

- a meta-level object called scheduler, manages the scheduling of the objects in the node
- a meta-level object called node manager describe the computational state of one node of the system and can manage all other meta-level objects allocated in the node;

- a meta-level object called class manager can be associated to each class of the system to manage the resources shared by all the object within a class.

A well confined problem is faced in the **BirliX** system [Lux95]: not only the allocation policy for a given object could be in need of customisation, but the migration mechanism too can be customised, by means of the meta-level, to allow grater flexibility and efficiency.

Important characteristics of reflective systems are the clear separation of concerns provided by the meta-level - that confines the allocation code without mixing it with the algorithmic code - and its encapsulation - that makes the meta-level influence the allocation of the associated instance only.. In addition, reflective system gives the user full power of programming customised allocation policies for their application. Not only the behaviour of specifying the behaviour of single instance is given, but also the capability of building load balancing policies to manage groups of objects and that exploit the system-level information - i.e., the current status of the hardware resources - accessible from the meta-level.

In spite of this flexibility, the definition of the meta-level is prone to be low-level and, then, less portable and difficult to be written.

5.2.3 Parallel Objects

The Parallel Objects programming environment, based on the active object model [CorLZ94a], faces the allocation problem by integrating a system-level allocation policy with a set of *high-level directives* for specifying the peculiar allocation needs of application objects.

A load balancing policy in the Parallel Objects run-time supports automatically decides about the allocation of newly created objects and about the migration of already allocated one, depending on the current system load. Though capable of autonomous decisions, the policy is forced to act in agreement with the allocation directives, so to adapt its behaviour to the peculiar allocation needs of applications (see figure 8). In this sense, the Parallel Objects allocation directives tune the automated policy to the peculiar allocation needs of the application objects. Depending on the user skill with the allocation problem, the allocation directives could be more or less constrainable. At one extreme, the user can totally ignore them and let the allocation 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.

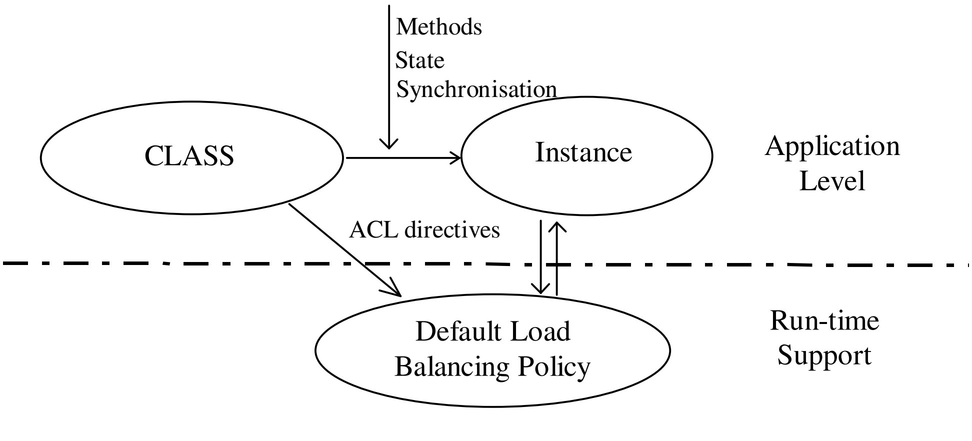

Figure 8. The integration of the ACL directives with the Parallel Objects support

Collectively, the Parallel Objects directives are called **ACL** (abstract configuration language) [CorLZ94b] and may refer either to general allocation properties of the objects eventually in relationship to other object in the system or to the allocation of the components internal to an object (a peculiar characteristic of Parallel Objects is that objects are not considered as the basic units of allocation but, instead, their state and the activities devoted to the services of the request can be distributed onto several nodes). Examples of ACL directives are:

- distributed(#nodes): allows an object to distributed its components onto several nodes;
- partitioned(partition_scheme); allows an object to partition and distribute its state on the basis of a given partition scheme;
- migratable: allows an objects to be migrated around the system;
- close_to(O1): specify the need of an object to be allocated near the object O1;
- neighbour(O2): specify the need of an object to be allocated near the object O2 but not on the same node, so to allow them to execute in parallel without competing for execution resources.

It is important to note that ACL directives can be parametrised and related to the current state of the object. For example, an ACL directive can specify the locality need of the instances of a class in relationship with an instance whose reference is maintained in an instance variable. Since the refereed object con change in time, ACL directives change.

ACL directives are claimed to be abstract (and portable) since they express logical rather than physical concepts related to the allocation of its application components. Only at compilation time these concepts assume concrete meaning, depending on the particular characteristics of the target architecture. For examples, the "closeness" concept can maintain their validity over a large spectrum of architectures but assume the meaning of "coresidence" in a distributed environment, the meaning of "on the same cluster" in a clustered one.

In addition ACL directives enforce both separation of concerns and encapsulation. One the one hand, they are specified in a special section of the class definition, as follows

```
PO_CLASS foo
    STATE_section
    /* state definition */
    METHODS_section
    /* methods definition */
    ACL_section
            /* list of ACL directives */
END foo
```

so to clearly separate them from the algorithmic code. One the other hand, even if ACL specifies the allocation needs of an objects in terms of the allocation of other objects, the run-time support does not interpret them as transitive and do not perform any allocation actions on one instance but the ones that agree with its ACL directives.

A limit of the Parallel Objects approach with respect to Emerald-like and Reflective system is the limited expressive power that high-level directives may provide. However, it is our feelings that the presence of an automated load balancing policy can supply this lack of expressiveness.

5.2.4 Amadeus

A peculiar approach to control object-allocation is implemented in the **Amadeus** object-based system [TanC92]. In Amadeus, the allocation of a parallel application onto the target architecture is based upon the concept of *cluster*, a group of objects that are managed as a unit. Any object, in Amadeus, belongs to a cluster. When a new object is created within in Amadeus it is placed in the “current” cluster, i.e., the cluster to which belongs the objects that create it. Since clusters are persistent entities that exists independently of the applications that exploits them, a cluster is said to be *unmapped* until it is still resident in persistent storage. As soon as an application requires the services of an unmapped cluster, it will be loaded from the storage and *mapped*, i.e., allocated, onto one node of the system.

The decision onto which node to map a cluster is taken by a load balancing policy integrated within he run-time system on the basis of current system load. With regard to the allocation, a cluster is *mapped* on one node at a time and, once it has been mapped to a node, it can migrate only if the user explicitly unmap it and then remap it onto a different node. Differently from the above presented object-based systems, Amadeus does not provide language construct related to the allocation of the clusters. Instead, a set of library calls are provided to change the cluster membership of objects and grant objects with a strong locality of reference to be placed into the same cluster. New clusters are created with the `StartNewCluster` system call and the current cluster of an object can be changed via the `ChangeDefaultCluster` system call. It interesting to note that portability is preserved in this calls: they do no longer refer - as in task-based systems - to concrete issues such as identifiers of the systems node but, instead, to abstract and architecture-independent locations, e.g., the clusters.

As in Parallel Object, the presence of a default allocation policy within the run-time system allows a user to completely disregard allocation related issues. Anyway, The Amadeus approach is poorer in terms of expressiveness than the above presented ones. At the current state of implementation, neither clusters nor objects within them can migrate. However, there are plans to relax these restrictions [Gou92] by means of a dynamic re-clustering algorithm which would permit migration of objects between clusters on the basis of the communication weight between objects. The proposed algorithm is designed in order to allow the user to dynamically influence it in two ways:

- by explicit co-locating commands, for example: `colocate(O1,O2,O3)`;
- by forcing the algorithms to consider a user-defined weight for objects’ communications;

5.3 Distributed Shared Memory Systems

In the area of Distributed Shared Memory Systems (DSM) the key to obtain high performances is locality of reference [ProTM96]: performance is strictly dependent on the amount of access to the shared memory that can be locally solved. Even in this case, a general purpose policy for managing the replication and the migration across nodes of memory pages cannot fit the best all possible application and data structures allocated in the memory. Thus, even in the DSM area, facilities are needed for customising the allocation of entities within the shared memory.

As a general rule, because of the passive nature of the entities (either shared variables, shared objects or memory pages, depending on the granularity of the memory model), the DSM approach to customise the allocation of different parts of the memory adds annotation to the definitions of the entities in the shared memory. This annotations, rather than explicitly specify where entities have to be allocated, specify the patterns of access to the entities by the processes of the application. It is in charge of the run-time system to exploit this information to optimise the allocation of the parts of the memory in the system.

5.3.1 Munin

The implementation of the Munin DSM system [CarBZ91] starts from the consideration that different shared variables can be accessed in different ways from the application processes. Munin permits programmers to declare the expected mode a process will access to a shared variables. These annotations are used as optimisation hints to the DSM system to help it in allocating variables to memory pages of memory and memory pages to system nodes and in issuing the appropriate coherence protocols. Accessing a shared variable in a manner incompatible with its annotation raises an exception.

Several annotations are provided that characterise different mode of access to an object. For example, the read_only annotation has the purpose of declaring a variable as immutable, thus allowing the run-time system to freely replicate it without incurring in any inconsistency problem. The migratory annotation specifies that a single thread will perform multiple updates to an object before another thread accesses the object (this pattern of access is common for variables that are accessed only inside a critical region); the run-time system will migrate such an object to the new thread. If the declaration of a shared variable does not include an annotation then a default annotation is assumed.

Since a shared variable may be accessed in different ways during different phases of a program's execution, Munin provides a ChangeAnnotation() call to dynamically change the annotation of a shared variable. It also provides some library routines that permit programmers to fine-tune various aspects of Munin's operation. For example, the SingleObject() routine can be used to tell Munin to treat a large, multi-page variable as a single object rather than breaking it into several page-sized objects.

The problem of Munin's annotated variables, is that the annotation is not strictly associated to the variable definition, but, instead, to the declaration made by the processes that need to use them, that are also allowed to dynamically change this annotation. Thus, neither separation of concerns nor encapsulation is granted.

5.3.2 Pooh

Pooh [JudC93, Jud95] extends to the C++ language with the presence of peculiar C++ class whose instances can be allocated in a DSM system and globally shared within an application. The shared annotation is used to specify DSM classes and it takes as a parameter the name of an *access mode* which tells the run-time system how to deal with the access to the instances of the class by the application process. Similarly to Munin, these annotations influence the allocation of shared instances in the system, in terms of their replication and migration.

As an example, the below annotated class SharedInt defines a ReadersWriter access mode, thus allowing each exported operation - annotated at their time - to be either a Read or a Write operation[3].

```
shared(ReadersWriter) class SharedInt {
public:
        // constructors
```

[3] The annotation syntax shown in the example is somewhat simplified but serves to illustrate the basic concepts of Pooh's annotations. The actual syntax for Pooh's annotations [Jud95] is more complex in order to accommodate some extra constructs which are of no relevance to the present discussion.

```
            access_mode(Write) SharedInt();
            access_mode(Write) SharedInt(int i);
            // operations
            access_mode(Read) int value();
            access_mode(Write) increment();
    private:
            int     number;
    }
```

Though Pooh annotations follow the pursue the same goal of Munin's ones, differences arises. One the one hand, while Munin annotations are spread out among the different processes that access shared data, Pooh annotations are part of the definition of a shared class, thus making them encapsulated. On the other hand, the access modes in Pooh are themselves classes. All access mode classes are subclasses of the base class AccessMode, defined in the standard Pooh library. Users - by subclassing from AccessMode - can define new access mode and add them to the system, thus making the system more flexible.

5.3.3 Other DSM Systems

Other DSM system exists that introduce annotations to specify how a shared data will be accessed and to consequently influence the actions performed on it by the run-time systems.

In Charm [KalK93], several new keywords for programmer-defined data-types; the presence of one of these keywords denotes that instances of that data-type will be shared/processed in a specific way. Using these keywords, variables and new data types may be declared that will be *read-only* variables, *write-once* variables, *accumulators*, *monotonic* variables or *distributed tables*. Because access to each of these data types is quite specialised, the compiler is able to optimise the implementation of each of them. For example, the run-time system will transparently replicate read-only and write-once variables.

In the Panda system [Ass93, Ass94], the tie and untie primitives can be associated to shared objects in order to fix and unfix their allocation to a given node of the system, thus avoiding page faults in the access to the object from that node. In addition, a co-location primitive can be invoked to specify that two or more objects needs to be always allocated on the same node. In Panda, differently from the above presented DSM systems, the annotation have the direct purpose of influencing the allocation, while in the other presented DSM systems they define the access mode to data and, only as a consequence, influence their allocation.

6. Open Issues and Future Work

The definition and the implementation of either tools or language construct to customise the allocation of parallel application is a quite new area of research. We feel several issues are open to be discussed by the scientific community and several other needs to be faced. This section tries to point out some of the issues the authors feel ought to be faced by future works in the stated area.

6.1 Inappropriate Use of the Offered Capabilities

Judicious application of application code can result in a significant performance increase. However, the use of allocation code is a double-edged sword, and if it is used inappropriately then it can degrade performance. For example, migrating an object close to the server it is currently accessing can increase performance if the number of access if high enough to outweigh the migration cost. Otherwise, that will result in a degrade of performance.

None of the systems surveyed in the paper introduce tools to evaluate the effectiveness of the allocation hints specified in an application. As such, it can be difficult to determine the appropriate places in a program where allocation hints should be used. Locating and fixing bottlenecks caused by inappropriate use of these hints can be a very tedious task [Ach93]. The development of tools to help determine suitable places to use allocation hints is an open research area.

6.2 Lack of Performance Analysis

The purpose of providing either language-level constructs or tools for allocation is providing better performance than obtainable with automated general-purpose policies implemented in the run-time system.

However, to give general validity to a work on these area performance figures are needed. Unfortunately, several of the surveyed systems neglect to provide performance figures, mostly focusing on the characteristics of the provided tools/constructs. Even when performance figures are available, they are reported only to prove that the performance of a given application is improved - under certain condition - by applying the presented tools/constructs.

We think a construct to be proved effective must be proved to be effective not with respect to their simple absence. Instead, it must proved to be outweigh the performance of several automated load balancing policy. It would certainly be useful if, in future, new proposal will provide a comparison of the relative performance of the proposal and of several automated allocation policies.

6.3 Performance Portability

A well recognised issue in parallel programming systems and environment is portability. Due to the lack of a dominant architecture in parallel and distributed computing systems, a major requirement for any proposal in the parallel programming area is the capability of being easily portable across different architectures.

Most of the surveyed systems specify allocation/migration policies in abstract, architecture independent terms. Thus, most of the proposals are not tied to a particular architecture and can be easily portable to other systems. However, only a few of the proposed systems have actually been ported to different architectures. Charm has been implemented on several, very different architectures. However, papers on Charm neglect to compare the efficiency of Charm programs across these different architectures. Other systems, such as PO, explicitly claim to be portable. However, again they describe and evaluate only one implementation. Thus, due to the lack of any comparative work, there is no way to evaluate the effect on performance of an allocation policy that is embedded in an application when the application is ported from one architecture to another. In other words, although the *functionality* of a distributed application may be portable across different architectures, its *performance* may not be portable.

We think there is a lot of work to be done in this area. The work can be tackled from two different directions. Firstly, there is a need to empirically test the performance of applications across different architectures in order to gain an understanding of which parallel programming constructs and dynamic allocation primitives/policies are efficient on different architectures. Secondly, there is a need to develop a formal model that can be used to *predict* the performance of a language constructs and dynamic allocation primitives/policies on different architectures.

6.4 Relationship with the Synchronisation Area

The area of allocation/migration overlaps with that of synchronisation for concurrent object-oriented languages in two important ways:

- An event-based programming model can be used for allocation/migration. For example, the underlying implementation of DSM in the Pooh language is event-based. Also, the interface to a load balancing policy is via a function which takes an event (a *message* in the parlance of Dynamo) as a parameter and dispatches it to an appropriate event handler. An event-based programming model can also be used for synchronisation. For example, the SOS paradigm [McH94] models synchronisation in terms of events.

- In a lot of the surveyed systems, allocation/migration primitives are applied to variables/objects which potentially can be accessed by several processes. Such concurrent access invariably implies the need for synchronisation. Indeed, Munin and Pooh take advantage of this by integrating synchronisation with DSM. However, not all systems do likewise. Emerald, for example, employs monitors for its synchronisation needs while the run-time system in Charm automatically provides mutual exclusion accesses to shared variables

The overlap between allocation and synchronisation suggests that researchers in the relatively young area of allocation might be able to borrow some useful ideas from the older research area of synchronisation.

6.5 The Inheritance Anomaly

A well-known issue in the synchronisation area is the *inheritance anomaly* problem, related to concurrent object-oriented programming languages.

This problem concerns a conflict between the inheritance of synchronisation code and the inheritance of sequential code [McH94]; this conflict can hinder code reuse. The problem can be briefly stated as follows. Inheritance involves not just code *reuse* but also code *change*. If a class contains two kinds of code - sequential code and synchronisation code - then a *change* to one of these kinds of inherited code can hinder the *reuse* of the of the other kind of inherited code (and vice versa). In the worst case, a subclass may not be able to reuse *any* inherited code. It has been argued that the inheritance anomaly is not confined to the domain of synchronisation but may appear whenever attempts are made to inherit two or more kinds of code within a single class hierarchy [McH94].

Inheritance anomalies have been reported even in the area of object-oriented real-time system with regard to deadline specifications [Askit94] and it is likely to represent itself in the analysed domain of allocation for object-oriented parallel programming environment. For example, the Pooh language [Judge94] disallows subclassing from a **shared** class, because of the conflict that may arise in its annotations. Also, Masuhara [Mas94] has pointed out that the mixing of load balancing code with sequential code can hinder code reuse. The PO environment claim the advantages of inheritance in the allocation behaviour but no examples to validate this claim.

To date, most of the analysis of the inheritance anomaly has been focused in the domain of synchronisation. Little is known about how serious the problem is in the area of allocation.

6.6 Toward Block Structured Environments

A **structured approach** to parallel programming starts from the idea that most parallel applications follow a few predetermined patterns of execution for the parallel activities and their communications. In other words, the components of a parallel application normally do not interact arbitrarily but follow regular basic patterns, a structured parallel programming approach introduces the abstraction of interaction pattern and describes parallel applications by composing and nesting these abstraction.

This belief has influenced the design of some parallel programming environment, e.g., P3L [Bac95] and LNA [CorLZ94d], which provide constructs that directly support the writing of parallel applications in terms of such encapsulated process structures. These process structures are akin to generic types which can be instantiated upon programmer-specified computations.

Process structures are intrinsically prone to encapsulate customised allocation policies. In other words, it is possible to specify allocation policies embedded at the level of each generic process structure. For example, an allocation annotation for a generic *pipe* structure might express the constraint that each process in a pipe should be on a node which is physically connected (for fast communication) to the nodes on which the previous and next processes of the pipe reside.

The P3L environment already adopt these approach to statically manage and optimise the mapping of a parallel application: a library of predefined mapping scheme is provided or each of the defined process structures and for different architectures. Future research in these areas may approach dynamic allocation to embed in encapsulated process structure dynamic allocation policies able to take into account the dynamic evolution of the system.

In **structured parallel systems** [Bac95] a parallel application is built by composing and nesting local interaction patterns, i.e., group of executing entities characterised by well defined patterns of execution and communication. In those system, it is meaningless to talk about the allocation of the single components, let it be processes or objects, within a group. Instead, each component of the pattern, being strictly correlated in its behaviour with the other component o the pattern must be allocated with regard to the allocation of the whole pattern, both load balancing and locality of communication.

7. Conclusions

In the last few years a great limit of general-purpose automated allocation tools has been outlined. In fact, different applications present different needs with regard to the allocation of their components. A policy that disregards those needs cannot provide to speed-up the application as much as it would potentially possible. Facilities for allowing programmers to customise the behaviour of their application with regard to their dynamic allocation have been proposed in different areas. The paper surveys a number of systems that provide such kind of facilities and explains the functionality offered by the different systems in this regard. In particular, both process-based systems, object-based systems and distributed shared memory systems introduces the capability of customising the allocation of the application components either with language level constructs or as tools in the environment.

The paper provide a discussion of the characteristics presented by all these introduce facilities. Among a variety of heterogeneous approach a very few issues can be outlined. The paper analyses the characteristics of the constructs with regard to their capability of being:

- expressively powerful, i.e., allowing a fine-tuning of customisation;
- modular, i.e., allowing separation of concerns;
- abstract, i.e., portable on different architectures;
- integrated with the an automated policy in the run-time system, so to not force the user of writing its own allocation policy unless it wants to.

Apart from the above ones, the paper tries to open the discussion on several issues disregarded by all the proposal, such as portability, performance study, the introduction of tools for helping programmers in efficiently exploiting the offered capability.

There seems to be great potential for a cross-fertilisation of ideas and researcher working in one of these areas would be well-advised to pay attention to research in the other areas in order to keep abreast of potential solutions to common problems. In particular, promising areas of research are offered by the strict relationship of the analysed problem with the synchronisation area and by the modern style of block structured parallel programming.

8. References

[Ach93] B. Achauer, “The DOWL Distributed Object-Oriented Language”, Communications of the ACM, Vol. 36, No. 9, pp. 48-55, Sept. 1993.

[Aks94] M. Aksit et al., “Real-Time Specification Inheritance Anomalies and Real-Time Filters”, Procededings of ECOOP ’94, LNCS No. 821, pp. 386-407, Springer-Verlag, July 1994.

[ArtF89] Y. Artsy, R. Finkel, “Designing a Process Migration Facility: the Charlotte Experience”, IEEE Computer, Vol. 22, No. 9, pp. 47-56, Sept. 1989.

[Ass93] H. Assenmacher et al., “PANDA - Supporting Distributed Programming in C++”, Proceedings of ECOOP ‘93, LNCS No. 707, pp. 361-383, Springer-Verlag, July 1993.

[Ass94] H. Assenmacher et al., “Meeting the Application in User Space”, Proceedings of the 6th ACM SIGOPS European Workshop, Wadern (D), pp. 82-87, Sept. 1994.

[Bac95] B. Bacci et alii, “A Structured High-Level Parallel Language and its Structured Support”, Concurrency: Practice and Experience, Vol. 5, No. 5, pp. 225-255, May 1995.

[Bal92] H. E. Bal, F. Kaashoek, A. S. Tanenbaum, J. Jansen, “Replication Techniques for Speeding Up Parallel Applications on Distributed Systems”, Concurrency: Practice and Experience, Vol. 4, No. 5, pp. 337-355, Aug. 1992.

[BalK93] H. E. Bal, M. F. Kaashoek, “Object Distribution in Orca using Compile-Time and Run-Time Techniques”, Proceedings of OOPSLA '93, ACM SigPlan Notices, Vol. 28, No. 10, Oct. 1993.

[BarS85] A. Barak and A. Shiloh, “A distributed Load-Balancing Policy for a MultiComputer”, Software - Practice and Experience, Vol. 15, No. 9, pp. 901-912, Sept. 1985.

[Ben90] J. Bennet, "Experience with Distributed Smalltalk", Software: Practice and Experience, Vol. 20, No. 2, pp. 157-180, Feb. 1990.

[Bro95] J. C: Browne et al., "Visual Programming and Debugging for Parallel Computing", IEEE Parallel and Distributed Technology, Vol. 3, No. 1, pp. 75-83, Spring 1995.

[Bru93] D. Bruschi et al., "A User-Friendly Environment for Parallel Programming", Proceedings of the 1st EUROMICRO Workshop on Parallel and Distributed Processing, IEEE CS Press, pp. 451-456, Jan. 1993.

[CarBZ91] J. B. Carter, J, K. Bennett, W. Zwaenepoel, "Implementation and Performance of Munin", Proceedings of the 13th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review, Vol. 25, No. 5, pp. 152-164, May 1991.

[CasK88] T. L. Casavant, J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing System", IEEE Transactions on Software Engineering, Vol. 8, No. 4, pp. 141-154, Feb. 1988.

[Cha89] J. S. Chase et al., "The Amber System: Parallel Programming on a Network of Multiprocessors", Proceedings of the 12th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review, Vol. 23, No. 12, pp. 147-158, Dec. 1989.

[ChiC91] R. S. Chin, S.T. Chanson, "Distributed Object-Based Programming Systems", ACM Computing Surveys, Vol. 23, No. 1, pp. 91-124, March 1991.

[CorL91] A. Corradi, L. Leonardi, "PO Constraints as Tools to Synchronize Active Objects", The Journal of Object-Oriented Programming, Vol. 4, No. 6, pp. 41-53, Oct. 1991.

[CorLZ92] A. Corradi, L. Leonardi, F. Zambonelli, "Load Balancing Strategies for Massively Parallel Architectures", Parallel Processing Letters, Vol. 2, No. 2 & 3, pp. 139-148, Sept. 1992.

[CorLZ93] A. Corradi, L. Leonardi, F. Zambonelli, "How to Apply Locality to Achieve Load Balancing: Migration in Massively Parallel Architectures", Proceedings of the 1st EUROMICRO Workshop on Parallel and Distributed Processing, pp. 42-51, IEEE CS Press, Jan. 1993.

[CorLZ94a] A. Corradi, L. Leonardi, F. Zambonelli, "Dynamic Load Distribution in Massively Parallel Architectures: the Parallel Objects Example", Proceedings of the Conference on Massively Parallel Computing Systems, pp. 318-322, IEEE CS Press, May 1994.

[CorLZ94c] A. Corradi, L. Leonardi, F. Zambonelli, "How to Structure Parallel Applications: Local Nested Aggregates", Proceedings of Joint Modular Languages Conference, Ulm (D), pp. 431-456, Sept. 1994.

[CorLZ96] A. Corradi, L. Leonardi, F. Zambonelli, "High-Level Management of Allocation in a Parallel Objects Environment", DEIS Technical Report, No.DEIS-LIA-96-003, June 1996.

[CPE95a] Concurrency: Practice and Experience, Special Issue on Resource Management in Parallel and Distributed Systems: Static Scheduling, Vol. 7, No. 3, March 1995.

[CPE95b] Concurrency: Practice and Experience, Special Issue on Resource Management in Parallel and Distributed Systems: Dynamic Scheduling, Vol. 7, No. 7, Oct. 1995.

[DouO91] F. Douglis, J. Ousterhout, “Transparent Process Migration: Design Alternatives and the Sprite Implementation”, Software: Practice and Experience, Vol. 21, No. 8, Aug. 1991.

[EagLZ86] D. L. Eager, E. D. Lazowska, J. Zahorjan, “A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing”, ACM Performance Evaluation Review, Vol. 14, No. 1, pp. 1-3, May 1986.

[EagLZ89] D. L. Eager, E. D. Lazowska, J. Zahorjan, “Speedup versus Efficiency in Parallel Systems”, IEEE Transactions on Computers, Vol. 38, No. 3, pp. 408-423, March 1989.

[Gou92] Y. Gourhant et alii, “Dynamic Clustering in an Object-oriented Distributed System”, Proceedings of OLDA-II ,Oct. 1992.

[HeuA89] L. Heuser, B. Achauer, “Language Constructs to Express Distribution of Object-Oriented Applications”, Proceedings of TOOLS ‘89, Paris (F), Nov. 1989.

[Jud95] A. Judge, “Consistency System API”, Working Document, Department of Computer Science, Trinity College, Dublin (IR), Apr. 1995.

[JudC93] A. Judge, V. Cahill, “Sharing Objects in a Distributed System”, Proceedings of the 3rd International Workshop on Object-Orientation in Operating Systems, Asheville (NC), pp. 136-140, Dec. 1993.

[Jul88] E. Jul et al., “Fine Grained Mobility in the Emerald System”, ACM Transactions on Computer Systems, Vol. 6, No. 1, pp. 109-133, Feb. 1988.

[KalK93] L. V. Kalé, S. Krishnan, “Charm++: A Portable Concurrent Object-oriented System Based on C++”, Proceedings of OOPSLA '93, ACM SIGPLAN Notices, Vol. 18, No. 10, Oct. 1993.

[KalS94] L. V. Kalé, A. B. Sinha, “Information Sharing Mechanisms in Parallel Programs”, Proceedings of the International Parallel Processing Symposium, Cancun (M), April 1994.

[Lux95] W. Lux, “Adaptable Object Migration: Concept and Implementation”, ACM Operating Systems Review, Vol. 29, No. 5, pp. 54-69, May 1995.

[Mar90] D. C. Marinescu at al., “Models for Monitoring and Debugging Tools for Parallel and Distributed Software”, The Journal of Parallel and Distributed Computing, Vol. 9, No. 2, pp. 171-184, June 1990.

[Mas94] H. Masuhara, “Study on a Reflective Architecture to Provide Efficient Dynamic Resource Management for Highly Parallel Object-Oriented Applications”, Master Thesis, University of Tokyo, Tokyo (J), Feb. 1994.

[McH94] C. Mc Hale, “Synchronisation in Concurrent Object-Oriented Lanaguages: Expressive Power, Genericity and Inheritance”, PhD Thesis, Department of Computer Science, Trinity College, Dublin (IR), Oct. 1994.

[NitL91] B. Nitzberg, V. Lo, “Distributed Shared Memory: a Survey of Issues and Algorithms”, IEEE Computer, Vol. 24, No. 8, pp. 52-60, Aug. 1991.

[NorT93] M. G. Norman, P. Thanisch, “Models of Machines and Computation for Mapping in Multicomputers”, ACM Computing Surveys, Vol. 25, No. 3, pp. 263-302, Sept. 1993.

[Nut94] D. Nuttel, “A Brief Survey of Systems Providing Process or Object Migration Facilities”, ACM Operating Systems Review, Vol. 28, No. 4, pp. 64-79, Oct. 1994.

[OkaI94] H. Okamura, Y. Ishikawa, “Object Location Control Using Meta-level Programming”, Proceedings of ECOOP ’94, LNCS No. 821, pp. 299-319, Springer-Verlag, July 1994.

[PowM83] M. L. Powell, B. P. Miller, “Process Migration in DEMOS/MP”, Proceedings of the 9th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review, Vol. 17, No. 5, pp. 110-118, May 1983.

[ProTM96] J. Protic, M. Tomasevic, V. Milutinovic, “Distributed Shared Memory: Concepts and Systems”, IEEE Parallel and Distributed Technology, Vol. 4, No. 2, pp. 63-79, Summer 1996.

[ShiKS92] N. G. Shivaratri, P. Krueger, M. Singhal, “Load Distributing for Locally Distributed System”, IEEE Computer, Vol. 25, No. 12, pp. 33-44, Dec. 1992.

[ShiWP90] B. Shirazi, M. Wang, G. Pathak, “Analysis and Evaluation of Heuristic Method for Static Task Scheduling”, The Journal of Parallel and Distributed Computing, Vol. 10, No. 3, pp. 222-232, March 1990.

[SinK93] A. B. Sinha, L. V. Kalé, "A Load Balancing Strategy for Prioritized Execution of Tasks", International Symposium on Parallel Processing, Newport Beach, April 1993.

[Smi88] J.M.Smith, “A Survey of Process Migration Mechanisms”, ACM Operating Systems Review, Vol. 22, No. 3, pp. 28-40, July 1988.

[Sto96] A. D. Stoyenco et al., “Load Balanced Mapping of Distributed Objects to Minimize Network Communication”, The Journal of Parallel and Distributed Computing, Vol. 34, No. 2, pp. 117-136, May 1996.

[TanC92] B. Tangney and A. Condon, “Some Issues in Load Balancing in Amadeus”, ECOOP'92 Workshop on Load Balancing in Object-oriented Systems, July 1992.

[TanO91] B. Tangney, A O'Toole, "An Overview of Load Balancing in Amadeus", Proceedings of the 4th ISMM/IASTED Conference on Parallel and Distributed Computing and Systems, pp. 144-146, Oct. 1991.

[Tar94] E. Tarnvik, “Dynamo: a Portable Tool for Dynamic Load Balancing on Distributed Memory Multicomputers”, Concurrency: Practice and Experience”, Vol. 6, No. 8, pp. 613-639, Dec. 1994.

[The85] M. M. Theimer, K. A. Lantz, D. R. Cheriton, “Preemptable Remote Execution Facilities for the. V-System”, Proceedings of the 10th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review, Vol. 19, No. 5, pp. 2-12, May 1985.

[UehT91] M. Uehara, M. Tokoro, “An Adaptive Load Balancing Method in the Computational Field Model”, ACM OOPS Messenger, Vol. 2, No. 2, pp. 109-113, April 1991

[WanM86] Y. T. Wang, R. J. T. Morris, “Load Sharing in Distributed Systems”, IEEE Transactions on Computers, Vol. 34, No. 3, pp. 204-217, March 1986.

[WatY88] T. Watanabe, A. Yonezawa, “Reflection in an Object-Oriented Concurrent Language”, Proceedings of OOPSLA ‘88, ACM SigPlan Notices, Vol. 23, No. 11, pp. 306-315, Nov. 1988.

[Weg90] P. Wegner, “Concepts and Paradigms of Object Oriented Programming”, ACM OOPS Messenger, Vol. 1, No. 1, pp. 7-87, Aug. 1990.

[Zhou93] S. Zhou et al., “Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems”, Software Practice and Experience, Vol. 32, No. 12, pp. 1305-1336, Dec. 1993.