Experiences toward an Object-oriented Approach to Structured Parallel Programming

Antonio Corradi, Letizia Leonardi, Franco Zambonelli

DEIS Technical Report No. DEIS-LIA-95-007, December 1995

Experiences toward an Object-oriented Approach to Structured Parallel Programming

Antonio Corradi, Franco Zambonelli
Dipartimento di Elettronica, Informatica e Sistemistica - Università di Bologna
Viale Risorgimento 2 - 40136 Bologna - ITALY
Ph.: +39-51-6443001 - Fax: +39-51-6443073
E-mail: {acorradi, fzambonelli}@deis.unibo.it

Letizia Leonardi
Dipartimento di Scienze dell'Ingegneria - Università di Modena
Via Campi 213/b - 41100 Modena - ITALY
Ph. +39-59-374538 - Fax: +39-59-370040
E-mail: lleonardi@deis.unibo.it

DEIS Technical Report No. DEIS-LIA-95-007, December 1995

Abstract

In parallel programming, communication patterns are rarely arbitrary and unstructured. Instead, parallel applications tend to employ predetermined patterns of communication between their components. If the most commonly used patterns - such as pipelines, farms and trees - are identified (both in terms of their components and their communication), an environment can make them available as high-level abstractions for use in writing applications. This can yield a structured approach to parallel programming. The paper shows how this structured approach can be accommodated within an object-oriented language. A class library provides the most commonly used patterns and programmers can exploit inheritance to define new patterns. Several examples illustrate the approach and show that it can be efficiently implemented.

Keywords

Structured Parallelism, Object-Orientation, Communication Patterns

This work has been partially supported by the 'Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo' of the Italian National Research Council (grant No. 89.00012.69).

Experiences toward an Object-oriented Approach to Structured Parallel Programming

Antonio Corradi, Letizia Leonardi, Franco Zambonelli

1. Introduction

All concurrent/parallel programming environments suffer the problem of user acceptance: a full expression of the behaviour of parallel programs, in fact, requires an enormous effort compared with the effort needed to develop sequential programs. This happens also in the area of object-oriented systems: environments based on concurrent/parallel objects keep on appearing [CAMC93] but they are still used only by a very limited community of expert programmers. The only way of enlarging the parallel user community is to limit the degree of knowledge the user must have to express of concurrency in his/her applications.

One possible approach is to make the user specify his/her programs in a sequential fashion and let the system produce automatically their parallel counterpart: program transformation has been attempted in different areas by following different approaches [TPDS91]. However, general problems are intrinsically hard to be automatically parallelised without user involvement: only a special-purpose approach in very limited areas, such as loop parallelisation and unfolding for systolic architectures [Jez93], can be effective. Parallel operators can be accommodated within traditional sequential languages with the aim of helping compilers to decide how to exploit data parallelism [ZimC93], but the problem of automatically exploiting parallelism at the control level is still open.

A promising approach to explicitly deal with parallelism is **structured parallel programming**. In general, parallel applications follow predetermined patterns of execution for the parallel activities and their interactions: communication patterns are rarely both arbitrary and logically unstructured [Han93]. A structured approach to parallel programming introduces abstract aggregations to represent well-defined patterns of interactions: we call them **high-level parallel compositions** (**HLPCs** for short). If most commonly used patterns - such as pipelines, farms, trees - were identified (both in terms of components and their communication), an environment could make them available as high-level abstractions to applications. Applications can be built by making their components interact only by means of the above patterns and by allowing the user to disregard any issue related to the internal coordination of the components of its applications. A few proposals adopt this perspective: one can either embed these constructs in a language, as in P3L [Bac95], or introduce them as skeletons for parallel algorithms [Dar95].

In the paper, we define the structured approach by accommodating HLPCs within an **object-oriented** framework. Object orientation makes an HLPC described as a composition of objects, instances of a few predefined base classes, that are very general and reusable. The starting environment offers the user a set of components that implements the most common parallel patterns of communication: a user can compose and nest those patterns together for developing his/her applications. In addition, the

user is not restricted to the limited set of predefined communication patterns: if new ones are required, they can be obtained by sub-classing from existing patterns.
The **Parallel Objects** environment [CorL91, CorLZ94] is adopted as a case study to detail the structure of a possible implementation of HLPC and to evaluate their performance.
The paper is organised as follows. Section 2 presents the structured approach to parallel programming in its general terms, while section 3 shows how to model it an object-oriented way. Section 4 shows the Parallel Objects environment, section 5 specialises the structured approach within it. Finally, the implementation of some HLPCs in PO is shown in section 6 and their performances evaluated in section 7.

2. Structured parallel programming

Parallel programming is a complex activity to deal with. Traditional approaches model parallel applications as graphs, where each process is a node and any communication can be seen as one arc. However, even if several proposals introduce tools to describe parallel applications in terms of task graphs [Bru93], the abstraction level of these approaches is too low-level and obliges users to directly face process communications.
Models such as client-server [Adl95] and tuple space [Ah88] define higher-level abstractions for managing communications, but still do not face the problem of abstracting the interactions among several items: this is still left to the users. In addition, these models are unable both to introduce any modularity principle and to enforce reusability of the parallel behaviour of applications.
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 [Han93]. In such a scenario, a structured parallel programming approach introduces the abstraction of interaction pattern and describes parallel applications by means **high-level parallel compositions** (**HLPCs**) capable of implementing the above patterns. The obtained compositions can then be made self-contained modules and they can be used disregarding its internal parallel implementation. Moreover, the encapsulation of an HLPC not only follows the modularity principle but also achieves an effective reuse of parallel behaviours, by making it possible to describe **generic parallel pattern** of interaction independently of the functionality provided by its components.
The structured approach to parallel programming, in the last few years, has followed two main directions:

- the enrichment of traditional parallel environments with **libraries of "skeletons"**, each one representing a given communication pattern [Col89, Dar95];
- the definition of **restricted parallel languages** that express communication only in terms of a predefined patterns [Bac95].

The above approach can significantly help parallel programming development. Instead of programming a concurrent application from scratch and of controlling both process creations and their communication, the user can simply identify the HLPCs that implement the suitable patterns for his/her needs and bind them with the sequential code.
A major problem of both approaches is to find the "most used and usable" parallel communication patterns and to make them available either in a library or as language constructs. From the analysis of different proposals, in different areas and with different intentions [Han93, Skil93], it is possible to select a few parallel patterns of

inter-connection that are significant and can guarantee large reuse, in particular:

- **pipe**, composed of a set of stages interconnected one after another. This allows information to flow from one stage to another;
- **farm**, composed of a set of workers and one controller. The workers execute in parallel to reach a common goal; the controller is in charge of distributing jobs and of controlling their progress;
- **divide & conquer**, shaped after a tree: the information flows either from the root to the leaves or vice versa, allowing nodes at the same level of the tree to execute in parallel;
- **geometric**, formed by a set of items, each one interconnected with other items (its neighbours) to form a global topology (for example, mesh, torus, etc.). The parallel execution iterates several times and communication takes place at any step within the neighbourhood.

However, even if those parallel patterns are widely used, there is no formal agreement on their semantics. For instance, the farm is a concept generally well understood, but its realisation obliges users to choose among different strategies of implementation. When workers can execute in parallel, several policies can be followed to deal with answers:

- all answers are significant and awaited by the controller, until all workers have produced an answer;
- only a predefined number of answers is awaited by the controller while all other workers can proceed to completion;
- as soon as one answer is produced by one worker and given back to the controller, the controller aborts the jobs of other workers.

Moreover, a farm controller defines the operation semantics on data: it can force all workers to operate on the same data, each producing independently its results. In this case, workers should have different behaviour to produce results in different ways, towards a final comparison.

The lack of consensus on the semantic of parallel patterns and the difficulty of the idea of completeness makes it dangerous to limit the expressive power of a parallel environment either in few libraries or in a language. Even if, in the skeletons approach, new libraries can be added to feed new pattern needs, their definition is usually complex and must occur at very low-level, forcing the users to the awareness of architectural details.

Extendible environment have been proposed in the last few that makes it possible for the programmer to define, either via a high-level language or a graphical tool, any needed parallel pattern, and make it available in the environment at the same level of previously defined ones. The basic feature of all these systems - among the others Fortran-M [Fos95], Tracs [Bar95], PCS [Hin95] - is the definition of **context independent modules**, with a well defined interface, that can be connected to each other via **high-level communication channels**. The obtained parallel compositions represent statically determined communication patterns and can be made, in their turn, context independent and reusable modules (thus providing **encapsulation of parallel behaviours** and **nesting capability** of modules). Those systems still require programmers that wants to defines new modules to explicitly deal with parallelism and its related complexities. Nevertheless, they have the great advantage of providing these capability at a high abstraction level. A drawback is that those in those system, the definition of a parallel pattern has to be made by explicitly referencing and connecting the modules it composed of, without clearly separate the definition of the parallel pattern from its functionality, thuis diminishing the capability of reusing parallel behaviour.

3. The Object-Oriented approach

Object-orientation can be used to encapsulate and abstract common patterns of parallel interactions toward a structured style of parallel programming. The basic idea in the presented object-oriented approach is of defining HLPCs as objects. More in particular, a HLPC can be defined as an object in charge of controlling and coordinating the execution of its internal component, objects in their turn, following a given pattern*.

The proposed approach define an extendible HLPC environment that tends to overcome the limits of currently available. More in particular it provides the following feature:

- **uniformity**: all entities within the environment - including HLPCs - are objects and the communication channel - executing processes dicothomy is no more present. Parallel patterns of communication are described at the class level, as any other component of the applications and any new parallel patterns can be defined, at the same level of already defined ones, as a new class;
- **generality**: the capability of dynamically generating references in an object-oriented environment makes it possible generic HLPC class to be defined, by defining the components of an HLPC as generic references to objects. Thus, the specification of the parallel part can be clearly separated from the any functional specification; For instance, a generic farm can be defined without specifying what (i.e., objects of which class) its worker are.
- **ease of reuse**: the inheritance mechanism simplify the definition of specialised parallel patterns sub-classing from existing ones. It is clear that to define a new pattern the user has to explicitly deal with parallelism. However, inheritance applied to the concurrent object behaviour [CorL91] helps in the specification of the parallel behaviour of a HLPC. For instance, if the environment initially provides a farm pattern whose controller waits only for the first answer, it is very simple for the user - as shown in the following - to specialise this behaviour by defining a new farm that deals with all answers.

3.1. The HLPC base classes

The basic idea of the proposed object-oriented approach is to implement any kind of parallel patterns as classes, from which object can be instantiated and the execution issued from the external by means of a service request.

A trivial solution can implement a given parallel pattern by means of a given **manager class**: an instance of this class represents a HLPC and makes the HLPC an encapsulated abstraction by hiding its internal structure. The manager holds the reference to a set of object, that represent the components of the HLPC and whose execution must be coordinated by the manager itself (see figure 1).

When one request arrives to the manager, an execution flow start within the HLPC. More in detail, the manager has the duty of requiring services to its components objects, collecting the answers and redistributing them on the basis of the parallel

* The paper rules out any formal coordination model for concurrent/parallel object-oriented programming [Nie93, Tsi91,ForA93].

pattern it has to implement.
A big advantage that comes from the this approach is that the objects do not actively partecipate to the implementation of the composition, but they can be considerd as external entities that furnishes the algorithic part. This characteristic of the components objects, that makes us call them **slave objects**, grant a high-degree of genericity

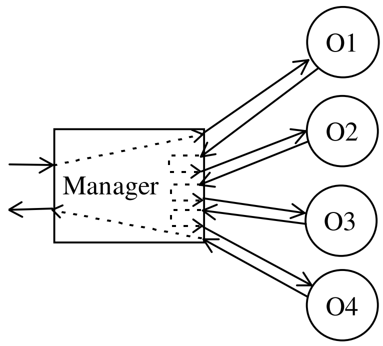

Figure 1. A trivial implementation of the pipe pattern

A drawback is that this solution is a very inefficient one: the parallel pattern is only simulated by the manager that is even likely to become the bottleneck of the composition.
To provide a real implementation of any needed communication pattern, a more composite implementation is needed: the client-server interface of the slave objects must be made homogeneous to the parallel pattern they should be a component of. For example, to be the component of a pipe an object has not to send the reply of a service request to the client that required it but, instead, to the object that follows it on along the pipe.

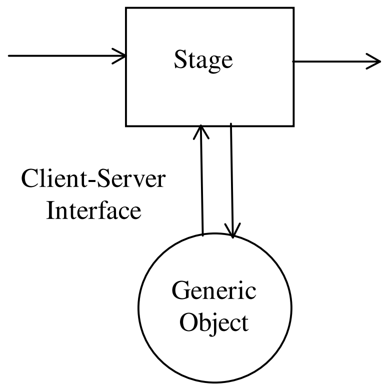

Figure 2. Adapting the client-server interface of an object

The above adaptation must be made without changing the implementation of the slave objects themselves: that would be a violation of the encapsulation principle and would no more guarantee the genericity of the approach. Instead, special purpose objects, generically called **stages**, are introduced.
Stages objects are in charge of encapsulating the client-server interface of the slave objects and of providing the connection semantic needed by the pattern. More in detail, each stage acts as a node of the graph that represents the pattern: the stage can be directly connected to the manager and/or to other component stages, depending on the particular pattern the HLPC implements (figure 2). The communication can flow directly from stage to stage, without any centralised manager control; this prevents the manager becoming a bottleneck and makes it possible to exploit locality in

communication.

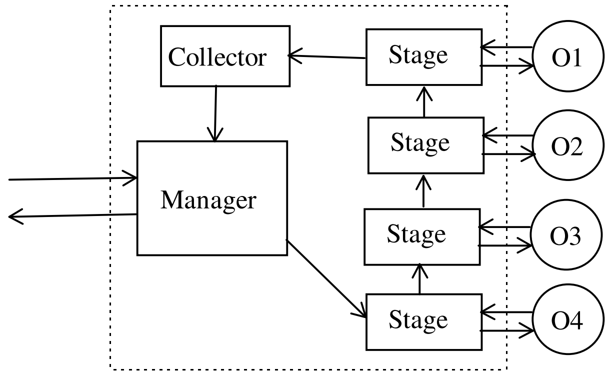

Figure 3 - Relationships among the objects that constitute the pipe pattern
(the dotted box represent the incapsulated HLPC, boxes are composing objects, circles are slave objects)

A further improvement comes from the introduction of another class of object, the collector class. During the service of a request, control flows within the stages of HLPC depending on the communication pattern the HLPC implements. Once the composition finishes its execution, the result does not return directly to the manager, but to an instance of the collector class in charge of collecting it and yielding it to the manager. The manager dynamically associate one collector to every service request with the possibility of distinguishing, within a HLPC, different computational flows: in this way, several requests can be concurrently served within the same HLPC and no interference of results occurs.
Figure 3 presents the schemes of a pipe HLPC and shows the role of the different components.

3.2. Portability of the approach

The approach presented in the next section is very general and assumes the avialability, into the destination onto the target object-oriented environment, of features that are widely available in current proposals of object-oriented concurrent programming environment:

- the capability of requiring services to objects in both **asynchronous and future asynchronous communication modes**;
- the capability of objects of intra-object parallelism, i.e., of servicing several requests concurrently;
- the availability of **synchronisation mechanims** for the servicing of parallel request;
- the availability of **flexible data typing** mechanism i.e., the capability of dynamically associating types to object and to method parameter;

The first feature is necessary for allowing the manager and the stages of an HLPC to implement any needed communication pattern without centralised control. A simple example can be found in the implementation of the farm pattern: in this case, to grant parallel execution of the farm's worker, the manager should be able to asynchronously command their execution. Another example may refer to pipe case: the stage must distribute the result obtained from its slave object the next stage of the pipe without being forced to wait for an unuseful result.
The second feature, the capaility of objects to serve several request concurrently, is necessary to make our HLPC capable of managing several execution flows at a time.

Again with reference to the pipe pattern, is the manager is not capable of serving several external request at a time - each one corresponding to a different execution flow within the farm , the potential parallel of the pipe itself wiould be completely unexploited.
The needs of synchronisation mechanism is an obvious requirement for any systems supporting intra-object parallelism: it is necessary to grant consistency on the object state. Moreover, with regard to the HLPC, synchronisation can be necessary for implementing parallel pattern. As an example, the stage of a geometric pattern must synchronise its execution with the one of the neighbour stage: the next execution stage can start only when all its neighour have completed the last one.
Finally, to grant genericity od HLPCs, the system mush furnish the capability of managing generic datatypes. If an HLPC is designed not to implement in itself the algorithmic part but in order to define only the parallel pattern of interaction, it must be capable of adapting its interface to manage very different type of input and of output data, depending on the particular algorithm the slave objects implement. To do that, two solution arises:

- giving the capability to these objects of managing generic data types, i.e., data whose type is dynamically specified as a parameter at their creation;
- giving the capability to these objects of managing untyped data, i.e., data to be treated as a stream without any typechecking.

In the following, we use the Parallel Objects programming model as a case study to detail the structure of the described components and to evaluate the performances of the approach. Several examples shows the use of use of the HLPCs in the stated approach.

4. The Parallel Objects Model

The Parallel Objects (from now on, **PO**) model defines active objects - i.e., objects that own in themselves execution capabilitie [ChiC91]. PO applications can exploit both inter-object and intra-object parallelism. The former stems from the presence of several objects that can execute in parallel, the latter from the possibility of serving several requests in parallel within the same PO object.
A PO object has a structure similar to the one of a Smalltalk object: it is composed of state and behaviour. In addition, any objects integrate an *a priori* synchronisation policy to specify the synchronisation policy for one or more operation of a class: the service of one request can be delayed and it is started when no interference can arise with already executing ones. PO synchronisation policies are expressed by means of high-level directives, called **constraints**. For example, the following constraints specify a readers/writers policy:

```
MaxPar(read); Mutex(write); MutEx(read, write);
```

For further details about PO synchronisation constraints see [CorL91].

4.1. PO Classes

All PO objects derive from the definition of classes, following the usual property of abstraction[22]. Objects of the same class share the same specification contained in the class itself: any PO object is always created by one class and it is tied to it for its whole life.
PO classes relate to each other via inheritance. Inheritance permits to derive a new

specification from existing ones. PO supports multiple inheritance and the inheritance lattice has a unique root in the class Object. The class Object describes the general structure of any object and the default synchronisation constraint, maximum parallelism (constraints MaxPar) for all its methods for active sub-classes: we claim inheritance apply to synchronisation constraints [CorL91].
From a syntactic point of view, the definition of one PO class follows the below template:

```
CLASS: <name of the class>
   INHERITS_FROM: <list of names of direct parent classes, Object by
   default>
   INSTANCE_STATE: <list of names and types of instance variables>
   INSTANCE_METHODS: <list of public methods>
   PRIVATE_METHODS: <list of private methods>
   SCHEDULING_PART: <list of synchronisation constraints>
END_CLASS <name of the class>;
```

The INSTANCE_STATE section contains the definition of the state variables of the class. In PO, all variables are typed and their definition has the form:

```
<type> <variable-name>;
```

The used <type> can be either a C-type or a class name. In addition, the "anytype" type is introduced to avoid any type checking on a variable and treat it as a simple stream (these characteristics of PO allow to build generic HLPCs).

```
int x; /* x is a variable of type integer */
ClassFoo foo; /*  foo is a reference to an object of the class ClassFoo */
anytype data; /* data is not typechecked */
```

All variables - including arrays - have a by-value semantic. Only in case <type> is a class name, the defined variable has always a by-reference semantic, differently from the C++ case[23]. However, as in C++, the type control is based on the inheritance relationship among classes. For example, a reference to an instance of the class Object can be used to refer to objects of any class.
PO objects must be explicitly created (and initialised) by using the following syntax:

```
obj  =  <class-name> Create ([initialisation_parameter [,....]])
```

Let us note that PO objects can also be explicitly deleted, even if this action is discouraged and usually left to a garbage collector tool.
The INSTANCE_METHOD section contains the definition of the public methods of the class. The syntax of a method definition is as follows:

```
METHOD:  Foo_Public (Type parameter [,...]) RETURNS: ReturnType;
        /* local variables */
        BEGIN
                /* body of the method */
        END_METHOD;
```

Whenever a method named init is defined in the INSTANCE_METHOD section, it will be considered as the initialisation method and automatically invoked at object creation: the parameter it accept explicitly passwd in the creation commands. Let us note that the init method is always executed before any other service request and in mutual exclusion with them.
The PRIVATE_METHODS section permits the definition of private methods that can

be invoked only by internal activities.
Apart from method definition and invocation (see next section), the syntax for the specification of the body of PO methods follows the syntax of the C language [KerR88]. In addition, C functions can be defined and called by PO methods.
The SCHEDULING_PART section of the definition of a class contains the synchronisation constraints we have already described.

4.2. PO Communications Modes

To require services to objects, PO defines three communication modes. The **synchronous** mode engages the client activity until the server active object yields the reply. The **asynchronous** mode does not force the client activity to wait: the client simply sends the request to the server active object and then it continues its execution. The **future** communication mode makes the client activity to wait for the reply only when it needs it. Asynchronous and future communication modes grant inter-object parallelism by allowing client and server objects to execute at the same time.
To send a message to an object by using a synchronous, a future and an asynchronous communication mode, respectively, the programmer uses this notation:

```
<object-reference> <method-name> (<parameter-list>);
@ <future-variable> <object-reference> <method-name> (<parameter-list>);
@ <object-reference> <method-name> (<parameter-list>);
```

The <future-variable> belongs to the specific primitive type FutureType. Any time a FutureType variable is used in a communication (with the @ symbol), it indicates a future communication. When, a FutureType variable is used in an expression (or as a parameter in a method invocation), it refers to the reply of the communication it represents; in case the reply has not been delivered yet, the use of a FutureType variable implies the suspension of the activity that uses it. When a client wants to invoke operations upon several active objects, an array of future variables can be used. The client can wait either for a specific reply (by using the normal indexed notation to access to one element of the future array), or for all replies (by using the name of the future array).
Let us note that the <object-reference> can also be the pseudo-variable Self (as in Smalltalk). In addition, the <method-name> can contain the pair <class-name>.<method-name>. This dot notation can be used when the operation is inherited from a super-class to refer to the overridden method.

5. High-level Parallel Compositions in PO

This section describes the PO classes that implement HLPCs. In particular, it is firstly described the interface provided by the most common HLPCs avaialable in the environment. Then, several examples are provided to show how to use the available HLPCs. Finally, a insight to the PO code for some implemented HLPC shows the role played by inheritance, inter-/intra-object parallelism and synchronisation constraints and the way a user can define new HLPCs by inheriting from already defined ones.

5.1. The interface of the HLPCs

As we have already described in section 3, an HLPC comes from the composition of a set of objects of three kinds. In particular, each HLPC is composed of a manager object, that represent the HLPC itself, several stage objects and one collector object for each request issued within the HLPC. In addition, for each stage of the HLPC, one

slave object is charged of implementing the needed functionalities.
In PO the basic templates for the above items any HLPC is composed of are: the ComponentManager and the ComponentStage abstract* classes and the concrete ComponentCollector class. One instance of a concrete class derived from the ComponentManager class - called the manager for short - represents an HLPC within a PO application: we call it the manager for short. Instances of concrete classes derived from the ComponentStage class - called stages - connect each other in order to implement the composition. Each stage commands the execution of a PO object, called slave, controlled by the stage itself.
The creation of the stages and of the collectors and their subsequent interactions are transparently managed by the manager. Thus, from the point of view of a user interested in reusing the parallel behaviour already defined in some HLPC classes, the only class of intereset is the manager one.
When a user is interested in using a HLPCs within an application, it has to create one instance of the interested manager class, i.e., the one that implements the needed parallel behaviour, and initialise it with the reference to the slave objects to be controlled by each stage and the name of the method to be requested to it.
More in detail, the operation *init* - that we recall is automatically issued at object creation time - accepts a null-terminated array of pairs (defined in the Association type) formed by the reference to a slave object and the name of the operation to be invoked for it:

```
typedef struct { Object obj; char meth[20] } Association;
METHOD: init (Association list[]) RETURNS: void
```

The *init* operation should create its internal stages and pass to each one, as initialisation parameters, the knowledge of its slave objects and the name of the method that the stage should invoke upon its slave object. This approach facilitates the definition of a "generic" stage that can drive any kind of slave object, without any addition or change in the definition of the stage classes. Anyway, as it is shown in the following, peculiar concrete manager classes could accept the initialisation list in a different form, i.e., a 2-D array.
After the initialisation, the user can request the manager start a computation an execution within the HLPC. To do that, it must request the manager the *execution* service request, either in synchronous or future modes:

```
METHOD: execution (anytype datain) RETURNS: anytype;
```

The *execution* method creates the collector object devoted to this request, commands the stages by passing them the input data - without any typechecking - and the reference to the collector, then requests the result from the collector object. When the result is obtained from the collector it is returned externally, again, without any typechecking.
The same manager can be used for more computations in parallel. In other words, once a manager object has been created and initialised, several requests of execution can be served in parallel.

5.2. Using the PO HLPCs

Some application examples illustrate the ease to use of the aobve described HLPCs.

---

* A class is abstract if no instance can be created from it: abstract classes are templates to be further specialized by inheritance [Str91, GolR83].

As a first, simple example, let us suppose we need to **sort an array** of information. It is well known that, depending on data, different sorting algorithms can show different performances. Thus, it could be useful to sort the same data by running different algorithms in parallel and to wait for the quickest answer. This parallel pattern is clearly identified in the previously described FarmManager class.
In an object-oriented environment, different sorting algorithms can be provided as public operation of objects of different classes. For example, suppose the following objects of different sorting classes have been already created:

```
QuickSort QS,ShakeSort SS,........,MergeSort MS
```

To build a farm composed of the above objects, the user has to insert their references in a list of pairs both with the name of their sorting operation:

```
Association *LIST = {QS,"qsort",SS,"ssort";.....,MS,"msort",'\0',0};
```

Then, the user creates one instance on the FarmManager class and initialises it by passing the above list to the init method:

```
FarmManager FASTSORT = Create(LIST);
```

At this point, the FarmManager instance is ready to work, i.e., the user can request it the execution operation:

```
int DataIn[100], DataSorted[100];
... /* DataIn array initialisation */ ...
DataSorted =  FASTSORT execution(DataIn);
```

The FarmManager is also capable - without any change - of sorting several arrays in parallel. For example, by using future communication modes:

```
int DataIn1[100], DataIn2[200];
Future_int DataSorted1[100], DataSorted2[200];
/* DataIn1, DataIn2 arrays initialisation */
@ DataSorted1 FASTSORT execution (DataIn1);
@ DataSorted2 FASTSORT execution (DataIn2);
.........
a = DataSorted2[4]; /* possibly blocking references */
b = DataSorted1[40];
```

A more complex approach, in the area of **molecular dynamics**, shows how several HLPCs can be easily composed/netsted to form higher level parallel abstraction.
Molecular dynamics simulations aims to determine the dynamic behaviour of complex molecules by computing the effects of the forces that act onto the components particles at a microscopic level [Byw94]. Data parallelism can be easily exploited by splitting the space of simulation and assigning the calculus to a grid of parallel entities, that periodically exchange in a iterative way the result of the calculus wiht their neighbourhood.
This kind of data parallism is easily find a lot of parallel application, and the PO environment furnishes the HLPC class that implement it. More in particular, a 2-D grid HLPC is available that accept a 2-D array of the Association type as initialisation parameter:

```
Association OLIST[X_DIM][Y_DIM] =
                {{O[1][1], "calculate_forces", O[1][2], "calculate_forces",...,'\0',0
 },
```

```
{O[2][1], "calculate_forces", O[2][2], "calculate_forces",...,'\0',0},
.....};

MeshManager MOL_DYN_SIM = Create(OLIST);
```

At this point, the MeshManager object is initialised and it hold the references to the slave objects (O[i][j]) and the name of the method to be invoked on them (calculate_forces). Then, execution can be started within the MeshManager by invoking the execution method and passing to the execution method a 2-D array of initial data and an addition number that specify the number of iteractions that have to be issued before any results return:

```
int Number_of_Iteractions = MAX_ITER;
data_type_of_input Input_Data[X_DIM][Y_DM] = ...../* input data initialisation */
data_type_of_output Output_Data[X_DIM][Y_DIM];

Output_Data = MOL_DYN_SIM execution (Input_Data, Number_of_Iteractions);
```

A further degree of parallelisation a molecular dynamic application can benefits from derives from the fact that different forces with different scope - i.e., short ranged, electrostatic, contraints and thermostatic forces - act at the same time on a given portion of the space. Thus, the calculus of the effect of different forces on a given portion of the space can proceed in parallel: this is a typical farm parallelism in which all the answer computer by the workers are needed. The array of O objects devoted to the calculation of the forces in a given region of the simulation space can then be substituted by an array of FarmMultipleResultManager objects, whose each worker is devoted to the calculus of single force.

Let us suppose the following object of different classes have been already created:

```
ShortForces SF[X_DIM][Y_DIM], ElectrostaticForces[X_DIM][Y_DIM], ....
```

An array of FarmMultipleResultManager objects can be created and initialised with the appropriate workers:

```
FarmMultipleResultManager FMRM[X_DIM][Y_DIM];
Association FLIST[X_DIM][Y_DIM][NUMBER_OF_FORCES];

for(i=0;i< X_DIM;i++)
  for(j=0;j< Y_DIM;j++)
        {FLIST[i][j] = {SF, "calc_short_forces", EF, "calc_elet_forces",...};
         FMRM[i][j] FarmMultipleResultManager = Create (FLIST[i][j]);}
```

At this point, the MeshManager can be used the same way we have shown above, with the only care of initialising it with the reference the FMRM objects, that assumes the role of its slave objects, and by specifying that the name of the method to be invoked on it is execution. In any case, the fact that the resulting application is a nesting of different HLPC is transparent to the programmer and do not increases the complexity of the application.

6. The HLPCs' Implementation

This subsection goes down to the detailed structure of the base abstract classes for the implementation of the HLPCs and shows how inheritance can simplify the definition of

new HLPCs from already defined ones. In particular, it shows the implementation of one pipe composition and some kinds of farms.
Figure 4 reports a part of the inheritance graph that interconnects manager and stage classes of some of the implemented compositions available in the starting PO environment.

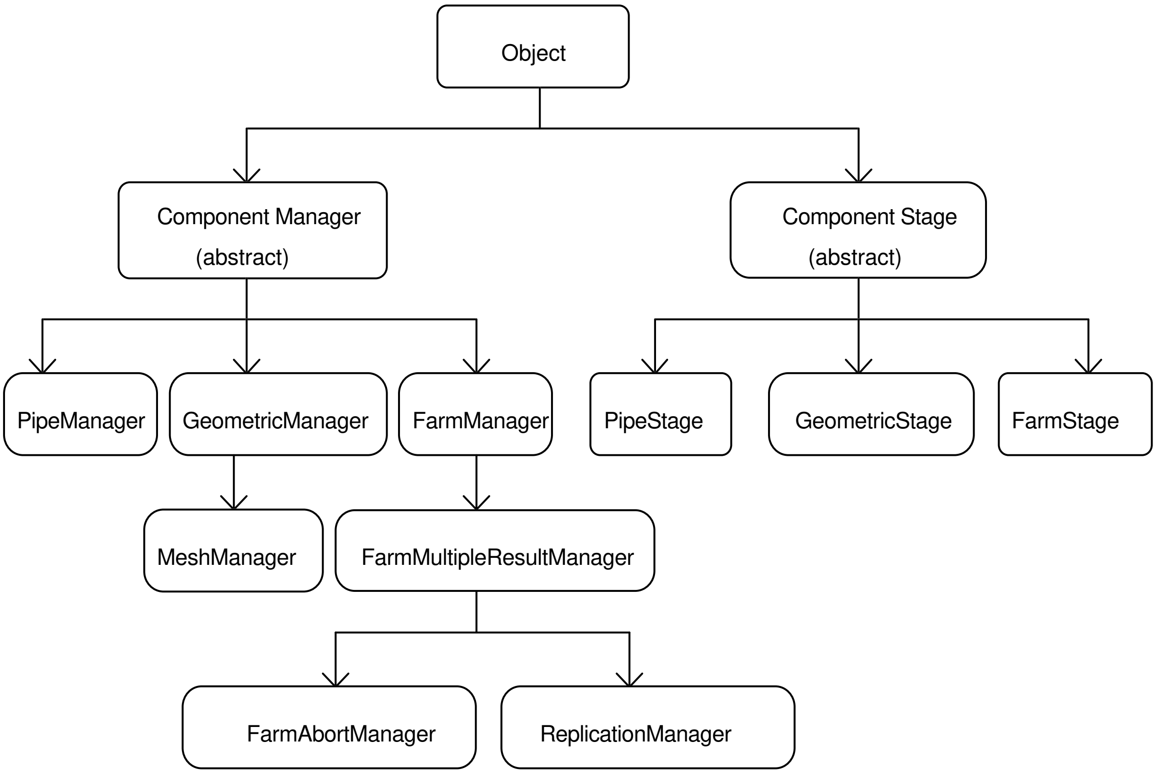

Figure 4. Inheritance graph portion for manager and stage classes.

6.1. The Base Abstract Classes

The abstract class ComponentManager defines the general structure of any manager from which all specialised managers derive (e.g., the PipeManager and the FarmManager classes discussed later).
Each instance of a class belonging to the manager hierarchy maintains the references to the stage components in its state variable called stages:

```
INSTANCE_STATE: ComponentStage stages[];
```

The public operations of the class Manager are, as already explained, *init* and *execution*.
The operation *init* accepts a null-terminated array of pairs (defined in the Association type) formed by the reference to a slave object and the name of the operation to be invoked for it:

```
typedef struct { Object obj; char meth[20] } Association;

METHOD: init (Association list[]) RETURNS: void;
BEGIN
END_METHOD;
```

In addition, a private operation *commandStages* is defined.
The operation The *init* operation is left empty in this abstract class and has to be redefined in each subclass to initialise the specific parallel communication pattern realised by the stages and to associate them with the approapriate slave objects (whose references and method's names are contatined in list):

```
METHOD: init (Association list[]) RETURNS: void;
BEGIN
END_METHOD;
```

The operation *execution* is used to start a computation in one HLPC and returns the correspondign result of the HLPC computation:

```
METHOD: execution (anytype datain) RETURNS: anytype;
    ComponentCollector res;
BEGIN
    res = ComponentCollector Create();
    Self commandStages(datain, res);
    RETURN  res get();
END_METHOD;
```

Let us note that the abstract class ComponentManager does not specify how the job is commanded to the stages (the private method *commandStages* is empty): it is responsibility of sub-classes on the basis of the high-level parallel compositions they represent:

```
PRIVATE_METHODS:
METHOD: commandStages(anytype datain, Result res) RETURNS:
void;
BEGIN
END_METHOD;
```

The same manager can be used for more computations in parallel. This synchronisation policy can be expressed by two MaxPar synchronisation constraints specified in the SCHEDULING_PART section of the ComponentManager class (that would be inherited anyway):

```
SCHEDULING_PART:
            MaxPar (execution);   MaxPar (commandStages);
```

The abstract class ComponentStage defines the general structure of the stage together with its interconnections.
Each instance of this class is a generic stage: it maintains a reference to its slave object (in the variable obj) and the knowledge of the operation to be invoked (in the variable meth). In addition, the BOOLEAN variable am_i_last records whether the stage is connected or not to other stages. In the former case, references to those stages must be recorded in the state via the variable otherstages:

```
INSTANCE_STATE:
    ComponentStage otherstages [];
    BOOLEAN am_i_last;
    Object obj; String meth;
```

This abstract class defines the public operations *init* and *request*, and a private one, *commandOtherStages*.

The *init* operation extracts from the list (via the head[*] function) the reference to the slave object and the name of the method to be invoked:

```
METHOD: init (Association list[]) RETURNS: void;
    Association item;
BEGIN
 item  =  head(list); obj =  item.obj; meth =  item.meth;
    if (tail(list) == NULL) am_i_last = true;
END_METHOD;
```

If the list contains more than one pair (the tail function does not return NULL), the stage is connected with some other ones: the specialisation of this operation in concrete sub-classes provides the initialisation of the specific interconnections.
The *request* operation accepts both the input data to be passed to the slave object and the reference to the collector object.
By means of the EVAL primitive, the method name can be dynamically computed to properly send the synchronous service request to the slave object[†]:

```
METHOD: request (anytype datain, ComponentCollector res)
RETURNS:void;
    anytype dataout;
BEGIN
    dataout = obj EVAL (meth, datain);
    if (am_i_last) @ res put (dataout)
    else Self commandOtherStages(dataout, res);
END_METHOD;
```

This operation commands the slave object for its job. Once the stage receives the reply two cases are possible:

- the stage is connected with other stages (am_i_last == FALSE). In this case, the private operation *commandOtherStages* (empty in the abstract class) is called to command the *request* operation to the other stages, with the data just received from the slave object and the reference to the collector;
- the stage is not connected with other stages (am_i_last == TRUE). In this case the stage passes the data received from the slave to the collector via the put operation.

The scheduling policy allows, again, an infinite number of services of the *request* operation to be concurrently executed:

```
SCHEDULING_PART:
            MaxPar (request); MaxPar (commandOtherStages);
```

The concrete class ComponentCollector implements a multi-item buffer to store the result of a computation:

```
CLASS: ComponentCollector
```

---

[*] The reported C functions head, tail and cons have the same semantic of the car, cdr and cons functions of LISP [Set91].

[†] The primitive EVAL allows to ask an object for an operation whose name is to be computed at execution time: <object-reference> EVAL(<string>,<parameter-list>);
EVAL requests that the operation whose name corresponds to <string> is sent to the specified object and and behaves closely to the primitive perform:with: of Smalltalk.

```
INHERITS_FROM: /* Object by default */
INSTANCE_STATE:
    anytype content[];
INSTANCE_METHODS:
METHOD: put (anytype item) RETURNS: void;
BEGIN
    cons(content[],item);
END_METHOD;
METHOD: get () RETURNS: anytype;
    anytype result;
BEGIN
    result = head(content[]);
    content = tail(content[]);
    RETURN result;
END_METHOD;
SCHEDULING_PART:
            Sync(put,get); Mutex(put); Mutex(get);
END_CLASS ComponentCollector;
```

Let us note that the Sync(put,get) synchronisation constraints for this class permit the service of a *get* operation only if a greater number *put* operations has been already served. In addition the Mutex constraint makes only one request of a given method served at a time. Other collection policies can be implemented by subclassing from the ComponentCollector.

6.2. Pipe Classes

The PipeManager class, derived from the ComponentManager one, implements a pipelined communication pattern among a list of stages, instances of the PipeStage class (derived from ComponentStage). Any instance of the class PipeManager is in charge of creating only the first stage of the pipeline at its initialisation. During the execution of a service request, only the first stage has to be commanded.

```
CLASS: PipeManager
INHERITS_FROM: ComponentManager;
INSTANCE_METHODS:
METHOD: init (Association list[]) RETURNS: void;
BEGIN
    stages[0] = PipeStage Create(list[]);
END_METHOD;
PRIVATE_METHODS:
METHOD: commandStages(anytype datain, ComponentCollector res)
RETURNS: void;
BEGIN
    @ stages[0] request (datain, res);
END_METHOD;
END_CLASS PipeManager;
```

Any instance of PipeStage is in charge of creating the next stage of the pipeline during the initialisation phase. Then, in the execution of the *request* operation, any stage object directly commands the next one; the last PipeStage object sends the result to an instance of the class ComponentCollector (see above) whose reference to it is dynamically passed along the whole pipe, from stage to stage.

```
CLASS: PipeStage
INHERITS_FROM: ComponentStage;
INSTANCE_METHODS:
METHOD: init (Association list[]) RETURNS: void;
BEGIN
    Self Stage.init(list[]);
    if (! am_i_last)          otherstages[0] = PipeStage Create(tail(list[]));
END_METHOD;
PRIVATE_METHODS:
METHOD: commandOtherStages (anytype datain, ComponentCollector
res)
RETURNS: void;
BEGIN
    @ otherstages[0] request (datain, res);
END_METHOD;
END_CLASS PipeStage;
```

An efficient exploitation of inheritance simplifies the definition of the PipeManager and of the PipeStage classes: the *execution* and the *request* operations are fully inherited from their respective superclasses and only the private *commandStages* (PipeManager class) and the *commandOtherStages* (PipeStage class) operations, together with the *init* ones, have to be overridden. Moreover, no synchronisation problems arise in their definitions: the scheduling part is fully inherited from abstract superclasses.

6.3. Farm Classes

A first policy for a farm composition requires the manager of the composition to wait only for the first result made available by any of the stages:

```
CLASS: FarmManager
INHERITS_FROM: ComponentManager;
INSTANCE_STATE:
    int nWorker;
INSTANCE_METHODS:
METHOD: init (Association list[]) RETURNS: void;
    Association newlist[]; int i = 0;
BEGIN
    while (! (newlist[] = tail (list[])))
    {        stages[i++] = FarmStage Create(Cons ( head(list[]), NULL));
             list[] = newlist[]; }
 nWorker = i;
END_METHOD;
PRIVATE_METHODS:
METHOD: commandStages(anytype datain, ComponentCollector res)
RETURNS: void;
    int i;
BEGIN
    for(i=0; i< nWorker;i++) @ stages[i] request (datain, res);
END_METHOD;
END_CLASS FarmManager;
```

The concrete class FarmManager derives from the ComponentManager one. The *init* operation creates all needed stages, while the *execution* operation distributes data to all the stages and waits the first result available by the collector object. As in the pipe case, the *execution* operation and the synchronisation constraints are inherited from the ComponentManager abstract class.

The stages of the farm are instances of the concrete FarmStage class, whose definition completely derives from the abstract class ComponentStage:

```
CLASS: FarmStage
INHERITS_FROM: ComponentStage;
END_CLASS FarmStage;
```

Farm stages do not connect to each other (am_i_last is always TRUE and the *CommandOtherStages* operation, as in the abstract class, is left empty). For all the stages, the *request* operation is issued by the manager and the result of the computation is directly sent to the same collector and made available to the manager.
In addition, a more specialised farm can implement a different policy of treatment of the result: the manager waits for all the results produced by the stages. This policy is implemented in the FarmMultipleResultsManager class, derived from the FarmManager one:

```
CLASS: FarmMultipleResultsManager
INHERITS_FROM: FarmManager;
INSTANCE_METHODS:
METHOD: execution (anytype datain) RETURNS:anytype [];
ComponentCollector res;
int i;
futureVariable futureVar[];
BEGIN
    res = ComponentCollector Create();
    Self commandStages(datain, res);
    for (i=0; I < nWorker;i++) @ futureVar[i] res get();
    RETURN futureVar[];
END_METHOD;
END_CLASS FarmMultipleResultsManager;
```

The manager collects all replies in an array future-variables to be returned. To do that, we only have to redefine the *execution* operation to deal with an array of results, but there is no need either to change the *init* and *CommandStages* operation, inherited from the FarmManager class, or to define a more specialised class for the stages of the new farm.

6.4. Other Available Classes

We have implemented several other PO classes to implement different HLPCs, apart from the above described, such as different kinds of farms and geometric iterative patterns shaped in meshes. In all cases, their definition is simple because of inheritance. The implemented compositions have been available to users in the PO environment. Moreover, new compositions can be added when in need in the same way the presented compositions have been derived.

7. Evaluation

Performance is always the first parameter to evaluate in new proposals for the parallel computing area. This is the reason the HLPC approach, apart from its conceptual validity, must prove its feasibility toward high performance. To test their efficiency, we have implemented several simple applications by using both the basic patterns presented in previous sections and some other ones (different kinds of farms and mesh shaped geometric patterns). Their performances have been compared with those shown by the same application implemented in PO in a trivial way, i.e., without using the manager-stage-collector scheme and by adapting the behaviour of the slave objects to directly communicate following the desired communication pattern. The paper reports the results for the multiple-results farm and the pipe. In particular, we refer to a farm of 10 workers and to a pipe of 10 stages. Our experiences have shown that their behaviour is representative of a larger class of compositions. The target architecture used in the tests is a transputer-based one, the Meiko CS-1 [Mei89]: though providing very poor absloute performance, this architecture can still be considered a good platform for rapid prototyping and comparative evaluation.

7.1. HLPC Overhead

Active objects have several advantages: they do not limit parallelism of applications and can easily allow to implement asynchronous communication modes. However, a drawback of active object is their overhead: in fact, the service of any request implies the analysis of synchronisation constraints and the cost of generating a new thread. Thus, active objects are beneficial for medium- to coarse-grained parallelism. In any other case, the use of passive data structures can be preferred for performance's sake. In the Meiko CS-1 implementation of PO, the activation of a request for an operation of a 10 ms of duration* causes an overhead of about the 30%. The overhead obviously diminishes with the increasing of the operation duration: for operations of a 100 ms of duration it is about 3%.

This characteristic influences the performances of our HLPC implementation too, since the introduced overhead would reflect on it. Figures 5 and 6 report, respectively, the average response time of a request sent to HLPC pipe and farm compositions compared with the ones of trivial PO implementation.

The overhead of the HLPC is high when the operations on the slave objects are of short duration, i.e., the operations produce fine-grained parallelism. As soon as the duration of slave object operations increases, the difference of response time between the HLPC and trivial PO implementation diminishes until it disappears. For an average duration of 100 ms, we have found this difference is under the 10% of the absolute values for the pipe composition and under the 13% for the farm one. Comparing these values with the intrinsic overhead of activating services within a PO object, HLPCs show themselves efficient.

7.2. HLPC Parallel Behaviour

The PO HLPCs are intrinsically parallel in serving requests: it is interesting to evaluate the response time when several clients issue requests in parallel. Figures 7 and 8 report, respectively for pipe and farm patterns, that the parallel service of requests reduces the average response time, both in case of the trivial PO compositions and in case of HLPC ones. However, when the number of parallel requests grows - and before an expected saturation level - HLPC exhibits a better response time than the simplest PO implementation. By implementing a communication pattern without introducing any stage to control the communication flow (as it is in the case of the trivial composition), the objects in charge of receiving and controlling the service request are likely to become a bottleneck. HLPCs, instead, avoid bottlenecks by means of a more effective workload distribution among their components.

---

* The length of an activity can be measured by its execution time: even operations constituted of few instructions can be of long duration if they request services to other objects.

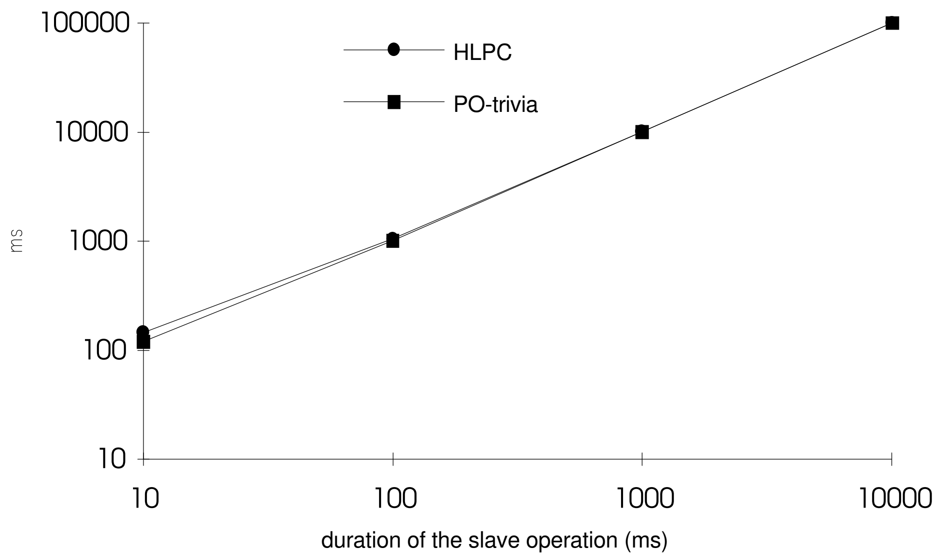

Figure 5. The overhead of the pipe composition

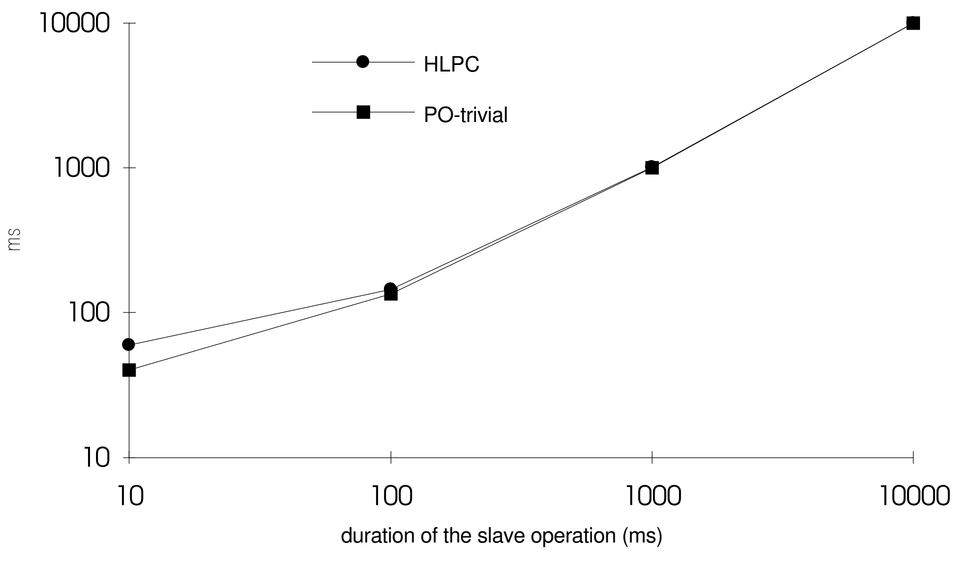

Figure 6. The overhead of the farm composition

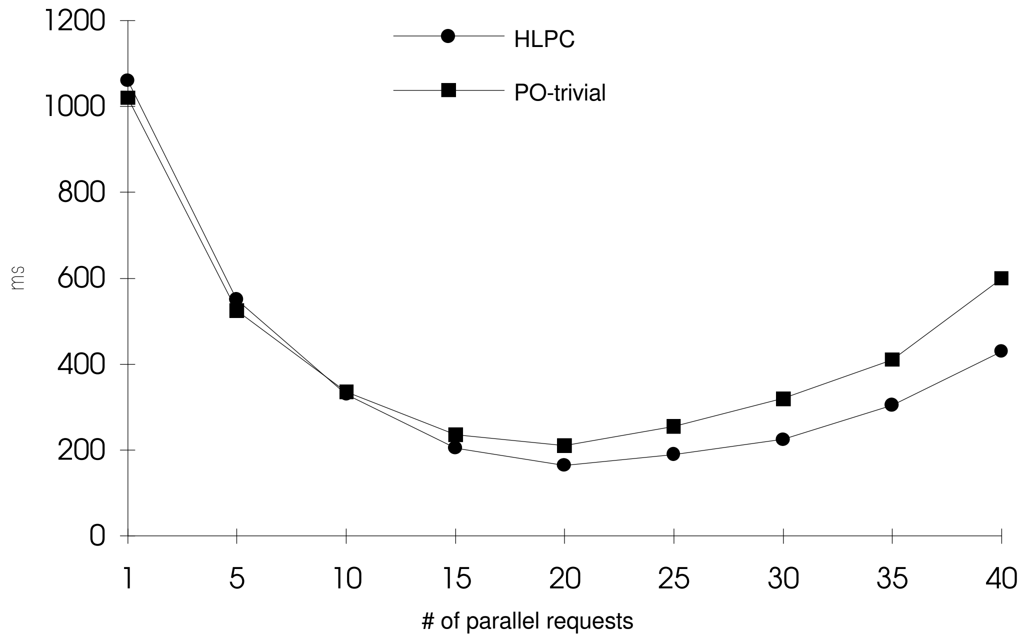

Figure 7. The pipe parallel behaviour (duration of the slave operation 100 ms).

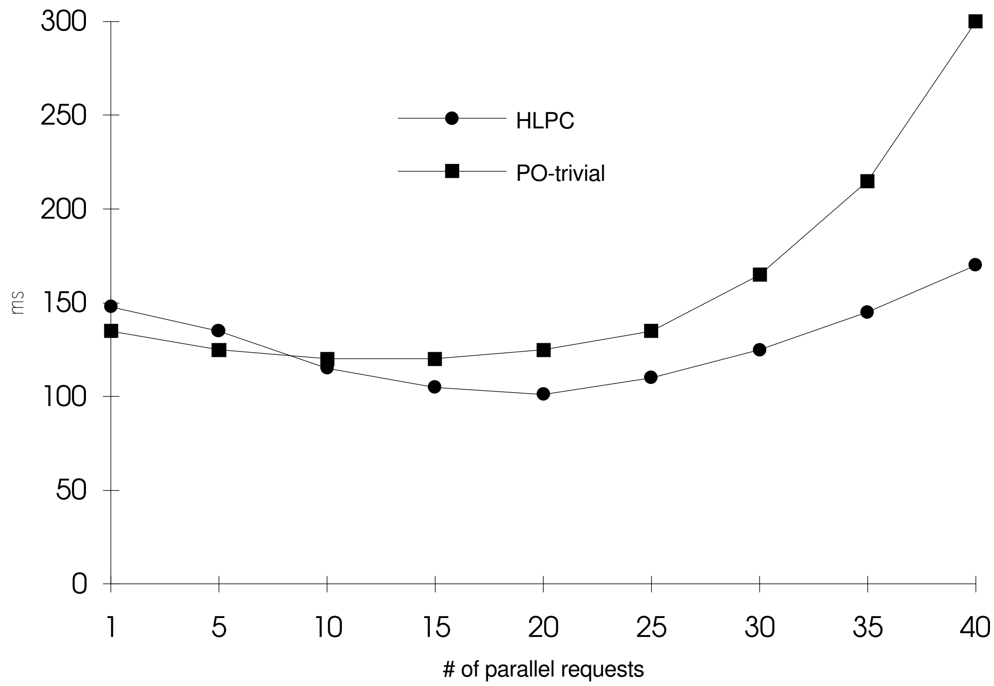

Figure 8. The farm parallel behaviour (duration of the slave operation 100 ms).

8. Conclusions

The paper describes a high-level structured approach to specify parallel applications. Instead of requiring the user to face all details of process creation and communication, it suggests a composition-based approach: the applications are built by choosing

among the available high-level parallel components and putting them together.
A definition of completeness is out of the scope of our approach: we rather want to define a framework where any needed composition can be seamlessly added to the available set. The paper shows that an object-oriented approach makes it possible an incremental scenario: new high-level parallel compositions can be derived by inheritance from already defined classes and can be offered at the same level of the already defined ones.
The HLPC style of composition allows firstly to easily achieve correct solutions, then an effective reuse in the definition of the parallel behaviour. Moreover, the approach is even efficient: the overhead is low and, in case of intensive traffic of requests, the proposed compositions exhibit not only a robust behaviour but also advantages over the non-structured approach.

9. References

[Adl95] R. M. Adler, 'Distributed Coordination Models for Client-Server Computing', *IEEE Computer*, **29** (4), April 1995.

[AhuCG86] S. Ahuja, N. Carriero and D. Gelernter, "LINDA and Friends', *IEEE Computer*, **19** (8), Aug. 1986.

[Bac95] B. Bacci et al., 'A Structured High-Level Parallel Language and its Structured Support', *Concurrency: Practice and Experience*, **7** (3), May 1995.

[Byw94] R. Bywater et al., 'Object-Oriented Simulation Software for Drug Design', *Conference on High-Performance Computing and Networking*, LNCS No. 796, April 1994.

[Bru93] D. Bruschi et al., 'A User-Friendly Environment for Parallel Programming', *Proceedings of the 1st EuroMicro Workshop on Parallel and Distributed Processing*, Gran Canaria (E), Jan. 1993.

[CACM95] 'Concurrent Object-Oriented Programming', *Special Issue of Communication of ACM*, **36** (9), Sept. 1993.

[ChiC91] R. S. Chin and S. T. Chanson, 'Distributed Object-Based Programming Systems', *ACM Computing Surveys*, **23** (1), Marzo 1991.

[Col89] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, The MIT Press, Cambridge (MS), 1989.

[CorL91] A. Corradi and L. Leonardi, 'PO Constraints as Tools to Synchronize Active Objects', *The Journal of Object-Oriented Programming*, **4** (6), Oct. 1991.

[CorLZ94] A. Corradi, L. Leonardi and F. Zambonelli, 'How to Structure Parallel Applications: Local Nested Aggregates', *Proceedings of the Joint Modular Languages Conference*, Ulm (D), Sept. 1994.

[Dar93] J. Darlington et al., 'Parallel Programming Using Skeleton Functions', *Proceedings of PARLE '93*, Munich (D), June 1993.

[ForA93] S. Frolung and G. Agha, 'A Language Framework for Multi-Object Coordination', *Proceedings of ECOOP '93*, Kaiserslauter (D), July 1993.

[FosC95] I. T. Foster and K. M. Chandy, ‘Fortran M: A Language for Modular Parallel Programming’, *The Journal of Parallel and Distributed Computing*, **26** (1), April 1 1995.

[Fro92] S. Frolung, ‘Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages’, of *Proceedings ECOOP '92*, July 1992.

[GolR83] A. Goldberg and D. Robson, *Smalltalk-80: the Language and its Implementation*, Addison-Wesley, 1983.

[Han93] P. B. Hansen, ‘Model Programming for Computational Science: a Programming Methodology for Multicomputers’, *Concurrency: Practice and Experience*, 5 (5), Aug. 1993.

[Jez93] J. Jezequel, ‘Transparent Parallelisation through Reuse: Between a Compiler and a Library Approach’, *Proceedings of ECOOP '93*, Kaiserslauter (D), July 1993.

[KerR88] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, second edition, Prentice-Hall, 1988.

[Mei89] Meiko, *Computing Surface Refeence Manual*, Meiko Ltd., 1989.

[Nie93] O. Nierstrasz, ‘Composing Active Objects’, Research Directions in Concurrent Object-Oriented Programming, MIT Press, Cambridge (MS), 1993.

[Set89] R. Sethi, *Programming Languages*, Addison-Wesley, 1989.

[Skil93] D. B. Skillicorn, ‘Models for Parallel Computation’, *Proceedings of the Advanced Workshop on Programming Tools for Parallel Machines*, Alimini (Italy), June 1993.

[Str91] B. Stroustrup, *The C++ Programming Language*, Addison Wesley, 1991.

[TPDP91] ‘Parallel Languages and Compilers’, *Special Issue of the IEEE Transaction on Parallel and Distributed Systems*, **2** (3), July 1991.

[Tsi91] D. Tsichritzis ed., *Object Composition*, University of Geneve, 1991.

[Weg90] P. Wegner, ‘Concepts and Paradigms of Object-Oriented Programming’, *ACM OOPS Messenger*, **1** (1), August 1990.

[ZimC93] H. S. Zima and B. M. Chapman, ‘Compiling for Distributed-Memory Systems’, *Proceedings of the IEEE*, **81** (2), Feb. 1993.