Design and implementation of communication patterns using Parallel Objects

OMITTED FOR BLIND REVIEW

Abstract: Within an environment of Parallel Objects, an approach of Structured Parallel Programming with the paradigm of Object–Orientation is presented here. The proposal includes a programming method based on High Level Parallel Compositions or HLPCs (CPANs in Spanish). C++ classes and CPANs are syntactically alike and differ in concurrency mechanisms. Different parallel programming patterns, synchronization operations and new constructs like futures have been discussed throughout the paper. To achieve software–reusability, a series of predefined patterns that use object–oriented programming concepts have been presented. Concurrency related constraints on process synchronization are set by only resorting to maxpar, mutex, sync primitives in the application code. By means of the method application, the implementation of commonly used parallel communication patterns is explained to finally present a library of classes for C++ applications that use POSIX threads.

Keywords: CPANS; High Level Parallel Composition, Parallel Objects; Communication Patterns; Structured Parallel Programming; High Performance Computing.


1 Introduction

As is known, there exist a great number of applications that try to obtain the system’s maximum performance by programming algorithms on uniprocessor computers, which—as today—can have many “cores”. When such system can not provide the performance that is required to obtain results within acceptable response times, a possible solution consists of opting for parallel processing. Parallel processing is therefore, an alternative to “uniprocessing” when performance limit of a system is reached. Sequential computation means that the central processor only carries out one operation at a time. On the opposite, in a parallel calculation, where several processors can cooperate to solve a given problem, the calculation time can be dramatically reduced since several operations are now carried out simultaneously. From a practical point of view, nowadays it is justified to carry out collaborative research in Parallel Processing together with related areas (Concurrency, Distributed Systems, Systems of Real Time, ...) since current advances in massively parallel systems, high–band communications, many and multicore processors allow us to carry out such investigation. Parallel Algorithmics, models and methodologies for Parallel Programming are in continuous development. To only mention some of the most excellent applications of Parallel processing at moment, we could say that it includes many topics such as computer architectures, algorithms, parallel programming languages and different methods of performance analysis.

The present investigation is mainly focused on Methods of Structured Parallel Programming, thus proposing an implementation of a new library of utility classes named High Level Parallel Compositions-HLPC or CPAN according to its Spanish acronym. We chose the name CPAN after the inspiring work of Corradi, A. Leonardo, L., Zumbonelli, F. (1995), proposing the HLPC structured patterns for doing structured parallel programming two decades ago. This paper recently had a follow-up publication by Danelutto M., Torquati S. (2014) to the initial HLPC construct and related method.

CPANs are based on the Object–Orientation paradigm to solve problems that are prone to parallelization by using a class of concurrent active objects that provide the programmer with the most common communication patterns in Parallel Programming, Capel et al. (1994). At the moment, the library includes the following patterns: Farm, Pipeline and Binary Tree. The latter one is used in the parallel version of Divide and Conquer algorithm design technique, Poldner M., Kuchen H. (2008), which is then applied to the implementation of a solution to the Traveling Salesman Problem (TSP). The TSP is then solved with the CPAN Farm that implements the Branch–and–Bound algorithmic design technique for...
solving NP-complete problems, see omitted4blind-review (2006), Capel et al. (1992).

2 Related Work

Currently the industry offers parallel hardware platforms such as GPUs, multi-core processors and the cloud, to speed up data processing with respect to uni-processor contention. For all these platforms performance and optimization of sequential algorithms is reaching its limit. One alternative is to opt for parallel and concurrent programming algorithms at a high level of abstraction by using patterns of communication/interaction between processes. The transformation of existing sequential applications into parallel ones for multiprocessors environments has been of great interest for decades. There is not however a solution of general application to solve the still pending issues regarding a sound parallelisation of algorithms and programs.

In Collins A. J. (2011) the effectiveness and applicability of automatic techniques has been explored. Six implementation parameters in the FastFlow parallel skeleton framework were tuned to obtain speed up of calculations. FastFlow is a C++ parallel programming framework intended to propitiate high-level, pattern-based parallel programming, as the research work of Torquati M., Aldinucci M. and Danelutto M. (2014) Aldinucci M., Danelutto M., et al. (2014) pointed out. FastFlow supports streaming and data parallelism and is targeted to perform parallel programming on heterogeneous environments, which are composed of clusters of shared-memory platforms, possibly equipped with computing accelerators such as NVidia GPUs, Xeon Phi, Tilera TILE64. The main design philosophy of FastFlow is to provide application designers with key features for parallel programming (e.g. time-to-market, portability, efficiency and performance portability) via suitable parallel programming abstractions and a carefully designed run-time support. FastFlow programming model is a structured parallel programming one. The framework provides several predefined, general purpose, customizable and composable parallel patterns or algorithmic skeletons such as the pipeline parallel pattern, the task-farm pattern, map or farm patterns for parallel loops and some valid compositions of pipeline and farm patterns, as described in the work of Torquati M., Aldinucci M. and Danelutto M. bis (2014). Any applications whose parallel structure can be designed using a set of provided parallel patterns, used alone or in composition, are susceptible of being implemented using FastFlow.

There are currently projects that develop frameworks and offer users constructs, templates and parallel communication patterns between processes, such as the ParaPhrase project Torquati M., Aldinucci M. and Danelutto M. (2015) aimed at developing a new structured design and implementation process for heterogeneous parallel architectures. Among these, it is worth to mention EC-STREP (7th EU Framework Program, 2011-2014), REPARA Torquati M., Aldinucci M. and Danelutto M. (2015) (Reengineering and Enabling Performance And powEr of Applications) aimed at defining a unified programming model for heterogeneous computers, RePhrase project (‘Refactoring Parallel Heterogeneous Resource-Aware Applications A Software Engineering Approach’) financed by the EU H2020 Program (2015–2018).

A more conventional approach to framework–based Parallel Programming provides application programmers with the possibility of obtaining loop parallelisation from sequential code, with a relatively small amount of programming effort. This is the approach followed in Danelutto M., Torquati S. (2014) with the “ParallelFor”. This skeleton is provided by the FastFlow framework to fill the existing gap between usability and expressiveness in the loop parallelisation facilities offered by frameworks such as OpenMP and Intel TBB.

It is worth to mention here the works carried out in Myoupo J.F., Tchendji V.K. (2014) that offers an efficient coarse parallel algorithm to solve the optimal binary search tree problem by using a binary tree as communication pattern between the processes involved. The absence of separation between the communication pattern (binary tree) and the algorithmic code (optimal binary search) to solve that problem is detrimental to the proposal’s usability. The work carried out in Ernsting S., Kuchen H. (2012) offers more flexible solutions and is very close to the results of the investigation presented here. The library skeleton “Muesli” that offers a simplified framework to perform Parallel Programming helps to find correct solutions to general problems. “Muesli” skeleton also allows us to write one application that can be executed with no change across a variety of parallel machines ranging from simple shared-memory multi-core processors to clusters of distributed-memory multi- and many-core processors, multi-GPU systems and GPU clusters. A possible objection of this work amounts to not giving information about the used methodology for integrating the proposed library in applications. Therefore, it becomes difficult to know which communication patterns are offered to developers and how low-level details, such as coordinating threads or synchronising processes, are hidden users.

MALBA Alba E., Luque G., et al. (2007) is another software tool intended for assisting in the solution of combinatorial optimization problems using generic algorithmic skeletons implemented in C++. Some aspects about MALLBA library software design are introduced in that article as well as details of the most recent implemented skeletons that are offering accurate computational results for a scheduling problem. Despite the fact that MALLBA mainly focuses on optimisation methods (exact, metaheuristic–based and hybrids) it provides some design directions of much use for achieving
different distributed platforms parallelizations (LAN, WAN...)

One possible way to solve the parallelization problem consist of considering certain patterns. This approach provides programmers with a predefined set of algorithmic templates that can be combined, nested and parameterised with sequential code to cope with the complexity that means to carry out the parallelisation of complex programs. The implementation of these skeletons in computer programs is currently a manual process that needs human expertise to choose the suitable implementation parameters for a good program performance.

Considering patterns as the fundamental algorithmic skeletons for deriving parallel programs has been an interesting way to solve the parallelization problem for high complexity algorithms since the seminal work of Bacci et al. Bacci et al. (1999). In this approach, low level entities are mixed with high level parallel programming structures. As result, the applications programmed with this scheme will loose portability from modular software development. The skeletons are common structures of parallel processing that become abstracted and then defined in a generic way. Each one is defined in function of certain parameters that, once substituted by their current ones, allow us to adapt the structure to a particular application context. Some environments of parallel programming, as the one called SkIECL. Steuwer, M., Kegel, P., Gorlatch, S. (2011), are based on skeletons and wrappers that make up the fundamental constructs of a coordination language, defining modules that encapsulate code written in a sequential language and 3 classes of skeletons: control, stream parallel, and parallel data. SkIECL applications can be composed to each other or nested to build complex parallel applications. SkIECL is an environment of integrated that allows quick programming of complex parallel applications in several architectures (distributed memory and shared memory, MPP, SMP, clusters of SMP, NOWs, meta-computers, etc.). It allows us to use notations of parallel programming (MPI, HPF, PVM, etc.) that have become an industry standard, as well as tools to trace, visualize the execution and evaluate the performance of parallel applications. And thus, the user can reuse big pieces of sequential code written in his favorite language (C, C++, F77, F90, Java, etc.), encapsulating it in modules. Also, the preprogrammed parallel code can be encapsulated in more parallel structures (using MPI and specialized libraries). The programming environment facilitates the design of the global structure of an application to the user, by means of a collection of typical parallel structures, named skeletons that can be instantiated and combined to define complex parallel applications. The skeletons can be used to coordinate and connect sequential or parallel modules encapsulated in wrappers to each other. These last ones are good to assure that parameter passing and data representation are consistent to each other module that makes up the parallel application.

Examples of commonly used skeletons are farms, i.e., a selection of workers processes that carry out a set of computation tasks; pipelines that are used to exploit the derived parallelism extracted from executing the different phases of a calculation simultaneously; and trees to which parallel Divide and Conquer techniques can be applied. Several Parallel Programming libraries and environments provide these skeletons. Regarding the latter ones, the advantage of SkIE amounts to allows composing the skeletons freely, and building more complex structures, and it is also able to generate optimized code for specific architectures. The development of parallel applications in SkIE is carried out through VisualSkIE, which is a graphic windows system, Steuwer, M., Kegel, P., Gorlatch, S. (2011). A user can define the global structure of his application in an interactive way. New sequential parts of applications can be modified using an editor integrated in the environment Already programmed sequential code can be encapsulated within the appropriate wrapper. By using explicitly, C and Fortran, besides MPI, or using the skeletons incorporated in SkI, the parallel structure of the application can also be defined.

3 Motivation

At moment the construction of concurrent and parallel systems has less restraints than ever, since the existence of parallel computation systems, more and more affordable, of high performance, or HPC (High Performance Computing) has brought to reality the possibility of obtaining a great efficiency in data processing without a great rise in prices. Even though, open problems that motivate research in this area still exist, efficient affordable parallel computing is a reality today. We are interested, in particular, to do research work that has to do with parallel applications that use predetermined communication patterns, among other component–software. At least, the following ones have currently been identified as important open problems:

- The lack of acceptance structured parallel programming environments of use to develop applications: structured parallelism is a type of Parallel Programming based on communication/interaction patterns (pipelines, farms, trees, etc.) that are predefined among the processes of a user application. Patterns also encapsulate parallel parts of the application, in such way that the user will only program the sequential code of the application. Many proposals of environments exist for the development of applications and structured parallel programs, but until now only a very limited circle of expert programmers use them. At moment, in HPC, a great interest exists in structured–parallel environments research, as the ones previously mentioned.
The necessity to have patterns or High Level Parallel Compositions: a high level parallel composition, as it is called in Corradi, A., Leonardi, L. (1991), or CPAN must be capable of being defined and used within an OO–infrastructure (language or environment of programming). Components of a parallel application do not interact in an arbitrary way, but they follow regular basic patterns Cole, M. (1989). An parallel programming environment must offer to its users a set of components that implement those patterns or CPANs most used in algorithms and parallel and distributed applications, such as trees, farms, pipes, etc. In order to develop programs and applications, the user in his turn must be able to compose and nest CPANs. The user can only use a set of predefined CPANs, but it does not mean self–constraining modelling capabilities at all, since he can adapt CPANs to the application requirements by means of the inheritance mechanism. Therefore, any environment development must include the concept of parallel objects class. Considerable interest exists in exploring the research line that aims at the definition of complete sets of patterns, including its semantic definition, for concrete classes of parallel applications.

**Determination of a complete set of patterns as well as of their semantics:** at that point, the scientific community does not seem to agree on any of the solutions that have been obtained to solve this problem today. Therefore, it does not seem easy to reach a sufficiently general and useful set of patterns. For example, a library of patterns or a set of constructs for programming languages, to be used in the development of parallel applications in a structured way.

**Adoption of an object–oriented approach:** Integrating a set of classes within an object–oriented infrastructure is a possible solution to the problem described in the previous point, since it would allow adding new patterns to an incomplete initial set by means of the subclasses definition. Therefore, one of the followed lines of research that we followed has been to find representations of parallel patterns as classes from which parallel objects (CPANs) can be instantiated, which are in their turn run as consequence of an external service request from the user application. For example, the derived pattern of the processes execution stage–by–stage would be defined by pipeline pattern class; the number of stages and the sequential code of each specific stage can not be established until a parallel object of this class is created; data to be processed and the results are obtained from the user application. Internal storage in the stages could be adapted in a subclass that inherits of the pipeline pattern instance. Several advantages are obtained when an OO–approach is followed, Corradi, A., Leonardi, L. (1991), with respect to an approach based on algorithmic skeletons and model–programs Cole, M. (1989) only, among the major improvements of the first approach we can mention the following ones:

- **Uniformity:** All the entities within the programming environment are objects. As consequence of that, all the entities of lower level than objects, as the communication channels are in the model of parallel and distributed programming proposed in Cole, M. (1989) will not be considered. In that way, the parallel patterns of communication cab be now described by means of a class, as it happens with any other software component necessary for the development of the applications, which are object of our interest in this study;

- **Genericity:** Within an OO software development environment, the capability of dynamically generating references makes possible the creation of generic patterns. The latter ones can be obtained by the definition of its components as generic references to objects. The specification of the parallel part can be separated from any functional specification. For example, a pipeline can be defined without specifying what will be performing the nodes that make it up.

- **Reusability:** The inheritance mechanism simplifies the definition of specialized parallel patterns. The inheritance applied to the behavior of a concurrent object helps in the specification of a pattern’s parallel behavior. For instance, if the environment initially provides a Farm pattern whose object controller waits after the first petition of service, then we can obtain a very simple implementation of the same one that specializes the initial behavior by resorting to the inheritance mechanism and defining a new farm that reacts to all the service–requests of a concrete application.

To solve the open problems has been the main motivation to pursue our investigation on communication patterns that we are going to define in next sections. In particular, the pattern of High Level Parallel Compositions or CPANS has deserved special interest from us. The fundamental objective is to offer a library of classes based on most representative and useful CPANs. The library of classes can be used in parallel applications within a programming environment based on implemented parallel objects omitted4blind-review (2008).
4 Exposition of the problem

The problem to tackle amounts to define a Parallel Programming Method based on High Level Parallel Compositions (CPANS), Corradi, A, Leonardo, L., Zambonelli, F. (1995) that can be of use in structured Parallel Programming. In order to achieve this, we have identified the following properties as indispensable requirements to be fulfilled by the programming language and development environment. It is required, in principle, a environment of OO–Programming that provides the following features:

1. Method invocation capability of objects that offers asynchronous communication mode and asynchronous future to client applications. The asynchronous communication mode does not force to wait for a result the client application that invokes an object method. The asynchronous future communication construct makes the client to wait only when it needs that result in any execution future instant. The above two communication constructs (asynchronous and future) allow concurrent execution of the client with the called method execution (parallelism inter-objects).

2. Objects must can have internal parallelism. A mechanism of threads, Birrell, Andrew (1989), must allow an object to concurrently serve several invocations of their methods (parallelism intra-objects).

3. Availability of synchronization mechanisms when parallel service requests will take place. It is necessary that objects can negotiate several execution flows concurrently and, at the same time, to guarantee the consistency of their data.

4. Availability of flexible mechanisms of control of types. The capacity of dynamic type association to the parameters of object methods. It is needed that the system can negotiate types of generic data, since CPANs only define the parallel part of an interaction pattern, therefore, they may be able to adapt themselves to different classes of possible pattern components.

5. Transparency of distribution of parallel applications. It must provide the transport of the applications from a centralized system to a distributed system without being affected the client application code. The classes must maintain their properties, independently from the execution environment of the application objects.

6. Performance. This is always the most important parameter to consider when one makes a new proposal of development environment for parallel applications. A proposal based on patterns that are represented as classes and parallel objects must solve the PPP–problem: Programmability, Portability, Performance, so that it can be considered as an excellent approach to the previously outlined problems solving.

The OO–programming language and platform that has been considered as appropriate to satisfy the mentioned properties is C++ together with the standard POSIX threads on the operating system Linux.

4.1 Scientific Objectives of Interest

The development of a programming method based on High Level Parallel Compositions or CPANs that implements a library of classes of utility in the Concurrent/Parallel Object Oriented Programming. The method must provide the commonly used parallel patterns of communication, in such a way that the programmer can exploit the usual mechanisms of inheritance and parametrization of OO programming languages to define new patterns according to each CPAN specific pattern. This study is aimed at achieving the following specific objectives:

- To develop a programming method based on High Level Parallel Compositions or CPANs;
- To develop a library of classes of Parallel Objects that provides the user the patterns (under the pattern of the CPAN) more commonly used for structured Parallel Programming;
- To offer the library to the programmer so that, with a minimum knowledge of Parallelism and Concurrency, he can exploit under the paradigm of Object Orientation the features and performance of parallel code and, by means of different reusability mechanisms, to also define his own patterns adapted to the communication structure of his application processes.

To transform well–known algorithms that solve sequential problems (and that they can be easily parallelizable) in their parallel/concurrent version to prove the proposed methodology and the good functioning of the component software developed to carry out the study.

5 High Level Parallel Compositions or CPANS

Some of the problems of parallel programming environments amount to the users acceptance, which usually depends on whether they can offer complete expressions of the parallel programs behavior that are built with these environments Corradi, A, Leonardo, L., Zambonelli, F. (1995). At moment in OO application systems, the scientific community interested in the study of Concurrency only accept standards for programming environments based on parallel objects. A first approach
that tries to tackle this problem is to let the programmer to develop his programs according to a sequential programming style, then he can automatically obtain the parallelized parts of the code with the help of a specific environment.

However, intrinsic implementation difficulties exist mainly due to the difficult definition of programming languages formal semantics that refrain from the automatic (without user participation) sequential code parallelization, and thus the problem of generating parallelism in an automatic way for a general application continues unsolved. The so called structured parallelism has become a promising approach to solve the mentioned problem. In general parallel applications follow predetermined patterns of execution. Communication patterns are rarely arbitrary and are not structured in their logic Brinch-Hansen (1993). High Level Parallel Compositions or CPANs are parallel patterns defined and logically structured that, once identified in terms of their components and of their communication, can be adopted in the practice and be available as high level abstractions in user applications within an OO-programming environment. The process interconnection structures of most common parallel execution patterns, such as pipelines, farms and trees can be built using CPANs, within the work environment of Parallel Objects that is the one used to detail the structure of a CPAN implementation.

5.1 Structured Parallelism

A structured approach to parallel programming is based on the use of communication/interaction patterns (pipelines, farms, trees, etc.), which are predefined structures of user’s application processes. In such a situation, the structured parallelism approach provides the interaction–pattern abstraction and describes applications through CPANs, which are able to implement the patterns mentioned already. The encapsulation of a CPAN should follow the modularity principle and it should provide a base to obtain an effective reusability of the parallel behavior to be implemented. When there is the possibility of attain this, a generic parallel pattern is built, which in its turn provides a possible implementation of the interaction structure between processes of the application, independently of the functionality of these. The structured approach for parallel programming has basically followed two ways over the last years:

- The enrichment of traditional parallel environments with libraries of program skeletons that concrete communication patterns represent.
- The definition of restrictive and closed parallel languages that provide communication in terms of the patterns that are already defined in the system Bacci et al. (1999).

The approach presented here responds to the first one, since it is generally considered more generic and more open than the second one. What it really means is a new design approach to parallel applications. Instead of programming a concurrent application from the beginning and controlling the creation of processes as well as the communications among them, the user simply identifies those CPANs that can implement the adapted patterns to the communication needs of his application and uses them together with the sequential code that implements the computations that individually carry out their processes. Several significant and reusable parallel patterns of interconnection can be identified, but there is currently no general agreement to formally define their formal semantics Corradi, A., Leonardo, L., Zambonelli, F. (1995). For instance, the Farm pattern is a concept that can be understood by the ample majority of its users, but its instantiation to a particular application forces its users to choose among different strategies for its implementation.

5.2 The Object Orientation

Sometimes the lack of consensus on parallel patterns semantics implies that its formal definition becomes quite complex and only can be performed at a low implementation level. Therefore, users have to go into the architecture details of the system when they need to use a specific pattern in a concrete program. However, in a development environment of extensible software, as the OO environments are, the programmer can end up defining any parallel pattern that he needs via a high level language or graphic tool that supports the paradigm, and later on to adapt the pattern to the characteristics of a concrete application by resorting to the inheritance mechanism or genericity. The basic characteristic of these systems is the definition of context independent modules that can be connected to each other via high level communication channels. Thus, obtaining parallel compositions means to have statically defined communication patterns that can be independently built from their context as reusable modules, then providing in that way the parallel behavior encapsulation and module nesting capability. The basic idea is to define to the CPANs as objects in charge of controlling and coordinating the execution of their internal components. Living up to this premise an environment of extensible software development can be created that, based on CPANs, will provide the important characteristics of uniformity, generality and reusability of software systems.

5.3 Definition of the pattern CPAN

The basic idea here is to use classes to implement any type of parallel communication patterns between the processes of an application or distributed/parallel algorithm, thus following the Object Orientation. Starting from these classes, an there will be objects (class instances) and the execution of any object method in can be carried out through a service request. A CPAN
comes from the composition of a set three object types: An object manager (Figure 1) that represents the CPAN itself and makes an encapsulated abstraction out of it that hides the internal structure. The object manager controls a set of objects references, which address the object Collector and several Stage objects and represent the CPAN components whose parallel execution is coordinated by the object manager.

The objects Stage (Figure 2) are objects of a specific purpose, in charge of encapsulating an client-server type interface that settles down between the manager and the slave–objects. These objects do not actively participate in the composition of the CPAN, but are considered external entities that contain the sequential algorithm that constitutes the solution of a given problem. Additionally, they provide the necessary inter–connection to implement the semantics of the communication pattern which definition is sought. In other words, each stage should act a node of the graph representing the pattern that operates in parallel with the other nodes. Depending on the particular pattern that the implemented CPAN follows, any stage of it can be directly connected to the manager and/or to the other component stages.

The Collector object (Figure 3) we can see an object in charge of storing the results received from the stage objects to which is connected, in parallel with other objects of CPAN composition. That is to say, during a service request the control flow within the stages of a CPAN depends on the implemented communication pattern. When the composition finishes its execution, the result does not return to the manager directly, but rather to an instance of the Collector class that is in charge of storing these results and sending them to the manager, which will finally send the results to the environment, which in its turn sends them to a collector object as soon as they arrive, without being necessary to wait for all the results that are being obtained.

5.4 Composition of the CPAN

If we observe the scheme as a black box, the graphic diagram of a CPAN representation would be the one that is shown in Figure 4.

In summary, a CPAN is composed of an object manager that represents the CPAN itself, some stage objects and an object of the class Collector, for each petition that should be managed within the CPAN. Also, for each stage, an slave object will be in charge of implementing the necessary functionalities to solve the sequential version of the problem being solved (Figure 5).

The Figure 5 shows the pattern CPAN in general, without defining any explicit parallel communication
5.5 The CPAN seen as a composition of parallel objects

Manager, collector and stages are included in the definition of a Parallel Object (PO), Corradi, A, Leonardo, L., Zambonelli, F. (1995). Parallel Objects are active objects, which is equivalent to say that these objects have intrinsic execution capability. Applications that deploy the PO pattern can exploit the inter-object parallelism as much as the internal or intra-object parallelism. A PO-instance object has a similar structure to that of an object in Smalltalk, and additionally defines a scheduling politics, previously determined that specifies the way in which one or more operations carried out by the instance synchronize. Synchronization policies are expressed in terms of restrictions; for instance, mutual exclusion in reader/writer processes or the maximum parallelism allowed for writer processes. Thus, all the parallel objects derive from the classic definition of a class plus the synchronization restrictions (mutual exclusion and maximum parallelism), which are now included in that definition. Objects of the same class share the specification contained in the class of which are instances. The inheritance allows objects to derive a new specification from the one that already exists in the superclass. Parallel objects support multiple inheritance in the CPAN model.

5.6 Communication types in the parallel objects

Parallel objects define 3 communication modes: synchronous, asynchronous communication and synchronous future communication.

1. The synchronous communication mode stops the client activity until it receives the answer of its request from the active server object. The notation: \texttt{ref obj.name meth ([lista param])} is a clear syntax construct that facilitates its use in application programming.

2. The asynchronous communication does not delay the client activity. The client simply sends the request to the active object server and its execution continues afterwards. Its use in application programming is also easy, because it is only necessary to create a thread and start it to carry out the communication independently from the client. We will use the following notation to refer to this communication primitive: \texttt{Thread ref, obj.name meth ([lista param]);} where an instance of Thread is started to execute the method \texttt{name meth([lista param])} addressed by \texttt{ref, obj}.

3. The asynchronous future will delay client activity when the method’s result is reached in the client’s code to evaluate an expression. The asynchronous futures also have a simple use, though its implementation requires of a special care to get a syntactical construct with the correct required semantics. The notation used for futures in the CPAN model is given by: \texttt{FutureType futureVar = ref Obj.name ([lista param])} that expresses the generation and future result assignment when the function is invoked by an object reference. Where \texttt{FutureType} defines the type of future and \texttt{Anytype ResulVar = ref Obj.futureVar;} is used for type conversion of the future returned by the function when it conforms to \texttt{AnyType}. The word \texttt{ANYTYPE} is used to suggest the use of any type that is of interest for the user application.

The asynchronous and asynchronous future communication modes carry out the inter-objects parallelism by executing the client and server objects at the same time.

5.7 The asynchronous future communication mode

Term “future” has been used to identify the mechanisms related to the generation of, and access to, a specific value returned by a function call, which is accessed through an object reference. The word \texttt{future} is used to suggest that the returned value—if not immediately available—can be obtained some time later, i.e., in the future. Because the importance of the element

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Internal structure of a CPAN. Composition of its components}
\end{figure}
“time”, the returned value carries on an associated synchronization mechanism that is added to the actions that are performed for type checking. Thus, we can define a future as a synchronization mechanism that returns a typed result, which is used to represent a value that can be available at some later instant, after the creation of the future Lavander, Greg R., Kafura, Dennis G. (1995). The OO programming languages, in this case C++, together with the standard POSIX Thread for the handling of threads usually provide the necessary tools to program more or less the futures in an easy way. Next, the implementation of a future type as a class and referred as a future of ANYTYPE is shown.

CLASS FutureType
{
    ANYTYPE res;
    
    PUBLIC VOID operator=(ANYTYPE fut)
    {
        MUTEX(this);
        res=fut;
    }

    PUBLIC operator ANYTYPE()
    {
        SYNC(operator=,this);
        RETURN res;
    }
};

Overriding of the assignment operator is necessary to be able to assign the function’s return value to a variable of FutureType. And thus, a producer process can only carry out this assignment each time, that is to say, a mutual exclusion (Mutex) synchronization is performed between parallel producer processes that are willing to assign the returned value of the function they are executing to a future. Also, any attempt of using that returned value is blocked on the side of consumer processes (when the type conversion operation is carried out) until the value has been assigned to the future’s instance variable. To inform that the variable is available, a traffic light exists (Sync) that serves to establish a synchronization restriction between producers and consumers trying to use the variable at the same time. Type conversion conversion is used to convert an object of type FutureType to a ANYTYPE value that can be used in the computations carried out by the code of the user’s application. The declaration of ANYTYPE for a future variable initially blocks, which can be interpreted as the variable value is not available to the variable in that moment, but will be in the future. That is to say, when the variable value is not available for immediate processing, it will be so in the future when it will be prepared. Future’s type will change to the value’s type that the application code needs for continuing processing, for instance, one syntactical expression that includes the future as one of its terms or factors.

5.8 Semantic and Syntactic definition of the base classes of any CPAN

As it has already been described, a CPAN comes from the composition of a set of objects of three types. In particular, each CPAN is made up of several objects: an object manager, some stage objects and a collector object for each request sent by client objects of the CPAN. Also, for each stage of the CPAN, a slave object will be in charge of implementing the sequential part of the computation that is sought and carried out in the application or in the distributed and parallel algorithm. In PO the necessary base classes to define the manager, collector, stages objects that compose a CPAN are the next ones:

Abstract class ComponentManager: It defines the generic structure of the component manager of a CPAN, from which will be derived all the manager instances depending on the parallel behavior that is assumed in the CPAN creation. All specific instances of a manager accept a list of n-associations as input. An association is a pair of elements, that is, an object slave and the name of the method that has to be executed by this object. The objects slaves are external entities that contain a sequential algorithm that have to be executed by one of their methods. Once the manager has obtained the list of n-associations, it will generate the concrete stages, one for each association and then each stage becomes responsible for an object slave together with its execution method. In turn, each stage is connected to each other, in accordance with the parallel pattern that has been implemented in the CPAN. Finally, the manager carries out a computation by the execution of one of its methods. To achieve the computation phase, it is necessary to pass on the input data that it requires to start to the method. The manager then generates a component collector and sends its reference to the stages, as well as the input data. The stages start processing the data according to the connection configuration that they keep to each other, results will be passed on as they become available. At the end the collector will gather the results sent by the stages to return them to the manager, which finally will transfer these results to the CPAN environment or to the code that uses them.

CLASS ABSTRACT ComponentManager
{
    ComponentStage[] stages;

    PUBLIC VOID init (ASOCIACION[] list)
    {
        ABSTRACT;
    }

    PUBLIC ANYTYPE execution(ANYTYPE datain)
    VAR
        ComponentCollector res;
    {

res = ComponentCollector CREATE();
commandStages(datain,res);
RETURN res.get();
}

PRIVATE VOID commandStages(ANYTYPE datain,
ComponentCollector res)
{
    ABSTRACT;
}
MAXPAR (execution);
;

The same manager can be used to carry out more than one calculation in parallel. The synchronization policy used for is understood as the synchronization constraint due to the maximum parallelism or MAXPAR applied to the “execution()” method.

Abstract class ComponentStage: It defines the generic structure of the component stage of a CPAN, as well as their interconnections, from which will be derived all the concrete stages depending on the parallel behavior that is assumed in the creation of the CPAN. All specific instances of a stage accepts a list of associations slave–object/method as input to work with them, whether they are connected or not with the following stage of the list of associations and depending on the parallel pattern they are willing to implement. When the manager send in parallel a command to the stages, each one of them makes the object–slave to carry out the execution of its method, then the stage captures the results and sends them to the following stage or to the collector, depending on the implemented structure, i.e., any code called request() and programmed in a method. In turn, each stage can command others in the execution of the computation initiated by the manager.

CLASS ABSTRACT ComponentStage
{
    ComponentStage[] otherstages;
    BOOL am_i_last;
    METHOD meth;
    OBJECT obj;

    PUBLIC VOID init (ASOCIACION[ ] list)
    VAR
        ASOCIACION item;
    {
        item = HEAD(list);
        obj = item.obj;
        meth= item.meth;
        if (TAIL(list) == NULL)
            am_i_last = true;
    }

    PUBLIC VOID request (ANYTYPE datain,
                            ComponentCollector res)
    VAR
        ANYTYPE dataout;
    {
        dataout = EVAL (obj,meth, datain);
        IF (am_i_last)
            TREAD res.put(dataout)
        ELSE commandOtherStages (dataout, res);
    }

    PRIVATE VOID commandOtherStages(ANYTYPE dataout, ComponentCollector res)
    {
        ABSTRACT;
    }
}
MAXPAR (request);
;

Again, as in the previous case, it can have more than one request carried out by the operation “request()” and executed in parallel by the slave–objects. Thus, for the sake of synchronization policies, the one that must be used for correct implementation is that which provides the synchronization restriction of “maximum parallelism” and applied now to the “request()” method.

Concrete class ComponentCollector It defines the concrete structure of the component collector of any CPAN. This component fundamentally implements a multi-item buffer, where it will store the results of stages that have the reference of this collector. This way one can obtain the result of the calculation initiated by the manager.

CLASS CONCRETE ComponentCollector
{
    VAR
        ANYTYPE[] content;

    PUBLIC VOID put (ANYTYPE item)
    {
        CONS(content, item);
    }

    PUBLIC ANYTYPE get()
    VAR
        ANYTYPE result;
    {
        result = HEAD(content[]);
        content = TAIL(content[]);
        RETURN result;
    }
    SYNC(put,get);
    MUTEX(put);
    MUTEX(get);
};

In this particular case the synchronization restrictions used are SYNC and MUTEX, needed to correctly synchronize the concurrent communication
between methods put() and get() and to provide mutual exclusion, respectively.

An instance of a specific PO–class derived from the class ComponentManager represents a CPAN (called manager) in the application code, programmed according to the parallel objects pattern. The instances (called stages) derived of class ComponentStage are connected to each other to implement the composition of stages. Each stage commands the execution of an object PO, called slave (slave) that is controlled by its own stage. The creation of the stages and of the collectors and their later interactions are transparently managed in the application code by the manager. From the point of view of a user already interested in reuse the parallel behavior defined in some classes CPAN, the class of interest will be that of the manager. When an user is interested in using a CPAN within an application, he has to create an instance of a class manager specific, it is, one that implements the parallel behavior needed by the application and that initializes it with the reference to the objects slaves that will be controlled by each stage and the name of the requested method. The following syntactic definitions have been written using the grammar free of context that is in the appendix A of the present document.

5.9 The Synchronization restrictions MaxPar, Mutex y Sync

It is necessary to have synchronization mechanisms available when parallel request of service take place in a CPAN, so that the objects that conform it can negotiate several execution flows concurrently and, at the same time, guarantee the consistency in the data that being processed. Within any CPAN the restrictions MAXPAR, MUTEX and SYNC can be used for correct programming of their methods.

5.9.1 MaxPar

The maximum parallelism or MaxPar is the maximum number of processes that can be executed at the same time. That is to say the MAXPAR applied to a function represents the maximum number of processes that can execute that function concurrently. In the case of CPAN, the maximum parallelism is applied to the function “execution()” of the ComponentManager class and to the function “request()” of the ComponentStage class. In the first case it means that the maximum number of processes that will be able to execute the function “execution()” in parallel will be at the most that one of representing the number of objects that have been created by the manager. For example, if two manager objects are generated and one of them solves three different problems, while the other ones solve five in total, it means that there will be eight processes prepared to execute the “execution()” function in parallel; however when applying the MAXPAR, the number of processes concurrently in “execution()” will be limited to a maximum of two. The other processes will remain blocked, defining a synchronization communication between those in execution and the ones that wait for it. In the second case the maximum number of processes that will be able to execute the function “request()” in parallel will be the number of stage objects that have been created by an object manager at maximum. For example, if an object manager (that is to say a CPAN) solves two different problems and for each problem three stage objects are created, then it means that there will be six processes wanting to execute in parallel the function “request()” to satisfy the service request made by the manager that commands them; however, when the MAXPAR is applied, the concurrent execution of “request()” will be limited to a maximum of three processes, being blocked the remaining ones, as it was discussed in the previous case. The implemented algorithm of the MAXPAR synchronization constraint is the following one:

1. To incorporate a class variable in the class of interest and to initialize it to zero.

2. Every time that an instance of that class of interest is generated, it will be necessary to update its class variable by adding one to the value that had previously, so that the number of created instances will correspond to the maximum number of processes that can execute a particular function.

3. To implement, in the function of interest, a semaphore in the following way:

   (a) If n is the maximum number of processes that can execute the function of interest and if k is the number of processes that are executing at this moment, then:

   (b) If k less than n, then execute the k processes in parallel, but block the remaining k–n processes and don’t wake them up until at least one of the n–processes that are executing has finished so that always the limit of at most n processes being executed concurrently is guaranteed.

5.9.2 Mutex

The restriction of synchronization mutex carries out a mutual exclusion among processes that want to access to a shared object. The mutex preserves critical sections of code and obtains exclusive access to the resources. In the case of the CPANs, the restriction mutex applied to a function represents the use of that function on the part of a process every time. In other words, the mutex allows that only one of the processes executes the function, blocking all the other processes trying to make use of the service until one of the ones that execute it finishes. The mutex within the CPAN is applied to the function get() as well as to the function put() of an object collector. The algorithm that implements the synchronization restriction Mutex is the following one:
1. To use a condition variable that works as a flag and a lock mutex.

2. To initialize the flag to false

3. To implement a semaphore, in the function of interest, in the following way:
   
   (a) The process that acquires the lock will put the flag to true and will be able to execute the function of interest.

   (b) While the flag is true all the other processes that try to acquire the resource to execute the function will be blocked and will wait until the flag is false and then some of them can acquire the lock.

   (c) The process which has got the lock at any moment will execute and do its work.

   (d) When its execution is finished, it will change the flag value to false and the lock will be free again.

   (e) Processes that are at the beginning on the wait line, will acquire the lock repeating the sequence from the step (a).

5.9.3 Sync

The restriction SYNC is not more than a producer/consumer type of synchronization; it is of use, for instance, for programming the methods put() and get() of the componentCollector class. SYNC helps to synchronize these methods, so that the method get() can be executed provided the method put() will make sure that at least one element has been placed in the shared container. Or equivalently, when the shared buffer is empty the process that executes get() will be blocked until the notification of the next execution of the method put() is issued. However, the method get() does not need to confirm to any put()-executing process whether it has used or not an element of the container that they share, since in the implementation that has been carried out, such a container does not have a predefined maximum size, but otherwise it dynamically goes growing according to the application to needs. Only when when the method get() has not obtained any element from the buffer is needed a notification to the pairing process. Therefore, the container will never be full of items and the implementation algorithm is the following one:

1. To use a condition variable (e.g.: n_items) that counts the number of elements placed in a container.

2. To initialize n_items to zero value.

3. To implement a semaphore shared by the functions get() and put() used by consumer and producer processes in the following way:
   
   (a) A consumer process will remain blocked while n_items is equal to zero, because with it is true that the producer process has not placed any data in the shared container.

   (b) A producer process will acquire the lock, place a data in the container, and the n_items value will increase by one.

   (c) The consumer process will be woken up and it will continue the execution given there are more data to be processed.

   (d) When n_items is bigger than zero, the consumer will acquire the lock; it will lower by one the n_items value and notifies processing of one datum out of the container.

   (e) Processes that are at the beginning on the wait line, will acquire the lock repeating the sequence from the step (a).

6 Design and Construction of the CPANs

Farm, Pipe y TreeDV

With the base–classes of the PO model of programming, it is now possible to build concrete CPANs. To build a CPAN, first it should have made clear the parallel behavior that the user application needs to implement, so that the CPAN becomes this pattern itself. Several parallel patterns of interaction have long been identified in Parallel Programming, such as farms, pipes, trees, cubes, meshes, a matrix of processes, etc. Once identified the parallel behavior, the second step consists of elaborating a graph of its representation, as an informal design of the objective system. This practice is also good for illustrating the general characteristics of the desired system and will allow us to define its representation with CPANs later on, by following the pattern proposed in the previous section. When the model of a CPAN has already been made clear, it defines a specific parallel pattern; let's say, for example, a tree, or some other mentioned pattern, then the following step will be to do its syntactic definition and specify its semantics. Finally, the syntactic definition prior to any programmed CPAN is transformed into the most appropriate programming environment, with the objective of producing its parallel implementation. It must be verified that the resulting semantics is the correct one. To attain this, we use several different examples to demonstrate the generality and flexibility of the application CPAN–based design and the expected performance and quality as a software component. Some support from an integrated development environment (IDE) for Parallel Programming should be provided in order to validate the component satisfactorily. The parallel patterns worked in the present investigation have been the pipeline, the farm and the treeDV, which can be considered as a significant set of patterns of use in multiple applications and algorithms.
1. **Pipeline**, as it is so named a composition of a set of interconnected states, one after another. The information flows through each state, from the first state to the last one.

2. **Farm**, it is made up of a set of worker processes and a controller. The workers are executed in parallel until they reach a common objective. The controller is the one in charge of distributing the work among the workers and to control the global calculation progress.

3. **TreeDV**, in this case the information flows from the root towards the leaves or vice-versa. The nodes that are on the same tree level are executed in parallel by deploying the design technique named Divide and Conquer to obtain the algorithm-solution of the problem.

These parallel patterns make up the fundamental library of classes that the CPAN pattern proposes.

### 6.1 CPAN Pipeline

It represents the aforementioned pipeline technique of parallel processing as a High Level Parallel Composition or CPAN, applicable to a wide range of problems that are partially sequential intrinsically. The CPAN Pipe guarantees the parallelization of sequential code using the pattern Pipeline.

#### 6.1.1 The Pipeline technique

By using the Pipeline parallel processes design technique, the problem becomes divided into a series of tasks that have to be completed with a sequential dependency between each and the next one, i.e., one after another. In a pipeline each task can be executed by a process, thread or processor independently, Roosta, Seller (1999), (Figure 6).

![Figure 6 Pipeline](image)

The pipeline processes are sometimes called stages of the pipeline, omitted4blind-review (2005). Each stage can contribute to the solution of the total problem and it can pass on the information that the following stage of the pipeline needs. Many times, this type of parallelism is seen as a form of functional decomposition. If Divide-and-Conquer is applied, the problem solution is decomposed into separate functions that can be individually executed, hopefully in parallel. With this technique, there is a sequentiality over imposed to the solution since functions must be executed in succession.

Figure 7 represents the Pipeline parallel pattern of communication as a CPAN.

![Figure 7 The CPAN of a Pipeline](image)

The stage-object i and the Manager on the graph pattern of the CpanPipe are instances of concrete classes that inherit the characteristics from the ComponentManager and ComponentStage classes, respectively.

#### 6.1.2 Semantic and Syntactic definition of the Cpan Pipe

The Cpan Pipe is represented by the class PipeManager that inherits from ComponentManager, and a communication pattern pipeline implements those stages as instances of the class PipeStage that inherits from ComponentStage. Any PipeManager object only takes charge of the first stage of the pipeline during its initialization time. During the execution of a service request, only the first stage is actually commanded.

**CLASS CONCRETE PipeManager EXTENDS OF**

```
ComponentManager {
    PUBLIC VOID init(ASOCIACION[] list) {
        stages[0] = PipeStage CREATE( list );
    }

    PRIVATE VOID commandStages (ANYTYPE datain, ComponentCollector res) {
        THREAD stages[0].request(datain,res);
    }
};
```

The objects of the PipeStage class create the following stage of the pipeline during its initialization phase. During the execution of their request() operation, an object stage directly commands the following one and only the last one sends the retransmittedsult to the Collector object, which is an instance of the class ComponentCollector, whose object reference is dynamically stage by stage.

**CLASS CONCRETE PipeStage EXTENDS OF**
ComponentStage
{
PUBLIC VOID init (ASOCIACION[ ] list) {
    stage.init(list);
    IF (! am_i_last) {
        otherstages[0] = PipeStage CREATE(
            TAIL(list));
    }
}
PRIVATE VOID commandOtherStages(
    ANYTYPE datain, ComponentCollector res) {
    THREAD otherstages[0].request(datain,res);
}
}

The execution() and request() operations are inherited from their respective superclasses and the private operations: commandStages() (from PipeManager class) and commandOtherStages() (from PipeStage class), together with the operation init() are redefined. There are, however, no synchronization problems since, in their definitions, the synchronization restrictions are inherited of their superclass.

6.2 The Cpan Farm

The technique of the parallel processing of the FARM as a High Level Parallel Composition or CPAN is shown here.

The so named Farm parallel pattern of interaction is made up of a set of independent processes, called worker processes, and a process that controls them, called the process controller by Roosta, Seller (1999), omitted4blind-review (2005). The worker processes are executed in parallel until all of them reach a common objective. The process controller is in charge of distributing the work and of controlling the progress of the farm until the solution of the problem is found. The Figure 8 shows the pattern of the Farm.

With this model it could be interesting to observe the parallel execution performance of several sorting algorithms that use the same set of data for all them.

6.2.1 Representation of the Farm as a CPAN

![Diagram of Cpan Farm]

The representation of parallel pattern FARM as a CPAN is shown in Figure 9.

As in the previous pattern, the objects Manager and stage-i are instances of the classes that inherit of the base-classes named ComponenManager and ComponentStage, respectively.

6.2.2 Semantic and Syntactic definition of the Cpan Farm

A first policy for the FARM composition is that the manager only waits for the first available result given by any of the stages, which corresponds to a service request carried out in an asynchronous way.

CLASS CONCRETE FarmManager EXTENDS OF ComponentManager {
    VAR INT nWorker;
    PUBLIC VOID init (ASOCIACION[ ] list) {
        asociacion[ ] newlist, INT i = 0;
        WHILE (! (newlist = TAIL(list))) {
            stages[i++] = FarmStage CREATE(
                CONS(HEAD(list),NULL));
            list = newlist;
        }
        nWorker = i;
    }
}

Figure 8  Farm with a controller and five workers

Figure 9  The Cpan of a Farm
PRIVATE VOID commandStages(ANYTYPE datain, ComponentCollector res)

VAR INT i;
FOR i =(0,nWorker)
{ THREAD stages[i].request(datain, res);
}

The concrete class FarmManager inherits of ComponentManager. The operation init() creates all the necessary stages, while the operation execution() is started in parallel, in an asynchronous way with other components, then it distributes data to all the stages, waiting for the first available result in the object collector. As in the case of the Pipeline, the synchronization restrictions are inherited from the abstract class ComponentManager without any problem. The stages of the farm are objects of FarmStage that inherits of ComponentStage:

CLASS CONCRETE FarmStage EXTENDS OF ComponentStage
{
};

The stages of the Farm are not connected each to the other. The manager commands all of them in his execution and the result of each one is sent to the collector object and made available to the Manager object. As the stages are executed in parallel according to an asynchronous way, the inherited scheduling policy guarantees that the collector access, which is necessary to return any results, will be synchronously carried out by these objects.

6.3 The Cpan TreeDV

Finally, the programming technique Divide and Conquer is presented as a CPAN here, which is indeed applicable to solve a wide range of problems that can be parallelized by following that algorithm scheme.

6.3.1 The Technique Divide and Conquer

Divide and Conquer design technique is characterized by the division of a problem into subproblems that have the same form that the initial complete problem Blloch, Guy, E. (1996), Kumar, Vipin; et al. (1994). The division of the problem in smaller subproblems must be carried out by using recursion, i.e., a recursive function that any subproblem (an the initial problem) calls to divide itself further. The recursive method continues dividing any caller subproblem until the division reach an atomic case that can not longer divide, then the partial results obtained from each sub-problem solution are combined to obtain the solution to the initial problem at the end Blloch, Guy, E. (1996), Kumar, Vipin; et al. (1994). Adopting this technique, problem division can be carried out in two subproblems at each step. Therefore, a parallel programming recursive formulation of that instance of Divide and Conquer method can be represented as a binary tree whose nodes are processors, processes or threads.

Figure 10  Binary Tree

The root node of the tree receives as input a complete problem that is divided into two parts. One part is sent to the left–son node, while the other is sent to the node that represents the right–son, see Figure 10. This division process progresses recursively until the lowest atomic level of nodes in the tree is reached. Lapsed a certain time, all the leaf–nodes receive as input a subproblem given by its father–node, then they solve it and the solutions (that are the exit of the node leaf) are again sent to their ancestors. Any father in the tree will obtain two partial solutions from its children and will combine them to provide only one solution that will be send to its own father node when it finishes. Finally the root node will deliver the complete solution of the problem on finishing, Brinch–Hansen (1994). While in a sequential implementation of Divide-and-Conquer, only one node of the tree can be processed or visited at the same time, within a parallel implementation more than one node can possibly be executed at the same time but at different levels of the D-and-C tree. When a subproblem is divided into two subproblems, both of them can be processed in a simultaneously.

6.3.2 Representation of the TreeDV as a CPAN

The representation of the tree–pattern that represents Divide and Conquer decomposition and the processing logic carried out by the nodes of that tree can be programmed as CPAN, which is depicted in Figure 11.

Contrary to the previous models, where the object–slaves were predetermined outside of the CPAN pattern, only one object–slave is statically predefined and associated to the first stage of the tree now if we use this model. The other object–slaves will be internally created by a tree–node in a dynamic way at each stage, because
the actual number of tree levels to obtain a solution depends on the problem to solve and not known a-priori nor the maximum depth of the tree.

6.3.3 Semantic and Syntactic definition of the Cpan TreeDV

The class TreeDVManager is created, which inherits of ComponentManager, and a binary-tree-shape communication pattern will implement the Divide and Conquer-based programming technique. commanded by the manager and each father-node internally creates its children-nodes.

The TreeDVManager class is created, which inherits from ComponentManager, and a binary tree communication pattern that implements the programming technique Divide and Conquer is created. The nodes of the binary tree are represented by the stages that are objects of the TreeDVStage class that inherits of ComponentStage one. Any instance of the class TreeDVManager only takes charge of the first stage or root-node of the tree during its initialization. During the execution of a service request, the root of the binary tree, that is to say, the first stage of the structure is commanded by the manager and internally each father-node created on the lowest levels of the tree will command its respective children to obtain the solution of the subproblem handed over.

CLASS CONCRETE TreeDVManager EXTENDS OF ComponentManager

{ PUBLIC VOID init (ASOCIACION[ ] list)

Any TreeDVStage object will take charge of creating a node of the binary tree (left or right). When the root-node or initial stage executes the request() operation in parallel, the problem will start calling with the method of the associated slave-object, then returning the division of the problem in two parts. Later on, it will call the method commandOtherStages() that hands over the subproblems, creates two associated stage-nodes, bounds the latter ones to its object-slaves that are dynamically created as the other nodes of the binary D-and-C get created too, and sends to each one a part of the problem to solve. The children stage-nodes will recursively receive the subproblem handed over by its ancestor and will execute their request() methods in parallel, and then carrying out the same process until subproblems can no longer be divided any more. The last TreeDVStage objects of the tree, that is to say, the leaves, send the result from their calculation process to their ancestor that combines these partial solutions in one which is sent now to its ancestor and so on. When the recursion finishes, the node root will send the final result when it arrives to a collector object, which in its turn will pass on the result to the manager of the composition.

CLASS CONCRETE TreeDVStage EXTENDS OF ComponentStage

{ VAR ASOCIACION list;

PRIVATE void executeStages(ANYTYPE datain, ComponentCollector res)

VAR ANYTYPE data_izq, data_der;

IF(datain.inicio<datain.fin)

{ otherStages[0]= TreeDVStage CREATE(list);

otherStages[1]=TreeStage CREATE(list);

stages[0] = TreeDVStage CREATE (list);

PRIVATE VOID commandStages(ANYTYPE datain, ComponentCollector res)

{ THREAD stages[0].request(datain,res);
 THREAD res.put(datain);
 }

private VOID commandOtherStages(ANYTYPE datain, ComponentCollector res)

{ IF(datain.inicio<datain.fin)

{ otherStages[0]= TreeDVStage CREATE(list);

otherStages[1]=TreeStage CREATE(list);

}
6.4 Use of a CPAN within an application

Once implemented the CPANs of interest, the way in which it is used within a user application is the following one:

1. The user asks the manager to begin a calculation by the corresponding method execution (\texttt{()}, defined in the CPAN.

   (a) To initialize the instance with the reference to the slave–objects that will be controlled by each stage and the name of the requested method as an association of pairs: (slave obj, associated method).

   (b) By using the operation \texttt{init()}, an association (slave obj, associated method) is handed over to each internal stage that is created by invoking the associated method on their slave object.

2. This execution is carried out as it continues:

   (a) The object collector is created in response to the request.

   (b) The input data (without verification of types) and the reference to the collector are handed over to each internal stage that is created by invoking the associated method on their slave object.

   (c) The results are obtained from the object collector.

   (d) The collector returns the results again to the CPAN environment, without carrying out any type checking.

3. An object manager has been created and initialized and some execution requests can be dispatched in parallel.

For more details see omitted4blind-review (2004).

An speedup–bases performance analysis of the Farm, PipeLine and TreeDV HLPCs appears in omitted4blind-review (2008).

6.4.1 CPANs utility of a real problem

DNA sequencing is the process of determining the precise order of nucleotides within a DNA molecule. It includes any method or technology that is used to determine the order of the four bases: adenine, guanine, cytosine, and thymine in a strand of DNA. Masoudi-Nejad A., Narimani Z. and Hosseinikhan N. (2013). The advent of rapid DNA sequencing methods has greatly accelerated biological and medical research and discovery. Knowledge of DNA sequences has become essential to carry out basic biological research, and in numerous applied fields such as medical diagnosis, biotechnology, forensic biology, virology and biological systematics. The rapid speed of sequencing attained with modern DNA sequencing technology has been instrumental in the sequencing of complete DNA sequences, or genomes of numerous types and species of life, including the human genome and other complete DNA sequences of many animal, plant, and microbial species, Pareek C., Smoczynski R. and Tretyn A. (2011).

In this regard use of CPANs for grouping DNA sequence fragments from the parallelization of a clustering algorithm to evaluate a set of fragments are made, which have a high probability of being aligned in an assembly task, DanishAli S. and Farooqui Z. (2013). The assembly of DNA strings is proposed as a combinatorial optimization problem and is classified as NP-hard and is based on the paradigm divide and conquer using a structure type farm, so that the computational cost of finding the sequence alignments and its splice is substantially reduced with respect to its sequential version. The number of processes required to process the fragments of DNA sequences of a specific genome such as that of a virus or bacteria is determined by the splice of the strings found by the sequential solution algorithm, which looks in parallel for overlaps in the remaining fragments. Two sub-strings of each fragment are taken for comparison with other fragments; and thus, particular splices are located and associated with the processes (a process for each splice sequence of DNA strings). A splice graph is then generated that shows the relationship between pairs of nodes (processes), as well as the lack of communication among others. The set of nodes (processes) of the graph that are inter–related are grouped together within a worker process pattern FARM. Each set of related nodes in the graph are independent and represent the grouping of fragments found. This is shown in 12. For more details see, Hernandez R., Olmos I. and Olvera A. (2016)

Similarly, the relationships between nodes within each worker FARM process can be represented as the patterns Pipeline. A double communication precisely represents the splices found in the DNA sequences. In 13 is shown the representation of graph–splices as a CPAN.

The new CPAN named CpanGraphADN is structured as a FARM of \texttt{n}–worker processes, i.e., \texttt{n}–fragments of DNA sequences and each worker process
is itself a two directions–communication pipeline CPAN formed by \( m \)-stages where each stage of CpanPipe represents a splice sequence of DNA strings connected with both, the previous stage as the next stage. The collector object receives the number of formed groups and the elements that belong to each of the formed groups. With the latter information collected, a in–depth search is performed to locate these items and obtain the sequence groups formed by the sequential algorithm assigned to each of the CPAN’s slave objects. With this result, the user can use an assembly of DNA sequences to try to complete a particular genome or to finish an incomplete sequence of DNA strings of some animal or plant type species.

### 6.4.2 experiment and results

An experiment was designed by using the CpanGraphADN with genomes of viruses and bacteria available on the web, see figure 14, whose data were obtained from European Nucleotide Archive, ENA www.ebi.ac.uk. Sequencer readings were simulated to create pseudo–random synthetic readings, so that the number of contigs can be formed. A contig is a continuous sequence of DNA that has been assembled by spliced DNA fragments, see Vera F., and Gonzalez B.. (2014), thus larger DNA fragments can be created.

<table>
<thead>
<tr>
<th>Genome</th>
<th>Length</th>
<th>Number of readings</th>
<th>Genoma coverage (average)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avalone herpesvirus victoria</td>
<td>211519</td>
<td>12691</td>
<td>0.9999</td>
</tr>
<tr>
<td>Adoxophyes orana granulovirus</td>
<td>99658</td>
<td>5979</td>
<td>0.9997</td>
</tr>
<tr>
<td>Adoxophyes orana nucleopolyhedra</td>
<td>111725</td>
<td>6703</td>
<td>0.9996</td>
</tr>
<tr>
<td>African swine fever virus benin</td>
<td>182285</td>
<td>10937</td>
<td>0.9999</td>
</tr>
<tr>
<td>Feline coronavirus</td>
<td>29215</td>
<td>1752</td>
<td>0.9998</td>
</tr>
<tr>
<td>Shrimp white spot syndrome</td>
<td>307288</td>
<td>18437</td>
<td>0.9999</td>
</tr>
<tr>
<td>Trichoplosia ni ascomycetes 2c</td>
<td>174060</td>
<td>10443</td>
<td>0.9999</td>
</tr>
<tr>
<td>Vaccinia virus GLU-1 h68</td>
<td>203058</td>
<td>12183</td>
<td>0.9998</td>
</tr>
</tbody>
</table>

### Figure 14  genome used by the CpanGraphADN for Experimentation

The CPAN used a fragmented genome, and the resulting fragments were pooled ensuring the same number of groupings and the follower contigs results were obtained, see figure 15. However, it should be noticed that the same number of genomes groups on the contigs was not always obtained. Thus, in the last three contigs that read the genome, only 4 groups were obtained by grouping the simulated fragments readings in CPAN. Similarly, for the second genome only two groups were obtained and only one contig was read. For the remaining genomes, the number of expected readings searches (of different lengths) by the CPAN was obtained.

### Figure 15  DNA sequence groups found by the CPAN

Finally, the execution time of CpanGraphADN is shown in figure 16. The plot shows the number of processes deployed for the calculation of eight genomes in an experiment conducted on a computer with Intel Core i8 processor, and using a video accelerator card with 1664 CUDA cores and clock frequency 1178 Mhz.

### 7 Performance

Some CPANs adapt better to the communication structure of a given algorithm than others, therefore...
Design and implementation of communication patterns using Parallel Objects

1. It is necessary to create an instance of the adequate class manager, that is to say, a specialized instance (this involves the use of inheritance and generic instances) by implementing the required parallel behavior of the final manager object. This is performed by following steps:

   (a) Instance’s initialization from the class manager, including the information, given as associations of pairs (slave-obj, associated-method); the first element is a reference to the slave object being controlled by each stage and the second one is the name of its callable method.

   (b) The internal stages are created (by using the operation init()) and, for each one of these, the association (slave-obj, associated-method) is passed to. The second element is needed to call the associated-method on the slave object.

2. The user asks the manager to start a calculation by invoking the execution() method of a given CPAN. This execution is carried out as it follows:

   (a) a collector object is created to fulfill this petition;

   (b) input data are passed to the stages (without any type verification), and one reference to the collector;

   (c) results are obtained from the object collector;

   (d) the collector returns the results to the application environment without performing any type of verification.

3. An object manager will have been created and initialized and some execution petitions can then start to be dispatched in parallel.

To evaluate the performance of CpanFarm, CpanPipe and CpanTreeDV, a parallel sorting algorithm was programmed on a parallel computer with 64 processors, 8 GB of RAM memory, high-speed buses and distributed shared memory architecture. Performance measures obtained with CpanFarmQS, CpanPipeQS, CpanTreeDVQS that solve a sorting problem by using Farm, Pipeline and Binary Tree CPANs respectively, were carried out with the following execution conditions:

- In the case of CPAN Farm and CPAN Pipe, a list–based sequential sorting algorithm was parallelized; whereas in the case of CpanTreeDv, a sorting algorithm based on a Divide-and-Conquer binary tree was designed.

- CPAN Farm, CPAN Pipe and CPAN TreeDV implemented the same sequential algorithm, for result–values comparison, in each of the slave objects associated with the stages of CPANs.

- A set of integer numbers, randomly obtained from 0 to 50000, were sorted. Each processor experienced a sufficient load in order to observe any performance improvement by deploying CpanFarm, CpanPipe and CpanTreeDV in the proposed parallelizations.

- The execution of CpanFarm, CpanPipe and CpanTreeDV–based parallel programs used 2, 4, 8, 16 and 32 processors on a full-time basis.

Assuming that we want to apply Quicksort to a given array of data, we can observe that some CPANs adapt better to the communication structure between processes in charge of solving subproblems than others. Different parallel implementations of the same sequential algorithm will therefore yield different speedups due to level of matching between connection topology of processors and subproblem decomposition pattern dynamically applied to the sample data when the program runs (see figures 17, 18, 19). The program is structured in six sets of classes, each one gets instantiated from a CPAN in the High Level...
Parallel Compositions library, which represents the implementation of the parallel patterns named Farm, Pipe and TreeDV.

8 Case study: the branch & bound technique as a CPAN

Branch-and-bound (BB) makes a partition of the solution space of a given optimization problem. The entire space is represented by the corresponding BB expansion tree, whose root is associated to the problem, initially unsolved. At each node, its children nodes represent the subspaces obtained through branching rules, i.e., progressively subdividing the solution space initially represented by the parent node. Leaves of the BB tree represent nodes that cannot be subdivided any further, thus providing a final value of the cost function associated to a possible solution of the problem.

Three stages are performed during the execution of a program based on a BB algorithm:

1. Selection: A node belonging to the set of live nodes, i.e. those not pruned yet, is extracted. Node selection depends on the strategy search on the live node list, which was previously decided for its use in the algorithm.

2. Branch: the node selected in the previous step is subdivided in its children nodes by following a ramification scheme to form the expansion tree. Each child node receives from its father node enough information to enable it to search one suboptimal solution.

3. Bound: Some of the nodes created in the previous stage are deleted, i.e., those whose associated partial cost, which is given by the cost function associated to this BB algorithm instance, is greater than the best minimum bound calculated up to that point.

Branching is generally separated from bounding of nodes while the expansion tree is built in parallel BB implementations, and therefore we followed a Farm communication scheme that makes these two activities as separated implementation realms omitted\textsuperscript{4blind-review} (2006). For a given instance of the BB algorithm, the expansion tree is obtained by iteratively subdividing the intermediate stage objects according to a pattern until a stage representing a leaf-node of the expansion tree is found, see figure 20. Pruning is implicitly carried out within another farm construction by using
a totally connected scheme between all the processes. The manager can therefore communicate a sub-optimal bound found by a process to the rest of the branching processes and thus avoid unnecessary ramifications of sub-problems. The Cpan Branch & Bound is composed of a set of Farm Cpans; see figure 20. Each one represents a set of worker processes and one manager, therefore, forming a new type of structured Farm, the Farm Branch & Bound or FarmBB, which is also included in the library of CPANs. All the worker processes of the Farm BB are executed in parallel, thereby forming the expansion tree of nodes given by the BB algorithm technique. The initial problem, or the root of the expansion tree, is given to the manager process of the initial Cpan Farm, which is in charge of distributing the work and of controlling the global calculation progress. The manager is also responsible for sending results to the collector of the Cpan FarmBB which, in its turn, displays them (2006).

The CPAN–based parallel BB algorithm was tested by solving the Traveling Salesman Problem - TSP with 50 cities and by using the first best search strategy driven by a least cost function associated to each live node. The results obtained yielded a deviation ranging from 2% (2 processors) to 16% (32 processors) with respect to the optimal ones, as predicted by the Amdalh law for this parallelized algorithm; more details can be found in (2006)). This CpanBB speedup analysis is shown in figure 21.

![Figure 21 Speedup of parallel CpanBB with 50 cities in 2, 4, 8, 16 and 32 processors](image)

9 Present programming work

The programming work carried out until now consists of a library of six sets of classes that completely implement the following parallel patterns: Farm, Pipe and TreeDV as CPANs, which are the core of a more extensive library and framework of High Level Parallel Compositions. The sets of classes in its present development state are the following ones:

1. The set of the base-classes, which are necessary to build any CPAN, i.e., the classes that implement CPAN Parallel Objects.

   - **Class Object:** we have called object-classes according to this common superclass. It represents generic slave-objects, i.e., it is an abstract class inherited by concrete slave-objects (for instance: ISort, QSort, Invierte, etc.) necessary to implement the virtual method “resuelve()”, which is the method to be executed by a slave-object within the CPAN composition.

   - **Class FutureType:** It defines the type FutureType referred as a Future of type void * that will be the type of the value returned by a specific function. This class denotes instances of FutureType to mean that if the return value is not currently available, it can be at any future instant.

   - **Header Util.h:** It contains the definition of “primitive function” used in the code of the CPAN (the primitive CONS needed to append elements on a list, the primitive HEAD that obtains the head of the list, the primitive TAIL that obtains the rest of a list and the primitive EVAL that evaluates a function), as well as the definition of several types of abstract data, i.e., the association type that defines a pair (method,slave obj), the type of the method that defines the execution method associated to a slave-object, the vector type “sol” that is used as the result container. This header file is necessary for the implementation of CPANs.

   - **ComponentCollector:** this class is used to create instances of Collector objects. An collector–object will set the solutions from the connected stages to it and will store them in a variable “sol” of type vector.

   - **Sched GetPut:** Thus class defines the synchronization constraints for the member functions get() and put(), within the ComponentCollector class. Mutex for put() bound processes, Mutex for get() processes and Sync for the communication of processes put() and get(). All the members of this class are static.

   - **ComponentStage:** This class defines the generic structure of a STAGE in the construction of a CPAN and has to be inherited by the specific stages that can be built in the implementation of the parallel composition to instance a particular stage. It is made up of two parts: the part of initialization of the “stage” and the execution part, in parallel.

   - **ComponentManager:** This class defines the generic structure of a MANAGER in the construction of a CPAN that has to be inherited by the specific manager that can be built for the implementation of the CPAN. It is made up of two parts: the part of the “manager’s” initialization and the execution part, in parallel.

   - **Sched RequestExecution:** This class implements the constraints connected with maximum parallelism and use of futures for the
The set of classes that define the Cpan Farm that is formed of:

Main Program cpanfarm.cpp. It proves the execution of the Cpan Farm. The problem that is solved is the sorting in parallel of two arrays of integer numbers using three different algorithms of sorting (QuickSort, BubbleSort and Isort) sent concurrently.

Class FarmStage. It defines the component stage relative to the parallel pattern FARM and it inherits of ComponentStage.

Class FarmManager. It defines a concrete instance of a manager for a CPAN type Farm. The class inherits of ComponentManager. Class CpanFarm. It is used in the main program to create an active object that represents The High Level Parallel Composition Farm and to solve “n” problems in parallel through this implemented communication pattern.

5. The set of classes that define the CPAN Pipe, formed by the following classes:

Main Program cpanpipe.cpp. It proves the execution of the Cpan Pipe. The problem that is solved is the execution of a sequence in parallel of 3 algorithms: the algorithm Invierte, the algorithm Qsort and again the algorithm Invierte to invest a sequence of data in disorder, to order the sequence change and to invest the sorted sequence of two arrays of integer numbers sent concurrently again.

Class PipeStage. It defines the component stage relative to the parallel pattern PIPE and it inherits of ComponentStage.

Class PipeManager. It defines a concrete instance of a manager for a CPAN type Pipe. The class inherits of ComponentManager.

Class CpanPipe. It is used in the main program to create an active object that represents The High Level Parallel Composition Pipe and to solve “n” problems in parallel through this implemented communication pattern.

6. The set of classes that define the Cpan TreeDV, formed by the classes Main Program cpantreeDV.cpp. It proves the execution of the Cpan TreeDV. The problem that is solved is the sorting in parallel of two arrays of integer numbers using the QuickSort algorithm so many times as nodes stage of the tree leave creating in the solution process which are sent concurrently for its execution.

Class TreeDVStage. It defines the component stage relative to the parallel pattern TreeDV and it inherits of ComponentStage.

Class TreeDVManager. It defines a concrete instance of a manager for a CPAN type TreeDV. The class inherits of ComponentManager.

Class CpanTreeDV. It is used in the main program to create an active object that represents The High Level Parallel Composition TreeDV and to solve “n” problems in parallel through this implemented communication pattern.
10 Conclusions

A programming method has been presented, which is based on the High Level Parallel Compositions or CPANS adapted to be used with C++ programming language and POSIX standard for thread programming. We discuss the implementation of CPANS Pipe, Farm and TreeDV as generic and reusable patterns of communication/interaction between processes, which can even be used by inexperienced parallel application programmers to obtain efficient code by only programming the sequential parts of their applications. Selected study cases, consisting of DNA sequencing and TSP have been included to show speedup and low execution times w.r.t. best sequential version of the algorithms that solve these problems. We have also obtained good performance in their executions and speedup scalability (compared to Amdahl’s law) on the number of processors used to obtain the solution of several other complex problems. Among these, it is worth to mention parallelization of the B&B technique to offer an optimal solution of the TSP (NP-Complete problem), as well as other programming solutions to sorting, searching and optimization problems.

References


Kumar, Vipin; et al; (1994) Introduction to parallel computing. Design and Analysis of Algorithms. Benjamin-Cummings, USA.


OMITTED FOR BLIND REVIEW (2008). A Parallel Programming Methodology using Communication Patterns named CPANS or Composition of Parallel Object. 20TH European Modeling and Simulation. Italy.


