APPROACH CLASS LIBRARY OF HIGH LEVEL PARALLEL COMPOSITIONS TO IMPLEMENTS COMMUNICATION PATTERNS USING STRUCTURED PARALLEL PROGRAMMING

M. Rossainz-López[(a)], M. I. Capel-Tuñón[(b)]

[(a)] Universidad Autónoma de Puebla, Avenida. San Claudio y 14 Sur, San Manuel, Puebla, State of Puebla, 72000, México
[(b)] Departamento de Lenguajes y Sistemas Informáticos, ETS Ingeniería Informática y de Telecomunicación, Universidad de Granada, Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain

[(a)]rossainz@cs.buap.mx, [(b)]manuelcapel@ugr.es

ABSTRACT
This article presents through an environment of Parallel Objects, an approach to Structured Parallel Programming and the Object-Orientation paradigm, a programming methodology based on High Level Parallel Compositions (HLPC). By means of the method application, the parallelization of commonly used communication patterns among processes is presented, which is initially constituted by the HLPCs Farm, Pipe and TreeDV that represent, respectively, the patterns of communication Farm, Pipeline and Binary Tree, the latter one used within a parallel version of the design technique known as Divide and Conquer.

Keywords: Parallel Objects, Communication Patterns, Structured Parallel Programming, High Performance Computing.

1. INTRODUCTION

At the moment the construction of concurrent and parallel systems has less conditioning than one decade ago, since there currently exist, within the realms of HPC or Grid computing, high performance parallel computation systems that are becoming more and more affordable, being therefore possible to obtain a great efficiency today in parallel computing without having to invest a huge amount of money in purchasing a state-of-the-art multiprocessor. Nevertheless, to obtain efficiency in parallel programs is not only a problem of acquiring processor speed; on the contrary, it is rather about how to program efficient interaction/communication patterns among the processes (Brinch Hansen 1993; Capel and Palma 1992; Darlington 1993), which will allow us to achieve the maximum possible speed-up of a given parallel application. These patterns are aimed at encapsulating parallel code within programs, so that an inexperienced parallel applications programmer can produce efficient code by only programming the sequential parts of the applications processes (Brinch Hansen 1993; Brinch Hansen 1994; Capel and Troya 1994). Parallel Programming based on the use of communication patterns is known as Structured Parallel Programming (SPP) (Corradi and Zambonelli 1995; Danelutto and Orlando 1995). The widespread adoption of SPP methods by programmers and system analysts currently presents a series of open problems, which motivate us to carry out research in this area; we are particularly interested on those that have to do with parallel applications that use predetermined communication patterns among their processes. In regarding an improvement of the use of SPP methods, several open problems have currently been identified. It is worth mentioning, among the most important, the following ones:

- The lack of acceptance of structured parallel environments applicable to developing a wider range of software applications.
- Determination of a complete set of communication patterns as well as their concrete semantics.
- The necessity to make available predefined communication patterns or high level parallel compositions aimed to encapsulate parallel code within programs.
- Adoption without anomalies of a programming approach merging concurrent primitives and Object-Oriented (O-O) features, thereby fulfilling the requirements of uniformity, genericity and reusability of software components (Corradi and Zambonelli 1995).

This work study the Methods of Structured Parallel Programming, proposing a new implementation of a library classes of High Level Parallel Compositions-HLPC or CPAN according to its Spanish acronym (Corradi and Zambonelli 1995; Danelutto and Orlando 1995; Capel and Rossainz 2004; Rossainz 2005), which provide the programmer with the communication patterns more commonly used in Parallel Programming. At the moment, the library includes the following ones:

Farm, Pipeline and Binary Tree (Rossainz and Capel 2008), the latter one being used in a parallel version of Divide and Conquer algorithmic design technique and implementation of the solution of the traveling salesman problem using the HPLC Farm using branch & bound algorithmic design technique for solving NP-complete problems (Rossainz and Capel 2006).

2. HIGH LEVEL PARALLEL COMPOSITIONS (HLPC)

The basic idea of the programming method consists in the implementation of any type of communication patterns between parallel processes of an application or distributed/parallel algorithm as classes, following the O-O paradigm. Starting from these classes, an object can be instantiated and its methods can be invoked, and then be executed, on a client-server basis, thereby hiding parallelism or communication protocols to the application processes. A HLPC is made up of the composition of a set of objects of three types, (see Figure 1).

An object manager that represents the HLPC itself, making of it an encapsulated abstraction that hides its internal structure. The manager controls the references of a set of objects (a denominated Collector object and several denominated Stage objects) that represent the components of the HLPC and whose execution is carried out in parallel and should be coordinated by the manager itself.

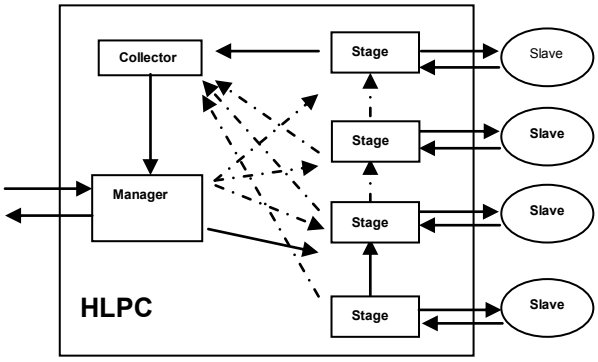

Figure 1: Internal Structure of a HLPC

The stage objects are objects of specific purpose responsible for encapsulating a client-server type interface between the manager and the object slaves (objects that are not actively participative in the composition of the HLPC, but rather, are considered external entities that contain the sequential algorithm constituting the solution of a given problem) as well as providing the necessary connection among them to implement the communication pattern semantics, whose definition is being sought. In other words, each stage should act in parallel as a node of the graph that represents the communication pattern and should be capable of executing its methods as an active object. A stage can be directly connected to the manager and/or to another component stage depending on the particular pattern of the HLPC.

An object collector which is an object in charge of storing in parallel the results that it receives from the stage objects connected to it, i.e., during the service of a petition. The control flow within the stages of a HLPC depends on the communication pattern implemented between these. When the HLPC concludes its execution, the result does not return to the manager directly, but rather to an instance of the class Collector, which takes charge of storing these results and of sending them to the manager, which then sends them to the exterior as they arrive, i.e., without being necessary to wait for all the results to be obtained at the end of the computation.

2.1. The HLPC as composition of parallel objects

The objects manager, collector and stages are included within the definition of Parallel Object - PO (Corradi and Zambonelli 1995). The Parallel Objects are active objects, that is to say, objects that have execution capacity in them. The applications within the pattern PO can exploit the parallelism so much among objects (inter-object) as the internal parallelism of them (intra-object). An object PO has a similar structure to that of an object in Smalltalk, but it also includes a politics of scheduling, determined a priori that specifies the form of synchronizing an or more operations of a class in parallel (Rossainz and Capel 2012). The synchronization policies are expressed in terms of restrictions; for example, the mutual exclusion in processes readers/writers or the maximum parallelism in processes writers. All the parallel objects derive then of the classic definition of "class" more the incorporation of the synchronization restrictions (mutual exclusion and maximum parallelism). The objects of oneself class shares the same specification contained in the class of which you/they are instantiates. The inheritance allows deriving a new specification of one that already exists. The parallel objects support multiple inheritance.

2.2. Types of communication between parallel objects

1. The synchronous way stops the client's activity until the object's active server gives back the answer to the petition.
2. The asynchronous way does not force any waiting in the client's activity; the client simply sends its petition to the active server and then it continues.
3. The asynchronous future way makes only to wait the client's activity when the result of the invoked method is needed to evaluate an expression during its code execution.

2.3. Semantic and Syntactic definition of the classes bases of a HLPC anyone

The abstract class ComponentManager defines the generic structure of the component manager of a HLPC, from which all the concrete manager classes are derived,

depending on the parallel behaviour which is needed to create a specific HLPC.

```
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 abstract class ComponentStage defines the generic structure of the component stage of a HLPC as well as its interconnections, so that all the concrete stages needed to provide a HLPC with a given parallel behaviour can be obtained by class instantiation.

```
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);
 };
```

The concrete class ComponentCollector defines the concrete structure of the component collector of any HLPC. It implements a multi-item buffer, which permits the storage of the results from stages that make reference to this collector.

```
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);
  };
```

2.4. The synchronization restrictions MaxPar, Mutex and Sync

Synchronization mechanisms are needed when several petitions of service take place in parallel in a HLPC, being capable its constituting parallel objects of interleaving their concurrent executions while, and at the same time, they preserve the consistency of the data being processed. Within the code of any HLPC, execution constraints are automatically included when the reserved words MAXPAR, MUTEX and SYNC of the library are found. The latter ones must be used to obtain a correct programming of object methods and to guarantee data consistency in applications.

3. THE HLPC PIPE, FARM AND TREEDV

The parallel patterns worked until now have been the *pipeline*, the *farm* and the *TreeDVD*, since they constitute a significant set of reusable communication patterns in multiple parallel applications and algorithms.

3.1. The HLPC PipeLine

It is presented the technique of the parallel processing of the pipeline as a High Level Parallel Composition or HLPC, applicable to a wide range of problems that you/they are partially sequential in their nature. The HLPC Pipe guarantees the parallelization of sequential code using the patron PipeLine.

3.1.1. The technique of the Pipeline

Using the technique of the Pipeline, the idea is to divide the problem in a series of tasks that have to be completed, one after another. In a pipeline each task can be executed by a process, thread or processor for separate (Roosta 1999). The processes of the pipeline are sometimes called stages of the pipeline. Each stage can contribute to the solution of the total problem and it can pass the information that is necessary to the following stage of the pipeline. This type of parallelism is seen many times as a form of functional decomposition. The problem is divided in separate functions that can be executed individually, but with this technique, the functions are executed in succession.

3.1.2. Representation of the Pipeline as a HLPC

The figure 2, represent the parallel pattern of communication Pipeline as a HLPC.

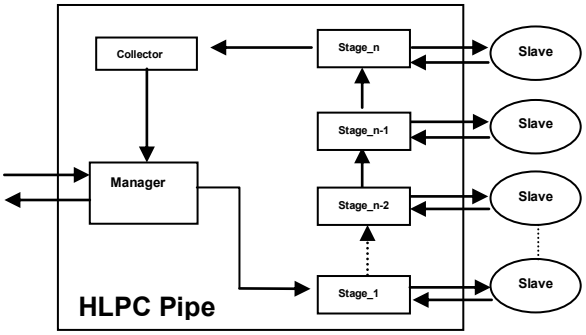

Figure 2: The HLPC of a Pipeline

The objects *stage_i* and *Manager* of the graphic pattern of the HLPC-Pipe are instances of concrete classes that inherit the characteristics of the classes ComponentManager and ComponentStage.

3.1.3. Semantic and Syntactic Definition of the HLPC-Pipe

The HLPC-Pipe is represented by the class PipeManager that inherits of ComponentManager, and a pattern of communication pipeline implements whose stages is instances of the class PipeStage that inherits of ComponentStage. Any object PipeManager only takes charge of the first stage of the pipeline in its initialization. During the execution of a petition of service, the first stage is only 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 class PipeStage creates the following stage of the pipeline during its initialization phase. In the execution of their operation request(), an object stage commands directly to the following one and it is the last one that sends the result to the object Collector (instance of the class ComponentCollector) whose reference is transmitted 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 operations *execution()* and *request()* are inherited of their respective super class and the operations private *commandStages()* (of the class PipeManager) and *commandOtherStages()* (of the class PipeStage), together with the operation *init()* are redefined. However there are not synchronization problems since in their definitions the synchronization restrictions they are inherited of their super class.

3.2. The HLPC Farm

It is shown the technique of the parallel processing of the FARM as a High Level Parallel Composition or HLPC.

3.2.1. The technique of the Farm

The parallel pattern of interaction Farm, is formed to each other with a set of independent processes, called worker processes, and a process that controls them, call the process controller (Roosta 1999). The worker processes are executed in parallel until reaching a common objective for all them. The process controller is in charge of distributing the work and of controlling the progress of the farm until finding the solution of the problem. With this model it could be interesting to observe the performance of the execution in parallel of several sorting algorithms with oneself set of data for all them.

3.2.2. Representation of the Farm as a HLPC

The representation of parallel pattern FARM as a HLPC is show in figure 3.

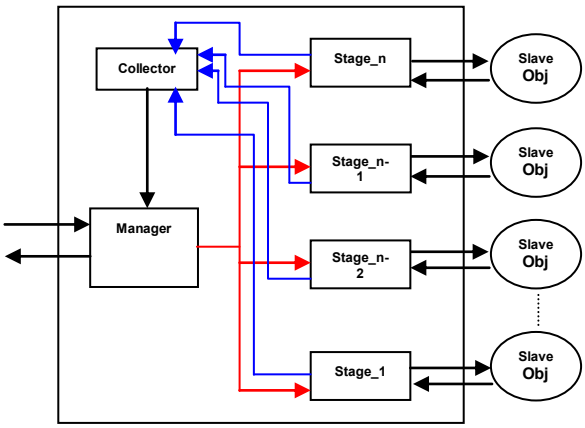

Figure 3: The HLPC of a Farm

The same as in the previous pattern, the objects Manager and *stage_i* are respectively instances of the classes that inherit of the classes base denominated ComponenManager and ComponentStage.

3.2.3. Semantic and Syntactic Definition of the HLPC Farm

A first policy for the composition of the FARM is that the manager only waits the first available result given by anyone of the stages, which respond to a petition of service in an asynchronous way.

```
CLASS CONCRETE FarmManager EXTENDS OF
ComponentManager
  {
  VAR
    INT nWorker;

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

  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 ()* create all the necessary stages, while the operation execution () it is thrown in parallel in an asynchronous way, distributing data to all the stages, waiting the first available result in the object collector. As in the case of the pipeline, the synchronization restrictions are inherited of 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 some with others (*am_i_last* is always *true* and the *CommandOtherStages* operation, as in the abstract class this empty). The manager commands them to all in his execution and the result of each one is a correspondent to the object collector and position to the manager's disposition. As the stages are executed in parallel in an asynchronous way, the policies of scheduling inherited of the super class guarantees that the access to the collector, necessary to return the results, it will be made in a way synchronized on the part of this objects.

3.3. The HLPC TreeDV

Finally, the programming technique is presented it Divide and Conquer as a HLPC, applicable to a wide range of problems that can be parallelizable within this scheme.

3.3.1. The technique of the Divide and Conquer

The technique of it Divide and Conquer it is characterized by the division of a problem in sub-problems that have the same form that the complete problem. The division of the problem in smaller sub-problems is carried out using the recursion. The method recursive continues dividing the problem until the parts divided can no longer follow dividing itself, then they combine the partial results of each sub-problem to obtain at the end the solution to the initial problem (Blelloch 1996). In this technique the division of the problem is always made in two parts, therefore a formulation recursive of the method Divide and Conquer form a binary tree whose nodes will be processors, processes or threads. The node root of the tree receives as input a complete problem that is divided in two parts. It is sent to the node left son, while the other is sent to the node that represents the right son.

This division process is repeated of recursive form until the lowest levels in the tree. Lapsed a certain time, all the nodes leaf receives as input a problem given by its node father; they solve it and the solutions (that are the exit of the node leaf) are again correspondents to its progenitor. Any node father in the tree will obtain his children's two partial solutions and it will combine them to provide an only solution that will be the node father's exit. Finally the node root will give as exit the complete solution of the problem, (Brinch Ansen 1993). This

way, while in a sequential implementation a single node of the tree can be executed or visited at the same time, in a parallel implementation, more than a node it can be executed at the same time in the different levels, it is, when dividing the problem in two sub-problems, both can be processed in a simultaneous way.

3.3.2. Representation of the TreeDV as a HLPC

The representation of the patron tree that defines the technique of it Divide and Conquer as HLPC has their model represented in figure 4.

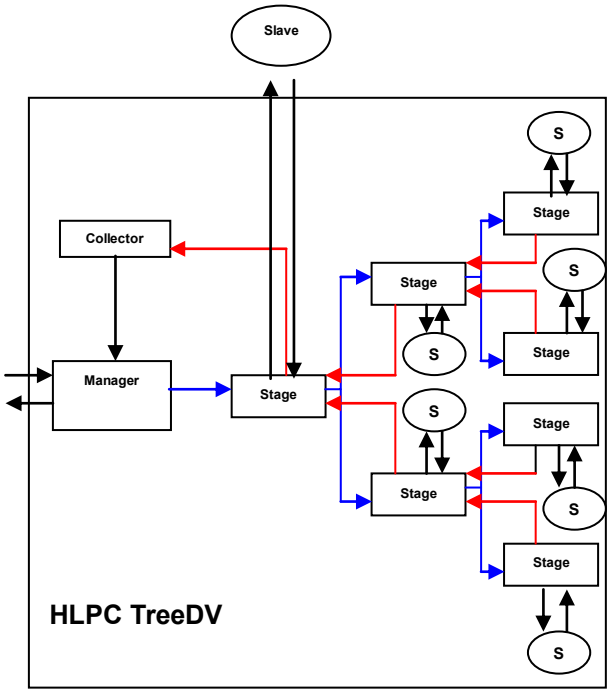

Figure 4: The HLPC of a TreeDV

Contrary to the previous models, where the objects slaves were predetermined outside of the pattern HLPC, in this model an object slave is only predefined statically and associated to the first stage of the tree.

The following objects slaves will be created internally by the own stages in a dynamic way, because the levels of the tree depend from the problem to solve and a priori the number of nodes that can have the tree is not known, neither its level of depth.

3.3.3. Semantic and Syntactic Definition of the HLPC TreeDV

The class TreeDVManager is created, which inherits of ComponentManager, and a pattern of communication of a binary tree implements within the programming technique it Divide and Conquer. The nodes of the binary tree are represented by the stages that are objects of the class TreeDVStage that inherits of ComponentStage. Any instance of the class TreeDVManager only takes charge of the first stage or node root of the tree in its initialization. During the execution of a petition of service, the root of the binary tree, that is to say, the first stage of the structure is commanded by the manager and internally each node father created in the lowest levels in the tree will command his respective nodes children in the solution of the problem.

```
CLASS CONCRETE TreeDVManager EXTENDS OF
ComponentManager
   {
    PUBLIC VOID init (ASOCIACION[] list)
     { stages[0] = TreeDVStage CREATE (list); }

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

Any object TreeDVStage will take charge of creating a node of the binary tree (left or right). When the node root or initial stage execute the operation *request()* in parallel, the problem is evaluated with the method of the slave object associate, returning the division of the problem in two parts. Later on, will call himself to the method *commandOtherStages()* who will take this sub-problems, it will create two nodes stages associated to their node father, associating these last their respective objects slaves that will be created dynamically conform to they go creating the nodes of the binary tree, and will send to each one a part of the problem to solve. Of recursive form the nodes stages children will receive the sub-problem and they will execute their method *request()* in parallel, carrying out the same process until the sub-problem can no longer be divided more. The last objects TreeDVStage of the tree, that is to say, the leaves, send the result from their calculation process to their progenitor and this combines the solutions to pass them, in turn, to their progenitor and so on, in the return of the recursion, until arriving to the node root who will send the final result to an object collector who in turn will pass the result to the manager of the composition.

```
CLASS CONCRETE TreeDVStage EXTENDS OF
ComponentStage
 {
  VAR
    ASOCIACION list;

  PUBLIC VOID init(ASOCIACION[] *list)
  {
    this.list=list;
    stage.init(list);
    am_i_last= FALSE;
  }

  PRIVATE VOID commandOtherStages
    (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);

   THREAD otherStages[0].request
     (dv(datain,datain.inicio, datain.medio),res);

   THREAD otherStages[1].request
     (dv (datain,datain.medio+1, datain.tam-1),res);
  }
}

PRIVATE ANYTYPE dv(ANYTYPE dataout,
                   int inicio, int fin)
  {
   datain.inicio=inicio;
   datain.fin=fin;
   RETURN datain;
  }
};
```

4. USE OF A HLPC WITHIN AN APPLICATION

Once implemented the HLPCs of interest, the way in that you/they are used in user's application is the following one:

1. It will be necessary to create an instance of the class manager of interest, that is to say, one that implements the required parallel behavior in agreement with the following steps:
   1.1. To initialize the instance with the reference to the objects slaves that will be controlled by each stage and the name of the method requested as an association of even (slave_obj, associated_method).
   1.2. The internal stages is created (using the operation *init()*) and they are passed each one an association (slave_obj, associated_method) that will use invoking the associated_method on their slave object.
2. The user asks the manager to begin a calculation through the execution within the HLPC of the method execution(). This execution is carried out as it continues:
   2.1. The object collector is created with respect to the petition.
   2.2. They are passed to the stages the input data (without verification of types) and the reference to the collector.
   2.3. The results are obtained from the object collector.
   2.4. The collector returns the results again to the exterior without verification of types.
3. An object manager has been created and initialized and some execution petitions can be dispatched in parallel.

The speedup analysis of the Farm, PipeLine and TreeDV HLPCs appears in (Rossainz and Capel 2008)

4.1. Example of the use of a HLPC

Supposing that we want process an array that contains data that must be ordered. It is well-known that, depending on the data, different sorting algorithms can show different performances. It can be interesting to order the same data by means of the execution of different algorithms in parallel and to wait the results. The parallel pattern identified is FarmManager.

1. Objects of different sorting classes are created that represent the slaves objects and are stored in an array of references to object:
   ```
   Object obj[]= {
                  Qsort CREATE(. . .),
                  BubleSort CREATE( . . .),
                 Isort CREATE(. . . )
                };
   ```
2. The objects are created that they represent the methods associated to the objects slaves and they are stored in an array of references to method.
   ```
   method meth[]={
                  method CREATE (. . . resuelve),
                  method CREATE (. . . resuelve),
                 method CREATE (. . . resuelve)
                };
   ```
3. A list of associations is created (slave_obj, associated_method)
   ```
   Asociación pareja=
   crea_asociacion(obj,meth,3);
   ```
4. An instance of the class FarmManager is created initializing it with the previous list
   ```
   FarmManager cpanfarmInt=
   FarmManager CREATE(pareja);
   ```
5. The initial data are specified to process by means of the specific creation of the type and of the defined data as an object for the user, in this example this object it is denominated MyInt. Each object MyInt represents a problem to solve
   ```
   int nums[t1]={-11,-14,-6,-1,...};
   int nums2[t2]={5,1,9,11,4,8,...};
   ANYTYPE data[]= {
                    MyInt CREATE(t1,nums),
                    MyInt CREATE(t2,nums2)
                   };
   ```
6. The initial data are printed in screen
   ```
   FOR i =(0,num_problems)
       {imprime_datos(data[i]);}
   ```
7. The instance FarmManager is ready to work, that is to say, the execution of the operation *execution()* is requested that sort the array of data in parallel:

```
FUTURETYPE resul[num_problems];
FOR i =(0,num_problems)
 {
  resul[i]=THREAD
  cpanfarmInt.execution(data[i]);
 }
```

8. The final results are printed in screen

```
ANYTYPE resultados[num_problems];
FOR i =(0,num_problems)
 {
  resultados[i]=resul[i];
  imprime_datos(resultados[i]);
 }
```

5. THE WORK OF PRESENT PROGRAMMING

The work of programming made until this moment is made up of six sets of classes that constitute the implementation of the parallel patterns farm, pipe and treeDV as HLPCs that are part of what will be the library of High Level Parallel Compositions. These sets of classes are the following ones:

The set of the classes base (Table.1), necessary to build a HLPC, or said in another way, the classes that implement the Parallel Objects of a HLPC.

Table 1: Classes base for the construction of a HLPC

Class Object	It represents slave objects generic, that is to say, it is an abstract class inherited by the slave objects concrete (for example ISort, QSort, Invierte, etc.) to implement the virtual method it *resuelve()* that will be the method to be executed by the slave object within the HLPC.
Class FutureType	It defines the type FutureType referred as a Future of type void * that will be the type of the value of return of a function. This class denotes instances of FutureType to indicate that but this available one the return value in a given moment, if it can be he in some time in the future.
Header Util.h	It contains the definition of "primitive function" used in the code of the HLPC (the primitive CONS for to append elements in a list, the primitive HEAD that obtains the head of a list, the primitive TAIL that obtains the rest of a list, the primitive EVAL that evaluates a function), as well as the definition of several types of abstract data (the type association that defines a pair (method,slave_obj), the type *method* that defines the execution method associated to an slave object, the type vector_sol that is used as container of results) necessary in the implementation of the HLPCs.
Class ComponentCollector	It is used to create instances or objects Collector. An object collector will to set the solutions from the connected stages to him and it will store them in a variable of type vector_sol.
Class Sched_GetPut	It defines the synchronization restrictions for the functions member *get()* and *put()* within the class ComponentCollector: Mutex for the processes *put()*, Mutex for the processes *get()* and Sync for the communication of processes *put()* and *get()*. All the members of this class are static.
Class ComponentStage	It defines the generic structure of a STAGE in the construction of a HLPC and has to be inherited by the specific stages that can be built in the implementation of the parallel composition for to concretize a particular stage. It is constituted of two parts: the part of initialization of the "stage" and the execution part, in parallel.
Class ComponentManager	It defines the generic structure of a MANAGER in the construction of a HLPC that has to be inherited by the specific manager that can be built for the implementation of the HLPC. It is constituted of two parts: the part of the "manager's" initialization and the execution part, in parallel.
Class Sched_RequestExecution	It implements the restrictions of maximum parallelism and use of futures for the functions member *execution()* and *request()* of the classes ComponentManager and ComponentStage respectively. Their methods *request_t()* and *execution_t()* implement the maxpar restrictions for the functions *request()* and *execution()* and the use of futures in the assignment of the value of return of *execution()* to an instance FutureType.

The set of the classes that define the types of own data (Table.2), used for the creation of types of abstract data to be worked by the HLPCs in form of objects:

Table 2: Classes that define own data types

Class MyInt	It defines the specific data and their type (in this case integer) that the user wants to provide to the HLPC for his processing as an instance of this class. In particular the objects of this class are used within the HLPCs Farm and Pipe.
Class MyIntDV	It defines the specific data and their type (also integer) that the user wants to provide to the HLPC for his processing as a object. In particular instances of this class are used within the HLPC TreeDV.

The set of classes that define the slave objects (Table.3), they should inherit the characteristics of Object to be worked in a generic way by the HLPCs, implementing the virtual method *resuelve()* that is where the code of solution of the problem will be.

Table 3: Classes that define slave objects

Class QsortDV	It is used for instance slave objects type QSortDV and it solves the problem partially of sorting a set of data in disorder using the algorithm of QuickSort (implementation of the part of "conquer" of the technique divide and conquer) for HLPC type TreeDV.
Class Qsort	It is used for instance slave objects type QSort and it solves the one problem of sorting a set of data in disorder using the algorithm of QuickSort within the paradigm of it divide and conquer, for HLPC type Farm and type Pipe.
Class BubleSort	Used for instance slave objects type BubleSort and it solves the problem of sorting a set of data in disorder using the algorithm of the Bubble for HLPCs type Farm or type Pipe.
Class ISort	It is used for instance slave objects type ISort and it solves the problem of sorting a set of data in disorder using the algorithm of Sorting for Insertion for HLPCs type Farm or type Pipe.
Class Invierte	It is used for instance slave objects type Invierte and it solves the problem of investing the sequence of a set of data. Instances of this type were used in the implementation of the HLPC Pipe.

The set of classes that define the HLPC Farm (Table.4) that is formed of:

Table 4: Classes that define the HLPC Farm

Main Program cpanfarm.cpp	It proves the execution of the HLPC Farm. The problem that is solved is the sorting in parallel of two arrays of integer numbers using three different algorithms of sorting (QuickSort, BubleSort 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 HLPC 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.

The set of classes that define the HLPC Pipe (Table.5), formed by the following classes.

Table 5: Classes that define the HLPC Pipe

Main Program cpanpipe.cpp	It proves the execution of the HLPC 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 HLPC 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.

The set of classes that define the HLPC TreeDV (Table 6), formed by the classes:

Table 6: Classes that define the HLPC TreeDV

Main Program cpantreeDV.cpp	It proves the execution of the HLPC 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 HLPC 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.

6. CONCLUSIONS

Method of original programming has been developed based on HLPCs.

Patterns of communication/interaction have implemented themselves within the model of the HLPC commonly used in the parallel and distributed programming: the HLPC Pipe, the HLPC Farm and the HLPC TreeDV.

The implemented HLPCs can be exploited, thanks to the adoption of the approach oriented to objects using the different mechanisms of reusability of the paradigm to define new patterns already using those built.

Well-known algorithms that solve sequential problems in algorithms parallelizable have transformed and with them the utility of the method has been proven and of the component software developed in the investigation.

The HLPCs Pipe, Farm and TreeDV conform the library of classes that intends in this work.

The restrictions of synchronization have been programmed of original form suggested by the model of

the HLPC for their parallel and concurrent operation: the maximum parallelism (MaxPar), the mutual exclusion (Mutex) and the synchronization of communication of processes readers/writers (Sync).

Of equal it forms the programming in the asynchronous future communication way for results "futures" within the HLPCs it has been carried out in an original way by means of classes.

REFERENCES

Blelloch, G.E., 1996. Programming Parallel Algorithms. *Comunications of the ACM*. Volume 39. Number 3.

Brinch Hansen, 1993. Model Programs for Computational Science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*. Volume 5. Number 5.

Brinch Hansen, 1994. SuperPascal- a publication language for parallel scientific computing. *Concurrency: Practice and Experience*. Volume 6. Number 5.

Capel, M.I. and Palma A., 1992. A Programming tool for Distributed Implementation of Branch-and-Bound Algorithms. *Parallel Computing and Transputer Applications*. IOS Press/CIMNE. Barcelona.

Capel, M.I. and Troya J.M., 1994. An Object-Based Tool and Methodological Approach for Distributed Programming. *Software Concepts and Tools*.

Capel, M.I. and Rossainz, M., 2004. A parallel programming methodology based on high level parallel compositions. *Proceedings of the 14th International Conference on Electronics, Communications and Computers*. IEEE CS press. 0-7695-2074-X.

Corradi A., Leonardi L., 1991. PO Constraints as tools to synchronize active objects. *Journal Object Oriented Programming*. Volume10: 42-53.

Corradi, A., Leonardo, L., Zambonelli, F., 1995. *Experiences toward an Object-Oriented Approach to Structured Parallel Programming*. DEIS technical report. DEIS-LIA-95-007.

Danelutto, M., Orlando, S., et al., 1995. *Parallel Programming Models Based on Restricted Computation Structure Approach*. Technical Report-Dpt. Informatica. Universitá de Pisa.

Darlington, et al., 1993. Parallel Programming Using Skeleton Functions. *Proceedings PARLE'93*. Munich (D).

Roosta, Séller, 1999. *Parallel Processing and Parallel Algorithms*. Theory and Computation. Springer.

Rossainz, M., 2005. *Una Metodología de Programación Basada en Composiciones Paralelas de Alto Nivel (CPANs)*. Thesis (PhD). Universidad de Granada.

Rossainz, M., Capel, M.I., 2006. Design and Implementation of the Branch & Bound Algorithmic Design Technique as an High Level Parallel Composition. *Proceedings of International Mediterranean Modelling Multiconference*. Barcelona, Spain.

Rossainz, M., Capel M.I., 2008. A Parallel Programming Methodology using Communication Patterns named CPANS or Composition of Parallel Object. *Proceedings of 20TH European Modeling & Simulation Symposium*. Campora S. Giovanni. Italy.

Rossainz M., Capel M.I., 2012. Compositions of Parallel Object to Implement Communication Patterns. *Proceedings of XXIII Jornadas de Paralelismo*, pp.8-13. September 19-21. Elche, Spain.