PIPELINE AS HIGH LEVEL PARALLEL COMPOSITION FOR THE IMPLEMENTATION OF A SORTING ALGORITHM

M. Rossainz-López[(a)], M. I. Capel-Tuñón[(b)], P. Domínguez-Mirón[(a)], Ivo H. Pineda Torres[(a)]

[(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, [(a)]paty.dguez.m@gmail.com, [(a)]ipineda@cs.buap.mx

ABSTRACT
Shown representation by composing Parallel Objects, Pattern communication / interaction PipeLine for implementing a sorting algorithm, through a approach of Structured Parallel Programming using High Level Parallel Compositions or CPANs. Shown the definition, design and implementation of PipeLine as a CPAN under the paradigm of object orientation in order to provide the programmer the ability to reuse the pattern in solving problems, particularly those that deal with management, optimization and information search; CPANPipe are added to a set of restrictions synchronization between processes (maximum parallelism, mutual exclusion, producer-consumer synchronization type). Synchronous communication modes, asynchronous and asynchronous future used. The sort algorithm based on a Pipeline and its design and implementation as shown CPANPipe and finally execution performance is obtained in a parallel 64-processor machine.

Keywords: CPAN Pipeline, Structured Parallel Programming, Parallel Objects, Communication Patterns.

1. INTRODUCTION

Some of the problems of the environments of parallel programming it is that of their acceptance for the users, which depends that they can offer complete expressions of the behavior of the parallel programs that are built with these environments (Danelutto and Orlando 1995). At the moment in the systems oriented to objects, the programming environments based on parallel objects are only known by the scientific community dedicated to the study of the Concurrence. A first approach that tries to attack this problem it is to try to make the user to develop his programs according to a style of sequential programming and, helped of a system or specific environment, this can produce his parallel tally (Akl 1992). However, intrinsic implementation difficulties exist to the definition of the formal semantics of the programming languages that impede the automatic parallelization without the user's participation, for what the problem of generating parallelism in an automatic way for a general application continues being open (Darlington 1993).

A promising approach alternative that is the one that is adopted in the present investigation to reach the outlined objectives is the denominated structured parallelism (Capel and Troya 1994). In general the parallel applications follow predetermined patterns of execution. These communication patterns are rarely arbitrary and not structured in their logic (Brinch Hansen 1993, Brinch Hansen 1994). The High Level Parallel Compositions or CPANs are patterns parallel defined and logically structured that, once identified in terms of their components and of their communication, they can be taken to the practice and to be available as abstractions of high level in the user's applications within an environment or programming environment, in this case the one of the orientation to objects (Rossainz and Capel 2008, Rossainz 2005). A CPAN has the following properties that can be studied in detail in (Rossainz, Pineda and Dominguez 2014):

1. Capacity of invocation of methods of the objects that contemplates the asynchronous communication ways and asynchronous future (Danelutto and Orlando 1995). The asynchronous way doesn't force to wait the client's result that invokes a method of an object. The asynchronous future communication way makes the client to wait only when needs the result in a future instant of her execution. Both communication ways allow a client to continue being executed concurrently with the execution of the method (parallelism inter-objects).
2. The objects must can to have internal parallelism. A mechanism of threads it must allow to an object to serve several invocations of their methods concurrently (parallelism intra-objects).
3. Availability of synchronization mechanisms when parallel petitions of service take place. It is necessary so that the 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 must be had of associating types dynamically to the parameters of the methods of the objects. It is needed that the system can negotiate types of generic data, since the CPANs only defines the parallel part of an interaction pattern, therefore, they must can to adapt to the different classes of possible components of the pattern.
5. Transparency of distribution of parallel applications. It must provide the transport of the applications from a system centralized to a distributed system without the user's code is affected. The classes must maintain their properties, independently of the environment of execution of the objects of the applications.
6. Performance. This is always the most important parameter to consider when one makes a new proposal of development environment for parallel applications. An approach based on patterns as classes and parallel objects must solve the denominated problem PPP (Programmability, Portability, Performance) so that it is considered an excellent approach to the search of solutions to the outlined problems.

With the basic set of classes of the model of programming of PO they are possible to be constructed concrete CPANs. To build a CPAN, first it should be had clear the parallel behavior that one needs to implement, so that the CPAN in itself is this pattern. Several parallel patterns of interaction exist as are the farms, the pipes, the trees, the cubes, the meshes, the matrix of processes, etc. So you know what pattern is best suited to a particular application is an important design decision that cannot be fully automated. Once identified the parallel behavior, the second step consists on elaborating a graphic of its representation as mere technique of informal design of what will be later on the parallel processing of the objective system; it is also good to illustrate its general characteristics, etc., and it will allow later to define its representation with CPANs, following the pattern proposed in the next section.

When the model of a CPAN is already had concretized, that defines a specific parallel pattern; say for example, a tree, or some of those previously mentioned ones, the following step would be to carry out its syntactic definition and semantics.

Finally, the syntactic definition previous to a CPAN programmed is translated in the most appropriate programming environment for its parallel implementation. It would be verified that the resulting semantics is the correct one, it would be proven with several different examples to demonstrate its genericity and the performance of the applications would be observed that include it as component software.

The parallel patterns worked in the present investigation have been the pipeline, to be a significant reusable pattern in multiple applications and algorithms. Being used at the moment with different purposes, in different areas and with different applications according to the literature that there is on the topic (Cole 1989, Darlington 1993). In the present investigation the pipeline is constructed as a CPAN to implement a sorting algorithm and solving the problem of generating parallel an increasing sequence of numbers from lowest to highest. Finally performance shown considering measures such as the execution time in seconds, cycles per instruction (CPI), speedup and Amdahl's Law.

2. HIGH LEVEL PARALLEL COMPOSITIONS (HLPC)

A CPAN comes from the composition of a set of objects of three types (see Figure 1): An object manager that it represents the CPAN in itself and makes of him an encapsulated abstraction that it hides their internal structure. The manager controls the references of a set of objects (a denominated object Collector and several denominated objects Stage) that represent the components of the CPAN and whose execution is carried out in parallel and it should be coordinated by the own manager. The objects Stage that are objects of specific purpose, in charge of encapsulating an interface type client-server that settles down between the manager and the objects slaves (objects that are not actively participative in the composition of the CPAN, but rather they are considered external entities that contain the sequential algorithm that constitutes the solution of a given problem). And an object Collector that it is an object in charge of storing in parallel the results that he receives of the objects stage that has connected. That is to say, during the service of a petition, the control flow within the stages of a CPAN depends on the implemented communication pattern.

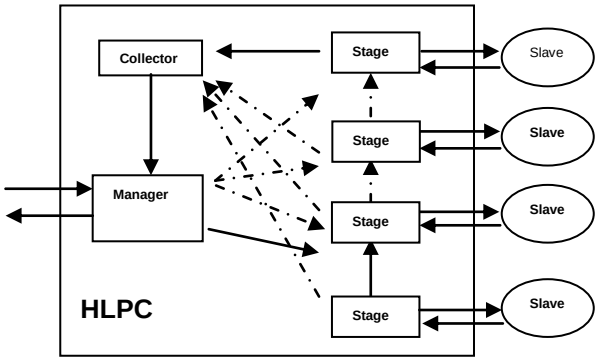


Figure 1: Internal Structure of a HLPC. Composition of its components

The objects manager, collector and stages are included within the definition of Parallel Object (PO) (Corradi 1991). The Parallel Objects are active objects, that is to say, objects that have execution capacity in them (Corradi 1991). 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 (Danelutto and Orlando 1995, Corradi 1991). 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) (Birrel 1989). 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 inheritances.

```
CLASS: <class name>
    INHERITS_FROM: <list of class names parents>
    INSTANCE_STATE: < list the names and
                      types of instance>
    INSTANCE_METHODS: <list of public methods>
   PRIVATE_METHODS: <list of private methods>
   SCHEDULING_PART: <list of synchronization
                      restrictions>
END_CLASS <class name>
```

2.1. The abstract class ComponentManager

It defines the generic structure of the component manager of a CPAN, of where will be derived all the manager concretes depending on the parallel behavior that is contemplated for the creation of the CPAN. All concrete instance of a manager accepts as input a list of n-associations. 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 execute through one of their methods.

Once the manager has captured the list of n-associations, this generates the concrete stages, one for each association and then each stage becomes in responsible for an object slave, together with the execution method of this. In turn, the stages are connected to each other, in accordance with the parallel pattern that has been implemented in the CPAN.

Finally the manager carries out the processing of a computation, through the execution of one of his methods (execution). For it is necessary to pass to the method the data input with those that one wants to work. The manager generates a component collector then and it passes to the stages the reference to this collector, as well as the data input. The stages processes them and, according to as be connected some with other, will spend the results that they go obtaining. At the end the collector will pick up the results of the stages to return them to the manager who finally will transmit the results to the exterior of the CPAN (to the code of user's application that uses it). The following syntactic definitions have been written using the grammar free of context that is in the appendix A of the present document.

```
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 calculations in parallel. The synchronization policies used for it is it the restriction of synchronization of the "maximum parallelism" or MAXPAR applied to the method "execution()."

2.2. The abstract class ComponentStage

It defines the generic structure of the component stage of a CPAN, as well as of their interconnections, of where they will be derived all the concrete stages depending on the parallel behavior that is contemplated in the creation of the CPAN. All concrete instance of a stage accepts as input a list of associations slave_object-method to work with her and it is connected or not with the following stage of the list of associations, depending on the parallel pattern that you/they implement, so that when the manager commands to the stages in parallel, each one of them makes the object slave to carry out the execution of his method, the stage captures the results and it sends them to the following stage or the collector, depending on the implemented structure (everything it within a method called request ()). In turn, each stage can command others in the execution of the computation initiate 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 a petition of the operation "request ()" being executed in parallel, for what the synchronization policies that must be used for its correct implementation is the one that provides the synchronization restriction of "maximum parallelism", applied now to the method "request()."

2.3. The concrete class ComponentCollector

It defines the concrete structure of the component collector of any CPAN. This component implements a buffer multi-item basically, where they will leave storing the results of the stages that make reference to this collector. This way one can obtain the result of the calculation initiate 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 case the used synchronization restrictions are SYNC and MUTEX to be able to synchronize the communication concurrently among the methods *put()* and *get()* and to provide mutual exclusion.

3. THE PIPELINE AND ITS REPRESENTATION AS CPAN

The Pipeline this compound for a set of interconnected states one after another. The information follows a flow from a state to another.

3.1. 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 (Robbins and Robbins 1999), (Figure 6):

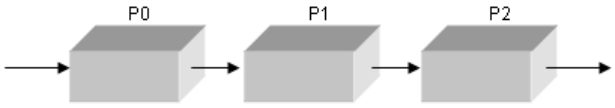


Fig. 2: Pipeline

The processes of the pipeline are sometimes called stages of the pipeline (Roosta 1999). 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.

An algorithm that solves a specific problem can then be formulated as a pipeline if can be divided into a number of functions that could be performed by the stages of pipe (Robbins and Robbins 1999). Suppose, for example, we want to order a disordered set of data from highest to lowest in descending order, but you have already implemented a sorting algorithm ascending; if this sort algorithm were used, it would be necessary to reverse the sequence of already sorted data, which can be carried out by adding an extra stage to the pipeline with an assigned role to carry out the specific process (see Figure 3).

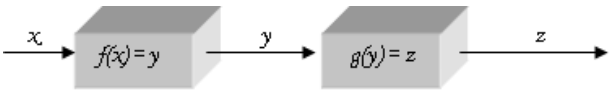


Figure 3: Pipeline seen as functional decomposition

The interpretation of the elements of the previous figure is as follows:

- x represents the initial dataset that we assume is in disarray;

- f(x) represents the function "orders" which receives as input the data set to order and outputs the sort in ascending order (from low to high) dataset that come;
- y represents the output of the function f (x), ie, the data sorted;
- g(y) represents the function "invest" that receives the result of the function "orders" to output the data set previously ordered but reversed in sequence to have a descending order (from highest to lowest);
- z is the output of the function g (y) in the last stage of the pipeline.

Assuming that the data set with which they work in this example are integers, the sequence of results within the pipeline would be as follows, see Figure 4.

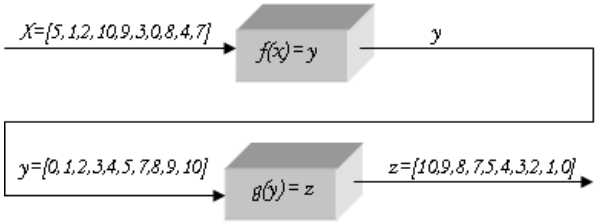


Figure 4: Sequence results in Pipeline

If a problem can be divided into a series of sequential tasks, pipelined approach provides increased speed of execution in the following three types of calculations:

1. When more than one instance of the problem can be run in parallel;
2. Either a series of data can be processed and each of these is used in multiple operations;
3. Or if the information required by the following process to start your calculation happens after the current process has completed all its internal operations.

With this technique many computational problems are performed sequentially can be easily parallelized as a pipeline. Examples of these problems are:

- the sum of numbers,
- the shorting numbers,
- generating prime numbers,
- Solving a system of linear equations.

The technique of parallel processing pipeline is then presented as a High Level Parallel Composition applicable to solving a range of problems that are partially sequential in nature, so that the Pipe CPAN guarantees code parallelization of sequential algorithm using the pattern Pipeline.

3.2. Representation of the Pipeline as a CPAN

The Figure 5 represents the parallel pattern of communication Pipeline as a CPAN.

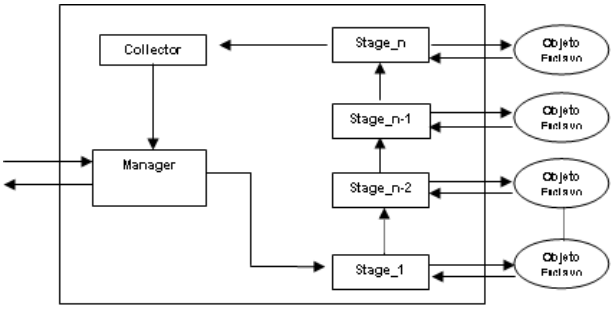


Figure 5: The CPAN of a Pipeline

The objects *stage_i* and *Manager* of the graphic pattern of the CpanPipe are instances of concrete classes that inherit the characteristics of the classes ComponentManager and ComponentStage. The Collector is the only object that is an instance of the base class ComponentCollector since this is a concrete class. Once the objects are created and properly connected according to the parallel pattern Pipeline, then you have a CPAN for a specific type of parallel pattern, and can be resolved after the allocation of objects associated with slave stages.

3.3. Semantic and Syntactic Definition of the Cpan Pipe

The Cpan 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. For details see (Capel and Rossainz 2004). 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);
  }
};
```

4. PARALLEL ALGORITHM SORTING WITH PIPELINE

Using a PipeLine is useful to introduce a scheme of parallelization of a sorting algorithm, so that to solve the problem have to perform a series of operations on a data set. Each of these transactions is considered a stage in the data processing and each is executed by a separate process that synchronizes with the above processes and form respective next stage. The complete data processing ends when they have passed through every stage.

Pipeline processing a serial data sequence they pass through the pipeline stages. Each stage is associated with a process that performs a specific operation when a fact comes through its associated slave object. Completed this operation, passes the result to the next stage. In a parallel sorting algorithm with a pipeline 3 phases are distinguished (Barry and Allen 1999; Blelloch 1996):

- The initial charge: data is allocated to all processes associated with the stages of the pipeline. In this phase the processes are running the same code in the second phase, the difference is that you must initialize properly to receive the first data, they will come from the previous stage or initial program load.
- The processing of the data stream with maximum efficiency: Processes behave cyclically in execution. Data support the previous stage, process and send the result to the next stage. Each process has to be synchronized with that of the previous stage to not send new data when it has not yet finished processing the data streams; but also to the next step, to not send the result to the process of this stage is not ready to receive it. The final process has a special behavior with respect to the processes associated with the above steps as you have to run a routine or exit code and presentation of results of the program. Its operation is to obtain the data sent by the process of the last stage of the pipeline and send them to an output device or send a termination condition the main program. The series of results it produces the last process must match the expected result of the algorithm has been parallelized, if the pipeline has been successfully parallelized.
- Download: In this last stage the processes send the result of the last processed data and themselves detect termination situation, as they will no longer receive more data from the input stream and should not pose any global control in the program tells them when they have finished. Processes for transmitting the data stored in its stages before completion, is usually introduced a special value at the end of the input sequence used to unload the pipeline.

To implement the parallel sorting algorithm, a pipeline process is used, which receives an unordered set of integers by a routine or entry code. It is obtained as a result the ordered sequence of integers ascending. The number of values in the input sequence cannot be greater than the number of pipeline stages. Each pipeline processes can store an integer, which will be the largest that has been received so far from the previous step. In each iteration, a process receives a integer, compared to the one that had stored and sends the smaller of the next stage of the pipeline, while the highest is stored (Barry and Allen 1999; Blelloch 1996). Suppose that at a certain moment of the execution of the algorithm, the first three stages of the pipeline have a stored integer, while the fourth stage has not yet received any. Suppose the integer 2 is received from the beginning of the pipeline (see figure 6).

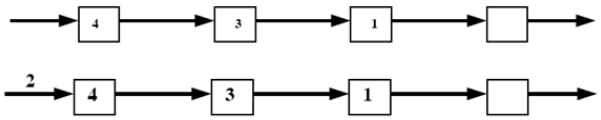


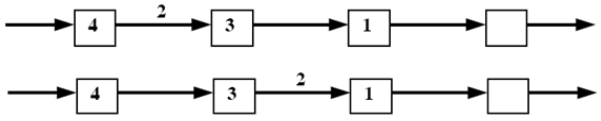
Figure 6: insertion sequence of integers in a pipeline

The first process compares the value received to its stored value, sends the least of them (2) to the next stage of the pipeline and stores the largest (4).

Similarly, the second process, after receiving the integer sent by the first (2), sends it to the next stage, keeping stored the integer (3).

The third process, however, sends the integer that was stored at the next stage and stores just received (2).

The fourth process, not having yet a stored integer, stores the received by the previous stage (see figure 7).



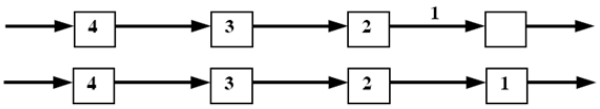


Figure 7: Sort sequence of integers on a Pipeline

The above figure represents the parallel sorting algorithm using a Pipeline. CpanPipe design shown in figure 8.

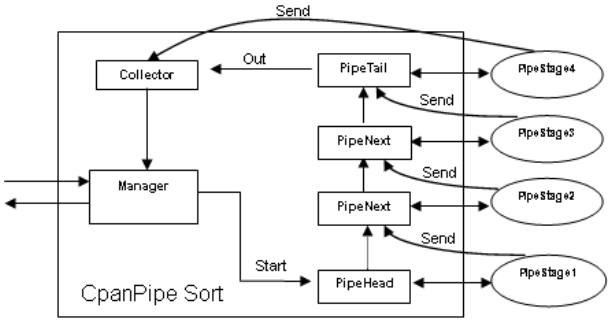


Figure 8: Ordering integers using CpanPipe

5. PERFORMANCE

CpanPipe performance to implement a parallel sorting algorithm was carried out on a parallel computer with 64 processors, 8 GB of main memory, high-speed buses and distributed shared memory architecture. Performance measures obtained in implementing the CpanPipe that solves the problem of sorting is carried out with the following restrictions execution:

- The same sequential algorithm of comparing values in each of the slave objects associated with pipeline stages except the first and last step as described in Chapter IV,
- 50000 a set of whole numbers randomly obtained in the range of 0-50000 ordered, allowing make a sufficient charge for processors and thereby observe the performance improvement CpanPipe,
- CpanPipe execution for 2, 4, 8, 16 and 32 full-time processors.

Table I and figure 9 show the series of measurements obtained including their corresponding sequential versions, CpanPipe execution time in seconds, cycles per instruction executed, magnitude speedup found and the upper bound on the magnitude of speedup using for that Amdahl's law.

Table I: Cpan Perfomance Pipe Parallel to the Management of 50000 Sorting integers

CPAN Pipe	Pipe Secuencial	cpuset2	cpuset4	cpuset8	cpuset16	cpuset32
Runtime in seconds	238.50	127.27	123.83	116.97	115.63	111.03
CPU time in seconds	231.82	224.79	217.80	203.98	198.30	191.89
CPI	1.862	0.909	0.904	0.901	0.886	0.871
Speedup	1.00	1.87	1.93	2.04	2.06	2.15
Amdalh	1.00	1.89	3.39	5.63	8.42	11.19

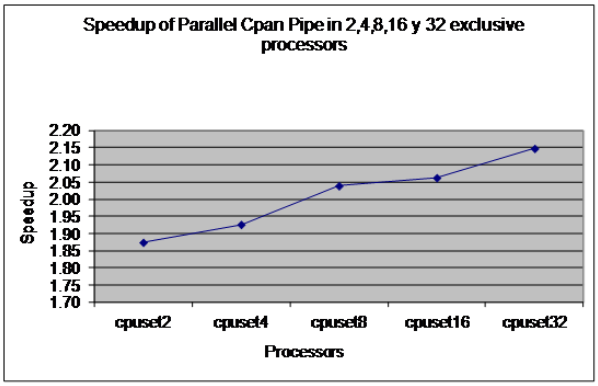


Figure 9: Scaling the magnitude of CpanPipe Speedup for 2, 4, 8, 16 and 32 exclusive processors

6. CONCLUSIONS

It has created a programming methodology based on High Level Parallel Compositions or CPANs (Rossainz and Capel 2008). CPAN Pipeline has been implemented which is part of a class library of parallel objects that provides the user the pattern cited as CPAN and other communication patterns as farms and trees and the details of which are published in (Rossainz and Capel 2012). The CPAN Pipe can be exploited using the approach object-oriented to define new communication patterns based on the already built. For more details see references (Rossainz and Capel 2006, Rossainz and Capel 2012).

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.

It has become a sequential sorting algorithm on a parallelizable algorithm using a Pipeline as CPAN. It has implemented the synchronization restrictions suggested by the model Cpan: maximum parallelism (MaxPar), mutual exclusion (mutex) communication and synchronization producer / consumer processes (Sync). Of equal it forms the programming in the asynchronous future communication way for results "futures" within the Cpans it has been carried out in an original way by means of classes.

It has been proven performance Pipe Cpan by metrics Speedup, Amdahl's Law and efficiency to demonstrate that parallel behavior CpanPipe is better than its sequential counterpart.

APPENDIX A. FREE GRAMMAR OF THE CONTEXT FOR THE DEFINITION OF CLASS PO (PARALLEL OBJECTS)

Definición de una clase PO::=CLASS <tipo de clase> <nombre de clase> [herencia]

{
[<Definición de variables de instancia>]
[<Definición de métodos públicos>]
[<Definición de métodos privados>]
[<Definición de restricciones de sincronización>]
};

tipo de clase::= ABSTRACT /CONCRETE

*nombre de clase::= <letra mayúscula>{letra /dígito}**

herencia::= EXTENDS OF <nombre de clase>

letra::= <letra mayúscula> /<letra minúscula>
letra mayúscula::= A /B /C /D /E /F /G /. . . /Y /Z

letra minúscula::= a /b /c /d /e /f /g /. . . /y /z

dígito::= 0 /1 /2 /3 /4 /5 /6 /7 /8 /9

Definición de variables de instancia::={<tipo> <nombre variable>{,nombre variable};}**

tipo::= <tipo primitivo> /<tipo array> /<tipo clase> / ASOCIACION /ANYTYPE /METHOD /FUTURETYPE

tipo primitivo::= CHAR /INT /REAL /DOUBLE / LONG /FLOAT /VOID /BOOL /STRING

tipo array::= <tipo>[ ]
tipo clase::= <nombre de clase>
*nombre variable::= letra{letra /dígito}**

Definición de métodos públicos::={PUBLIC <tipo> <Nombre Método>([<parámetros>])
[<declaración de variables locales>]
{
<cuerpo>
*} }**

Definición de métodos privados::= { PRIVATE <tipo> <Nombre Método>([<parámetros>])
[<declaración de variables locales>]
*{ <cuerpo> } }**

Nombre Método::= letra{letra}[{dígito}*]*

*Parámetros::= <tipo> <nombre parámetro>{,<tipo> <nombre parámetro>}**

*Nombre parámetro::= letra{letra /dígito}**

Declaración de Variables locales::= VAR
<variables locales>{; <variables locales>};*

*variables locales::= <tipo> <nombre variable>{,<nombre variable>}**

*cuerpo::= ABSTRACT; /{<sentencia>}**

sentencia::= <asignación>
/<condición>
/<ciclo>
/<ejecución de método>
/RETURN <expresión>
/<composición de objetos paralelos>

asignación::= <nombre de variable> = <expresión>;
/<definición de objeto> = <creación de un objeto>;

<definición de objeto>::= <tipo clase> <nombre de objeto>

*<nombre de objeto>::= <letra>{letra /digito}**

<creación de un objeto>::=<nombre de clase> CREATE ([<parámetros>])

<uso de un objeto>::= <nombre de objeto>. <nombre variable>; /<nombre de objeto>. <Nombre método>([argumentos]);

expresión::= HEAD(ASOCIACION[ ])
/TAIL (ASOCIACION[ ])
/CONS(ANYTYPE[ ], ANYTYPE)
/<creación de un objeto>
/<uso de objeto>
/<nombre de objeto> EVAL (<nombre metodo>, argumento)

condicion::= IF (expresión booleana)
{
*sentencia;{sentencia;}**
}
ELSE {
*sentencia;{sentencia;}**
}

ciclo::= <ciclo condicional> /<ciclo no condicional>

ciclo condicional::= WHILE (expresión booleana)
{
*sentencia;{sentencia;}**
}

ciclo no condicional::=FOR <nombre variable>=(expresión entera, expresión entera)
{
*sentencia;{sentencia;}**
}

ejecución de método::=<nombre método>([argumentos])

composición de objetos paralelos::= <modo síncrono>
/<modo asíncrono>

modo síncrono::= <nombre de objeto>.<nombre método>([argumentos]);

modo asíncrono::=THREAD<nombre de objeto>.<nombre método>([argumentos]);

Definición de restricciones de sincronización::=
MUTEX (<nombre método>)

/*MUTEX (<nombre método>, <nombre método>)*
/*SYNC (<nombre método>, <nombre método>)*
/*MAXPAR(<nombre método>)*

REFERENCES

Akl S.G., 1992. Diseño y Análisis de Algoritmos Paralelos. *Ra-Ma Serie Paradigma*, Madrid.

Birrell, Andrew, 1989. An Introduction to programming with threads. *Digital Equipment Corporation*, Systems Research Center.

Blelloch, Guy 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.

Barry W., Allen M., 1999. Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers. *Prentice Hall*. ISBN 0-13-671710-1.

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

Capel M., 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.

Cole, M., 1989. Algorithmic Skeletons: Structured Managment of Parallel Computation. *The MIT Press*.

Corradi A., Leonardi L., 1991. PO Constraints as tools to synchronize active objects. *Journal Object Oriented Programming* 10, pp. 42-53.

Corradi A, Leonardo L, Zambonelli F., 1995. Experiences toward an Object-Oriented Approach to Structured Parallel Programming. *DEIS technical* report no. 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).

Robbins, K. A., Robbins S. 1999. "UNIX Programación Práctica. Guía para la concurrencia, la comunicación y los multihilos". *Prentice Hall*.

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). *Universidad de Granada, PhD dissertation*, 02/25/2005.

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

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

Rossainz, M., Capel M., 2012. Compositions of Parallel Objects to Implement Communication Patterns. *XXIII Jornadas de Paralelismo. SARTECO 2012*. Septiembre de 2012. Elche, España.

Rossainz M., Pineda I., Dominguez P., Análisis y Definición del Modelo de las Composiciones Paralelas de Alto Nivel llamadas CPANs. *Modelos Matemáticos y TIC: Teoría y Aplicaciones 2014*. Dirección de Fomento Editorial. ISBN 987-607-487-834-9. Pp. 1-19. México.