USING PIPELINE AND BINARY-TREE AS CPANS FOR SOLVING THE SORTING PROBLEM. A COMPARATIVE STUDY

M. Rossainz-López[(a)], Manuel I. Capel[(b)], O. Carrasco-Limón[(a)], B. Sánchez-Rinza[(a)]

[(a)] Faculty of Computer Science, Autonomous University of Puebla, San Claudio Avenue and South 14th Street, San Manuel, Puebla, Puebla, 72000, México
[(b)] Software Engineering Department, College of Informatics and Telecommunications ETSIIT, University of Granada, Daniel Saucedo Aranda s/n, Granada 18071, Spain

[(a)]rossainz@cs.buap.mx, [(b)]manuelcapel@ugr.es, [(a)] odondavidcarrasco95@gmail.com, [(a)]brinza@hotmail.com

ABSTRACT
This paper proposes the model of the High Level Parallel Compositions or CPANS (Acronym in Spanish) to communication patterns/interaction Pipeline and Binary Tree for implementing a sorting algorithm by Structured Parallel Programming approach based on the concept of Parallel Objects. The CPANS Pipeline and TreeDV are displayed using the paradigm of object orientation and sorting problem is solved using two different algorithms; it is using a pipeline process to sort a dataset in disordered (CPAN Pipe) and one that by quick sort uses a binary tree for the ordering of the same dataset disordered by divide and conquer technique (CPAN TreeDVQS). Each proposal of CPAN contains a predefined set of restrictions of synchronization between processes (maximum parallelism, mutual exclusion and synchronization of producer-consumer type), and the use of synchronous, asynchronous and asynchronous future communication modes. Sorting algorithms, their design and implementation as CPANs and comparative performance metrics on a parallel machine 64 processors are shown.

Keywords: CPAN Pipeline, CPAN Binary Tree, Structured Parallel Programming, Communication Patterns.

1. INTRODUCTION

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 (Bacci and et-al 1999), The necessity to have patterns or High Level Parallel Compositions, the Determination of a complete set of patterns as well as of their semantics (Corradi, and Zambonelli, 1995), the adoption of an object-oriented approach (Corradi and Leonardi, 1991, Darlington 1993). The 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 (Rossainz 2005, Rossainz and Capel 2008). 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 and to solve the problem of sorting.

2. HIGH LEVEL PARALLEL COMPOSITIONS (CPAN)

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.

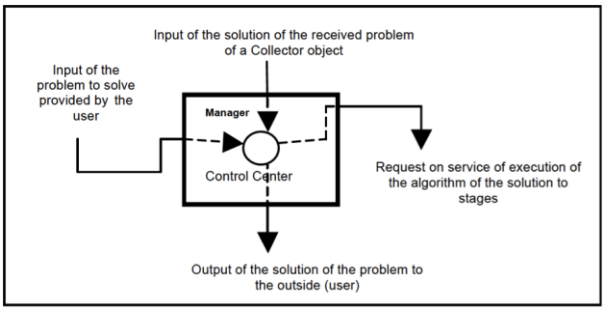

Figure 1: Component MANAGER of model CPAN

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.

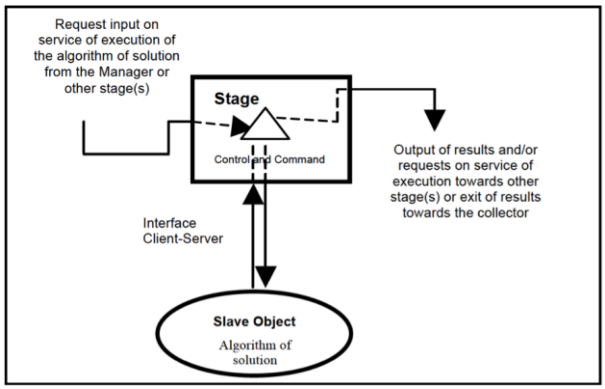

Figure 2: Component Stage of model CPAN and its associated slave object

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.

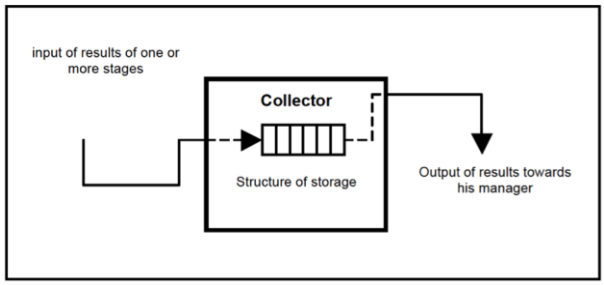

Figure 3: Component Collector of model CPAN

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, a slave object will be in charge of implementing the necessary functionalities to solve the sequential version of the problem being solved (Figure 4). For details CPAN model, see (Rossainz and Capel 2014).

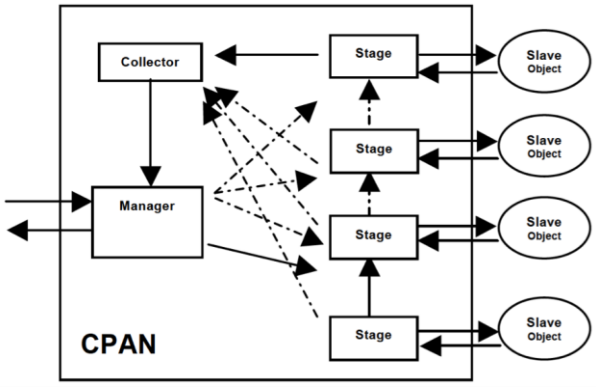

Figure 4: Internal structure of CPAN. Composition of its components

The Figure 4 shows the pattern CPAN in general, without defining any explicit parallel communication pattern. The box that includes the components, represents the encapsulated CPAN, internal boxes represent compound objects (collector, manager and objects stages), as long as the circles are the objects slaves associated to the stages. The continuous lines within the CPAN suppose that at least a connection should exist between the manager and some of the component stages. Same thing happens between the stages and the collector. The dotted lines mean more than one connection among components of the CPAN.

2.1. The CPAN seen as composition of parallel objects

Manager, collector and stages are included in the definition of a Parallel Object (PO), (Corradi 1991). Parallel Objects are active objects, which is equivalent to say that these objects have intrinsic execution capability (Corradi 1991). 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 (Danelutto and Orlando 1995, Corradi 1991). 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 (Birrel 1989). 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 super-class. Parallel objects support multiple inheritance in the CPAN model.

2.2. Communication types in the parallel objects of CPAN

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 (Andrews 2000).
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 (Andrews 2000).
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. For details see (Lavander and Kafura 1995).

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

2.3. 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 - the implementation details are in (Rossainz, Pineda and Domínguez 2014) - are the next ones:

1. 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.
2. 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.
3. 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.

2.4. The Synchronization restrictions of a CPAN

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.

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 functions of the ComponentManager class and to the functions of the ComponentStage class.

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 functions of an object collector.
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 of the componentCollector class. SYNC helps to synchronize these methods when accessing the shared resource at the same time, which in this case is a multi-item list.

The details of the algorithms and their implementation can be seen in (Rossainz, Pineda and Domínguez 2014).

3. CONSTRUCTION OF A CPAN

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, and 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 and the binary-tree to solve the sorting problem using two different algorithms.

4. THE CPAN PIPELINE

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

4.1. 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, see figure 5. In a pipeline each task can be executed by a process, thread or processor for separate (De Simone 1997, Robbins and Robbins 1999).

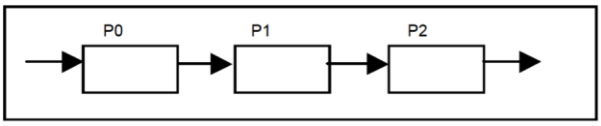

Figure 5: Pipeline

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.

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.

4.2. Representation of the Pipeline as a CPAN

The Figure 6 represents the parallel pattern of communication Pipeline as a CPAN.

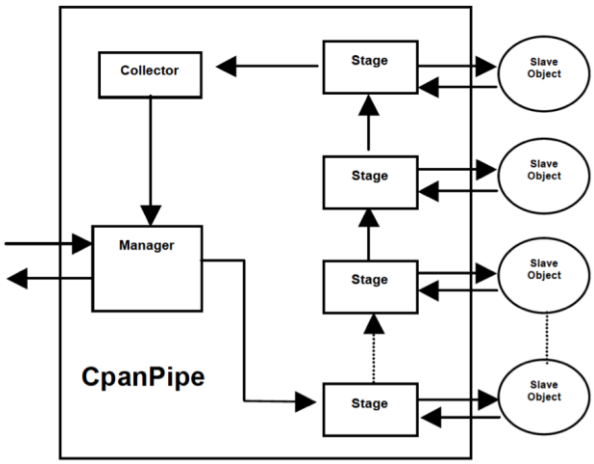

Figure 6: The CPAN of a Pipeline

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.

5. 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 (Wilkinson and Allen M., 1999).

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; Roosta 1999):

- 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). For more details see (Rossainz, Capel and Domínguez 2015).

6. THE CPAN TREEDV

The programming technique is presented it Divide and Conquer as a CPAN, applicable to a wide range of problems that can be parallelizable within this scheme, in particular to solve sorting problems in parallel (Rossainz and Capel 2012).

6.1. 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 (Brassard and Bartley 1997). 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, and then they combine the partial results of each sub-problem to obtain at the end the solution to the initial problem (Brassard and Bartley 1997). 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.

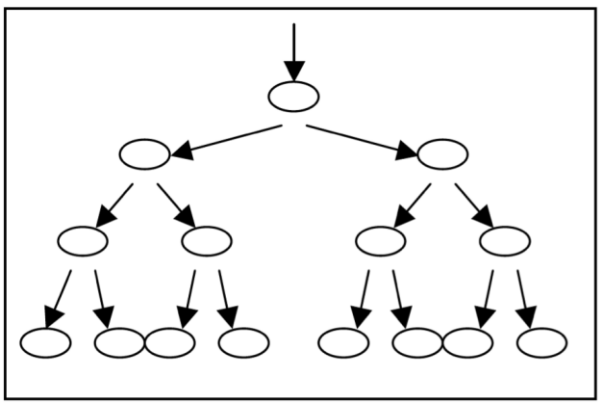

Figure 7: Binary Tree

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 (figure 7). 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 Hansen 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.

6.2. Representation of the TreeDV as a CPAN

The representation of the patron tree that defines the technique of it Divide and Conquer as CPAN has their model represented in figure 8.

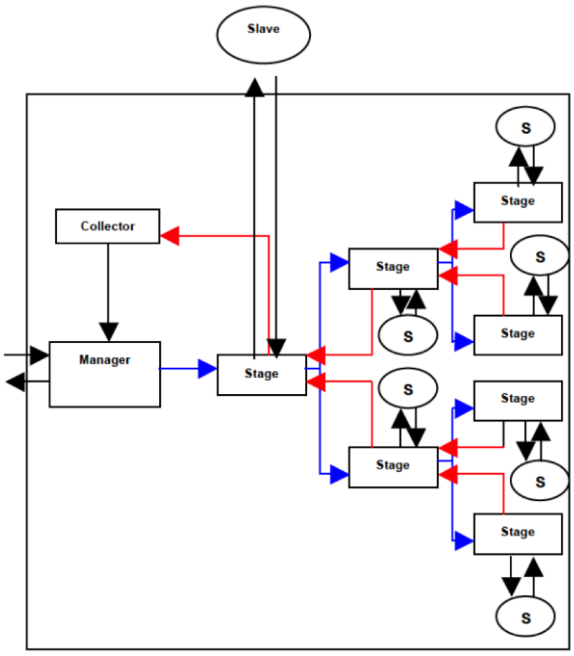

Figure 8: The Cpan of a TreeDV

Contrary to the previous model, where the objects slaves were predetermined outside of the pattern CPAN, 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.

6.3. Quicksort sorting algorithm using the CPAN TreeDV

The Quicksort sorting was created by Hoare and is based on the paradigm of divide and conquer. As a first step the algorithm selects as a pivot one of the elements of the data set you have to order. The array is then partitioned on either side of the pivot: elements are moved so that those greater than the pivot are to its right, whereas the others are to its left. If now the sections of the array on either side of the pivot are sorted independently by recursive and parallel calls of the algorithm (Brassard and Bratley 1997), in this case through the stage TreeDV CPAN objects, the final result is a completely sorted array, no subsequent merge step being necessary.

```
Algorithm QuickSort(T[i,..j])
{
 var l;
  if ( j-i  is sufficiently small then  insert(T[i..j]) )
 else {
         l= pivot(T[i..j]);
        QuickSort(T[i..l-1]);
        QuickSort(T[l+1..j]);
       }
 }
```

To balance the sizes of the two subinstances to be sorted, we would like to use the median element as the pivot. Unfortunately, finding the median takes more time it is worth. For this reason we simply use an arbitrary element of the array as the pivot, hoping for the best.

```
Algorithm pivot(T[i..j])
 {
  var l;
   p=T[i];  k=i;  l=j+1;
   repeat {  k=k+1;  } until (  (T[k]>p)   or  ( k>=j) );
    repeat {   l=l-1;   } until ( T[l]<=p );
    while ( k<l )
      {
       swap(T[k],T[l]);
        repeat {  k=k+1; } until (  T[k]>p );
        repeat {  l=l-1;  } until ( T[l]<=p );
      }
    swap(T[i],T[l]);
   return l;
 }
```

Suppose subarray *T[i..j]* is to be pivoted around *p=T[i]*. One good way of pivoting consists of scanning the subarray just once, but starting at both ends. Pointers *k* and *l* are initialized to *i* and *j+1*, respectively. Pointer *k* is then incremented until *T[k]>p*, and pointer *I* is decremented until *T[l]<=p*. Now *T[k]* and *T[l]* are interchanged. This process continues as long as *k<l*. Finally, *T[i]* and *T[l]* are interchanged to put the pivot in its correct position (Brassard and Bratley 1997).

7. PERFORMANCE

Performance analysis of CPANS Pipeline and TreeDV solving sorting problems are shown. The aim is to show that, at least for these problems, the performances obtained are "good" based on the model of the CPAN.

CpanPipe and CpanTreeDV 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 and CpanTreeDV that solves the problem of sorting using an Pipeline and Binary Tree respectively, is carried out with the following restrictions execution:

- Parallel implementation of sequential sorting algorithm based on a pipeline in the case of CPAN Pipe and parallel-sequential implementation is Quicksort sorting algorithm based on a binary tree using the technique of Divide and Conquer for the case of CpanTreeDV.
- In both cases, both the CPAN Pipe as CPAN TreeDV, it is implemented the same sequential algorithm of comparing values in each of the slave objects associated with the stages of CPANs.
- 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 and TreeDV,
- CpanPipe and CpanTreeDV execution for 2, 4, 8, 16 and 32 full-time processors.

The methodology that has been followed for the analysis of performance CPANS is:

1. The CPANs Pipeline and TreeDV are compiled in their sequential and parallel versions and run on the corresponding cpuset,
2. The following parameters of the execution performance of CPANs are measured. They show their behavior.
   - 2.1. Runtime of each CPAN, including its sequential version and measurement of page faults caused in the system during its execution.
   - 2.2. Cycles per instruction (CPI) for each CPAN, including sequential versions.
   - 2.3. Page faults caused during the execution of the CPANs.
   - 2.4. Magnitude speedup for each execution of the CPANs in Cpuset about their sequential versions.
   - 2.5. Upper bound of the magnitude speedup for each CPAN using Amdahl's law.

Tables 1, 2 and figures 9, 10, show the series of measurements obtained including their corresponding sequential versions for Cpans Pipe and TreeDV, 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 1: Cpan Perfomance Pipe Parallel to the Management of 50000 Sorting integers

CPAN PIPE	Sequential Pipe	CPU SET 2	CPU SET 4	CPU SET 8	CPU SET 16	CPU SET 32
Runtime in seconds	238.5	127.27	123.83	116.97	115.63	111.03
CPU time in seconds	231.82	224.79	217.8	203.98	198.3	191.89
CPI	1.862	0.909	0.904	0.901	0.886	0.871
Speedup	1	1.87	1.93	2.04	2.06	2.15
Amdalh	1	1.89	3.39	5.63	8.42	11.19

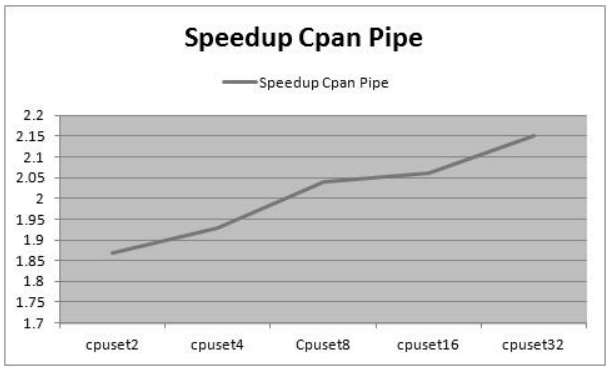

Figure 9: Scaling the magnitude of CpanPipe Speedup for 2, 4, 8, 16 and 32 exclusive processors

Table 2: Cpan Perfomance Binary Tree Parallel to the Management of 50000 Sorting integers

CPAN TREEDVQS	Sequential TreeDVQS	CPU SET 2	CPU SET 4	CPU SET 8	CPU SET 16	CPU SET 32
Runtime in seconds	20.55	11.48	7.21	4.59	3.68	3.40
CPU time in seconds	7.60	7.18	6.31	6.88	6.45	6.35
CPI	1.261	0.900	0.892	0.858	0.849	0.836
Speedup	1	1.79	2.85	4.48	5.58	6.04
Amdalh	1	1.82	3.08	4.71	6.40	7.80

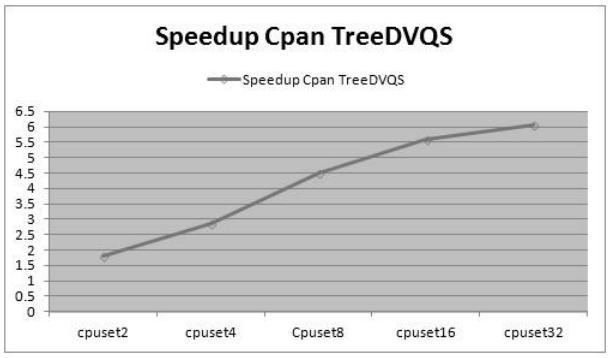

Figure 10: Scaling the magnitude of CpanTreeDV Speedup for 2, 4, 8, 16 and 32 exclusive processors

The measurements obtained from page faults caused by the implementation of Pipeline and TreeDV CPANs shown in Table 3 and Figure 11.

A measure page fault is useful to see if the CPANS cause excessive paging, especially if they use a lot of memory for execution, which has not been.

Table 3: Page faults in the execution of CPANs

CPANS	SEQ		Cpuset2		cpuset4		cpuset8		cpuset16		cpuset32	
	ma-yor	mi-nor	ma-yor	mi-nor	ma-yor	mi-nor	ma-yor	mi-nor	ma-yor	mi-nor	ma-yor	mi-nor
Pipe	0	25	0	37	0	37	0	37	0	37	0	37
TreeDVQS	0	25	0	38	1	38	0	38	0	38	0	38

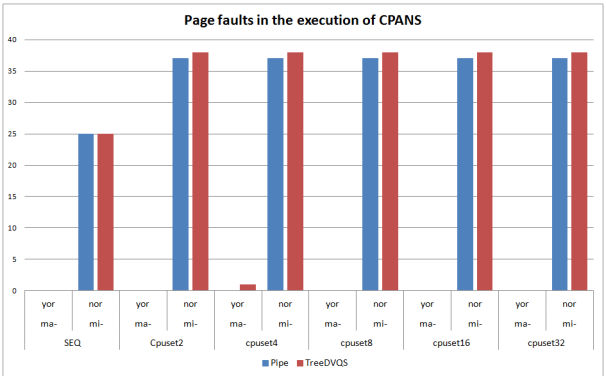

Figure 11: Equivalence of page faults in the execution of the CPANs

Parallel executions of CPANS have a time shorter than the time used by their corresponding sequential versions, as expected. The execution times of their parallel versions CPANS improve as the number of processors is increased, ie, as is increasing the number of processors with which CPANS are executed, their execution times are decreasing. A value of the magnitude called speedup is appreciated ever upward on improving execution times of parallel CPANS respect to its sequential counterpart, but always below the levels of Amdahl's Law calculated, obtaining "good" yields. Improved CPI is obtained by increasing the number of processors, that is, a larger number of processors used in performing the CPANS, the lower the value of the ratio of cycles per instruction. This indicates that while the number of instructions in the execution of the application within cpusetX remains more or less constant, the number of cycles per instruction decreases, resulting in a gain in the final value of the CPI.

8. CONCLUSIONS

We have implemented communication patterns Pipeline and Binary Tree as CPANS and with them has solved the problem of sorting parallelizing two sequential algorithms different, it is using a pipeline process to sort a dataset in disordered and one that by quick sort uses a binary tree for the ordering of the same dataset disordered by divide and conquer technique.

The implemented CPANS can be exploited, thanks to the adoption of the approach oriented to objects. Well-known algorithms that solve sequential problems in algorithms parallelizable have transformed and with them the utility of CPANS has been proven.

It has become a sequential sorting algorithm on a parallelizable algorithm using a Pipeline and a Binary Tree as CPAN. It has been proven performance Cpan Pipe and Cpan TreeDV by metrics Speedup, Amdahl's Law and efficiency to demonstrate that parallel behavior CpanPipe is better than its sequential counterpart. Furthermore the speedup TreeDVQS CPAN is much closer to the upper bound (Amdahl law) that the speedup obtained in CPAN Pipe. The runtime in seconds of CPAN TreeDV to solve the problem of sorting is much less than the CPAN Pipe that solves the same problem with the same input size, according increase the number of processors. The same applies to the CPU time in the execution of those CPANS (see Table 1, 2 and Table 3).

REFERENCES

Andrews G.R., 2000. Foundations of Multithreaded, Parallel, and Distributed Programming, *Addison-Wesley*

Brassard G., Bratley P., 1997. Fundamentos de Algoritmia, *Prentice-Hall*. 1997.

Bacci, Danelutto, Pelagatti, Vaneschi, 1999. SklE: A Heterogeneous Environment for HPC Applications. *Parallel Computing 25*.

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.

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

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).

De Simone, et al. 1997. Designs Patterns for Parallel Programming. *PDPTA International Conference*.

Lavander G.R., Kafura D.G. 1995. A Polimorphic Future and First-class Function Type for Concurrent Object-Oriented Programming. *Journal of Object-Oriented Systems*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.477.7088&rep=rep1&type=pdf

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., 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., Capel M., 2014. Approach class library of high level parallel compositions to implements communication patterns using structured parallel programming. *26TH European Modeling & Simulation* Symposium.Campora Bordeaux, France.

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.

Rossainz M, Capel M., Domínguez P., 2015. Pipeline as high level parallel composition for the implementation of a sorting algorithm. *27TH European Modeling & Simulation* Symposium.Campora Bergeggi, Italy.

Wilkinson B., Allen M., 1999. Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers. *Prentice-Hall*. USA.