ISSN 1870-4069

# Partitioning Strategy Divide & Conquer as CPANs: A Methodological Proposal

Mario Rossainz[1], Manuel Capel[2], Diego Sarmiento[1]

[1]Benemérita Universidad Autónoma de Puebla,
Faculty of Computer Science, Mexico

[2]University of Granada, Software Engineering Department,
Granada, Spain

rossainz@cs.buap.mx,manuelcapel@ugr.es,diegorojas0888@gmail.com

**Abstract.** This work proposes the use of Parallel Objects using the High Level Parallel Compositions or CPAN model (Acronym in Spanish), to implement the communication patterns between processes most used in solving parallel problems. Particularly the implementation of the partitioning strategy divide and conquer as a CPAN using the object orientation paradigm is shown. A CPAN comes from the composition of a set three object types: An object manager that 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. Both Manager, collector and stages are included in the definition of a Parallel Object (PO), [6]. 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 as cheduling politics, previously determined that specifies the way in which one or more operations carried out by the instance synchronize [6], [8]. 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 more synchronization restrictions (mutual exclusion and maximum parallelism), which are now included in that definition [3]. 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. With the strategy divide and conquer as CPAN the parallel processing technique called n-Tree is used to parallelize sequential code that solve classic problems that can be partitioned by divide and conquer a n-ary tree such as sum of numbers, ordering of numbers and N-body problem. It shows the performance analysis of these implementations (speedup, cpi, etc.), comparing them with their corresponding sequential implementations to demonstrate their usefulness: programmability and performance.

**Keywords.** CPANS, parallel objects, communication patterns, divide and conquer, n-tree, structured parallel programming.

## 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. Some of this problems of parallel programming environments amount to the users acceptance, which usually depends on whether they can offer complete expressions of the parallel programs behaviour that are built with these environments [7]. At the moment in OO application systems, the scientific community interested in the study of concurrency only accepts standards for programming environments based on Parallel Objects (POs). A first approach that tries to tackle this problem is to let the programmer to develop his programs according to a sequential programming style, then, he can automatically obtain the parallelised parts of the code with the help of a specific environment.

However, intrinsic implementation difficulties exist mainly due to the difficult definition of programming languages formal semantics that refrain from the automatic (without user participation) sequential code parallelisation, and thus the problem of generating parallelism in an automatic way for a general application continues unsolved. The so called structured parallelism has become a promising approach to solve the mentioned problem. In general, parallel applications follow predetermined patterns of execution. Communication patterns are rarely arbitrary and are not structured in their logic [10]. We are interested, in particular, to do research work that has to do with parallel applications that use predetermined communication patterns, among other componentsoftware. Even so, with this promising approach, 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, [2], The necessity to have patterns or High Level Parallel Compositions, the Determination of a complete set of patterns as well as of their semantics, [7], the adoption of an object-oriented approach, [6], [9]. CPANs are parallel patterns defined and logically structured that, once identified in terms of their components and of their communication, can be adopted in the practice and be available as high level abstractions in user applications within an OO-programming environment. The process interconnection structures of most common parallel execution patterns, such as pipelines, farms and trees can be built using CPANs, within the work environment of POs that is the one used to detail the structure of a CPAN implementation.

A structured approach to parallel programming is based on the use of communication/interaction patterns (pipelines, farms, trees, etc.), which are predefined

structures of users application processes. In such a situation, the structured parallelism approach provides the interaction-pattern abstraction and describes applications through CPANs, which are able to implement the patterns mentioned already. The encapsulation of a CPAN should follow the modularity principle and it should provide a base to obtain an effective reusability of the parallel behaviour to be implemented. When there is the possibility of attaining this, a generic parallel pattern is built, which in its turn provides a possible implementation of the interaction structure between processes of the application, independently of the functionality of these. In addition, it is in line with the structured approach we have adopted that is the enrichment of traditional parallel environments with libraries of program skeletons [9] that concrete communication patterns represent. What it really means is a new design approach to parallel applications. Instead of programming a concurrent application from the beginning and controlling the creation of processes as well as the communications among them, the user simply identifies those CPANs that can implement the adapted patterns to the communication needs of his application and uses them together with the sequential code that implements the computations that individually carry out their processes. Several significant and reusable parallel patterns of interconnection can be identified in multiple applications and parallel algorithms which has resulted in a wide library of communication patterns between concurrent processes such as CPANs whose details are found in [14] and [15]. In the present work we have implemented the partitioning strategy divide and conquer using N-Tree pattern as a generic CPAN and using the object orientation paradigm we have realized its concretion in three particular applications: the add of numbers, the sorting of numbers and the solution of N-body particles, using for this the choice of three different strategies for the parallel implementation as CPANs of its sequential algorithms. In this way it is the user's own applications that specify the semantics of the N-Tree-Divide and Conquer according to the requirements of the software that was developed. Finally we show an analysis of the performance in terms of acceleration Amdhal refers to the Cpans TreeDV in solving the above problems, for a restricted range of exclusive processors in a parallel computer.

## 2 High Level Parallel Compositions (CPAN)

A CPAN comes from the composition of a set three object types: An object manager that represents the CPAN itself and makes an encapsulated abstraction out of it that hides the internal structure. The object manager controls a set of objects references, which address the object Collector and several Stage objects and represent the CPAN components whose parallel execution is coordinated by the object manager.

The objects Stage are objects of a specific purpose, in charge of encapsulating an client-server type interface that settles down between the manager and the slave-objects. These objects do not actively participate in the composition of the CPAN, but are considered external entities that contain the sequential algorithm that constitutes the solution of a given problem. Additionally, they provide the

necessary inter-connection to implement the semantics of the communication pattern which definition is sought. In other words, each stage should act a node of the graph representing the pattern that operates in parallel with the other nodes. Depending on the particular pattern that the implemented CPAN follows, any stage of it can be directly connected to the manager and/or to the other component stages.

The Collector object 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. 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 1). For details see [14].

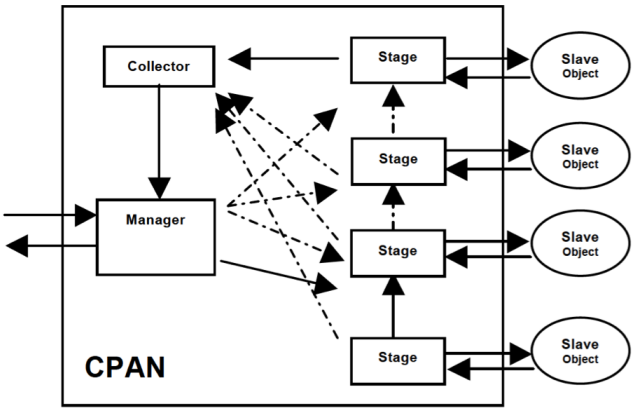


**Fig. 1.** Internal structure of CPAN. Composition of its components.

The Figure 1 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), [6]. Parallel Objects are active objects, which is equivalent to say that these objects have intrinsic execution capability, [6]. 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, [6], [8]. 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 [3]. 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 [1].
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 [1].
3. The asynchronous future will delay client activity when the method's result is reached in the client's code to evaluate an expression. For details see [11].

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 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.
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.
3. SYNC: The restriction SYNC is not more than a producer/consumer type of synchronization.

The details of the algorithms and their implementation can be seen in [14] and [15].

## 3 Construction of a CPAN

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. In PO the necessary base classes to define the manager, collector, stages objects that compose a CPAN - the implementation details are in [14] - are the next ones: Abstract class ComponentManager,

Abstract class ComponentStage and Concrete class ComponentCollector. 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 Technique of Divide and Conquer as a CPAN

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 [4]. 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 [4]. 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 [5], [12].

### 4.1 Representation of the Tree - Divide and Conquer (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 2. This parallel solution offers the prospect of traversing several parts of the tree simultaneously in the Cpan TreeDV. Once a division is made into two parts, both parts can be processed simultaneously executing the sequential algorithm contained in the slave object associated to the nodes of the tree. Though a recursive parallel solution could be formulated. One could simply assign one process o thread to each node in the tree.

### 4.2 Use and Utility of CPAN TreeDV

The potential of the CPAN TreeDV in the solution of various problems that can be solved by applying the technique of divide and conquer generating a binary tree is shown below.

**Adding a list of numbers:** A recursive definition for adding a list of numbers is:

```
int add(int *s) {
   if (number(s) <= 2) then return (n1+n2);
   else{
         divide(s, s1, s2);
         part_sum1= add(s1);
         part_sum2= add(s2);
         return (part_sum1+part_sum2);
        } }
```

In the code, $number(s)$ returns the number of numbers in the list pointed to by $s$. If there are two numbers in the list, they are called $n1$ and $n2$. If there is one number in the list, it is called $n1$ and $n2$ is zero. If there are no numbers,

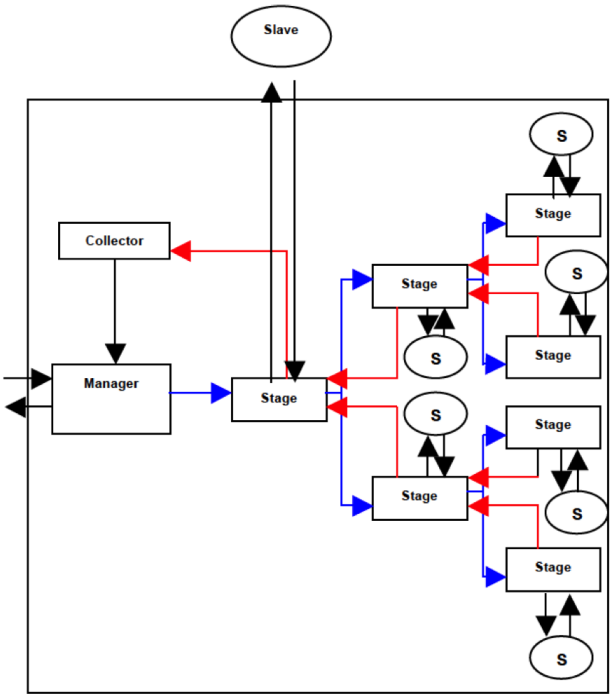


**Fig. 2.** The Cpan of a TreeDV.

both $n1$ and $n2$ are zero. Separate $if$ statements could be used for each of the cases; 0,1, or 2 numbers in the list. Each would cause termination of the recursive call, [16].

Our parallel proposal is to make use of the Cpan TreeDV. The nodes of the binary tree in the CPAN (stage processes) will be created dynamically through the execution of the proposed sequential algorithm and that is associated to the slave objects of each node in the tree. A more efficient solution adopted is to reuse stage process at each level of the tree, ie the combining act of summation of the partial sums can be done as ilustrated in figure 3. Once the partial sums have been formed, each odd-numbered stage process passes its partial sum to the adjacent even-numbered stage process, that is, $Stage1$ passes its sum to $Stage0$, $Stage3$ to $Stage2$, $Stage5$ to $Stage4$, and so on. The even-numbered stage processes then add the partial sum with its own partial sum and pass the result onward, as shown in figure 3. This continues until $Stage0$ has the final result which is passed to the Collector object of the CPAN and this in turn sends it to the Manager object that passes the result to the user.

**Quicksort sorting algorithm:** The Quicksort sorting was created by Hoare and is based on the paradigm of divide and conquer. As a first step the algorithm

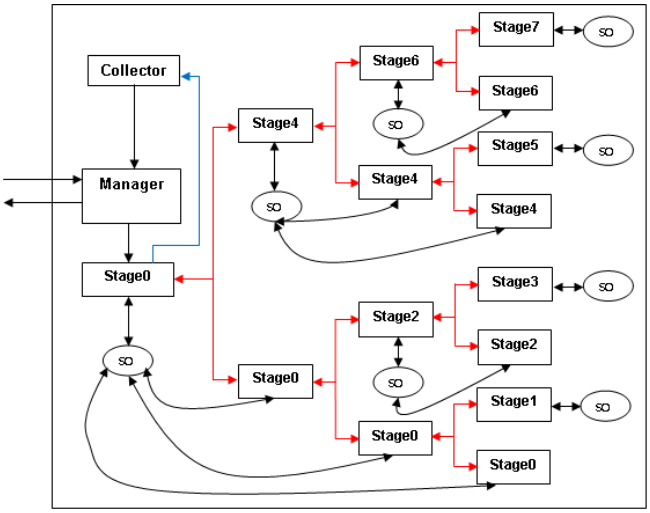


**Fig. 3.** Adding a list of numbers using CPAN TreeDV.

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 [4], 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[,..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 [4], (see figure 4) .

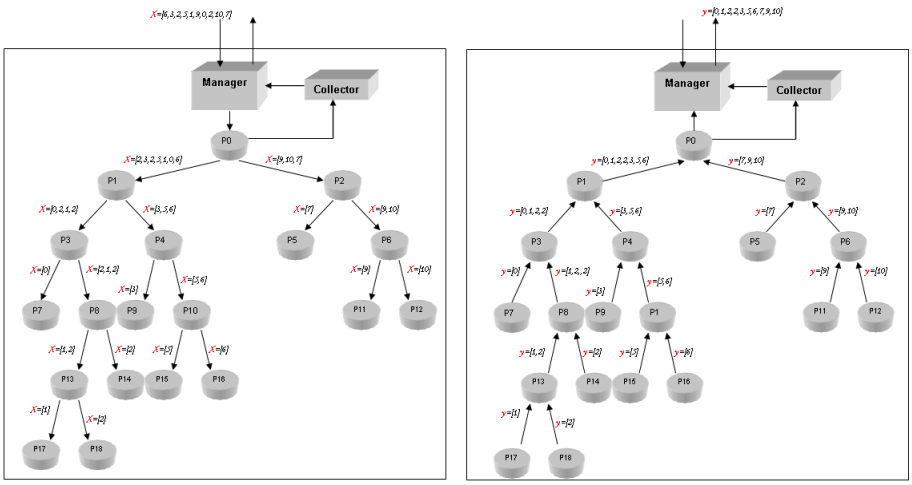


**Fig. 4.** Sequence of the QuickSort sort algorithm using CPAN TreeDV.

**N-Body Problem:** The N-Body problem is concerned with determining the effects of forces between bodies, por example, astronomical bodies that are attracted to each other through gravitational forces or charged particles are also influenced by each other according to electrostatic law. We provide the basic equations to enable the application to be coded as a CPAN TreeDV using as a case study the N-Body problem in terms of particles charged according to Coloumb's electrostatic law; particles of opposite charge are attracted and those of like charge are repelled. Also charged particles may move away from each other. The objective is to find the positions and movements of the particles

in the space that are subject to electrostatic forces from other particles using Coulomb laws.

For a computer simulation, we use values at particular times, $t_0, t_1, t_2$, etc., the time intervals being as short as posible to achieve the most accurate solution. Let the time interval be $\Delta t$. Then, for a particular particle of mass $m$, the force is given by:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t},$$

and a new velocity by:

$$v^{t+1} = v^t + \frac{F\Delta t}{m},$$

where $v^{t+1}$ is the velocity of the particle at time $t + 1$ and $v^t$ is the velocity of the particle at time $t$. If a particle is moving at a velocity $v$ over the time interval $\Delta t$, its position changes by:

$$x^{t+1} - x^t = v\Delta t,$$

where $x^t$ is position at time $t$. Once particles move to new positions, the forces change and the computation has to be repeated. The computation of the attraction or not of N-particles according to their electrostatic charge is described in the following algorithm:

```
for (t=0; t< tmax; t++) {
   for(i=0;i<N;i++)
     {
      F=force(i);
      v[i]_new = v[i]+F*dt/m;
      x[i]_new = x[i]+v[i]_new*dt;
     }}

for(i=0;i<nmax;i++) {
   x[i]=x[i]_new;
   v[i]=v[i]_new;
  }
```

For each time period $t$, for each particle $i$, compute force on ith particle, compute new velocity and new position. For each particle $i$ update velocity and position.

Parallelizing this algorithm can use partitioning where by groups of particles are the responsability of each process, and each force is carrried in distint messages between process. A large number of messages could result and it is not feasible if N is very large, [16]. The complexity can be reduced using the technique that a cluster of distant particles can be approximated as a single distant particle of the total mass of the cluster sited at the center of mass of the cluster.

This idea can be implemented as a CPAN by being applied recursively generating a m-ary tree, in particular way, a $quad-tree$ (a tree in which each node of tree has four children) based on the Barnes-Hut algorithm, as you can see in [13] and [16].

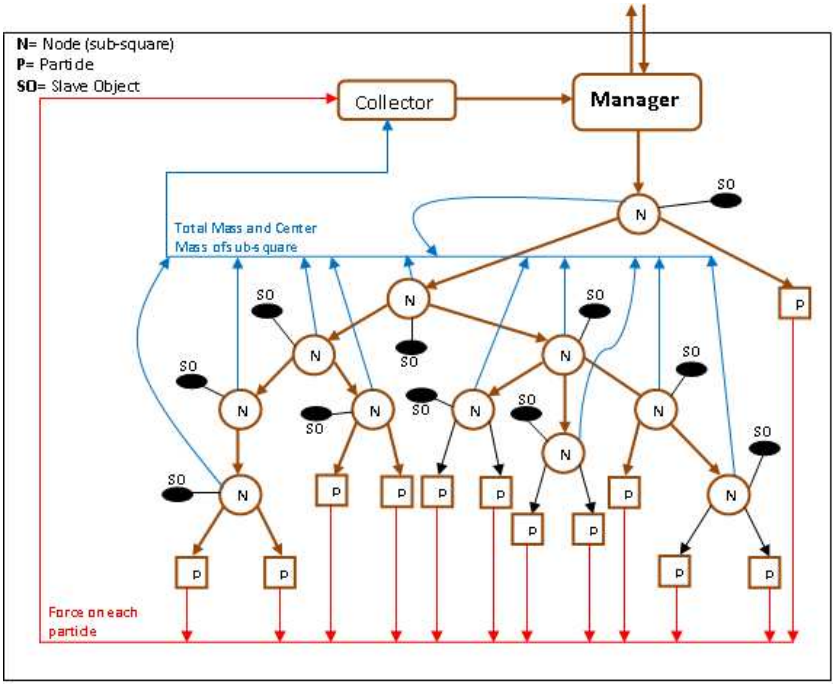


**Fig. 5.** Cpan QuadTree Particle.

A divide and Conquer formation to the problem using this clustering idea start for a two-dimensional space in which one square contains the particles. This square is recursively divided into four sub-squares creating a *quadtree*, ie a tree with up to four edges from each edge. If a sub-square contains no particles, the sub-square is deleted from futher considerarion. If a sub-square contains more than one particle, it is recursively divided until every sub-square contains one particle creates the *quadtree*. The tree will be unbalanced. The leaves represent cells each containing one particle.

The Figure 5 represents the resultant quadtree like a CPAN. In the "Cpan QuadTree Particle" of figure 5, the total mass and center of mass of the subsquare is stored at each node of tree. The force on each particle can be obtained by traversing the tree starting at the root, stopping at a node when the clustering approximation can be used for the particular particle, and otherwise continuing to traverse the tree downward.

## 5 Performance

Performance analysis of CPANS TreeDV for adding a list of numbers, sorting a list of numbers using Quicksort algorithm and the N-Body Problem 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. The CPAN TreeDV performance to solve the problems mentioned 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 CPANs TreeDV using Divide and Conquer Technique is carried out with the following restrictions execution:

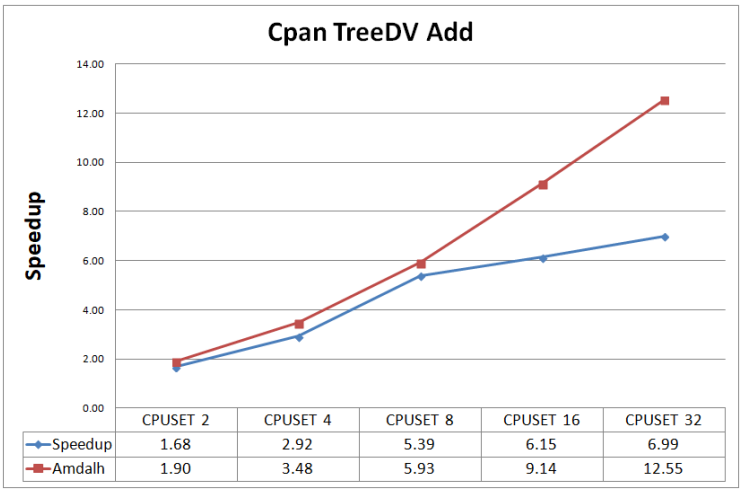


**Fig. 6.** Speedup scalability found for CpanTreeDV in solution of adding numbers problem for 2, 4, 8, 16 and 32 exclusive processors.

- In the adding numbers problem, the same sequential sum algorithm was used in each of the slave objects associated with the stages (nodes) of binary tree that is generated in the Cpan TreeDV (see figure 3).
- In the sorting numbers problem, parallel implementation of sequential sorting algorithm based on a CPAN TreeDV is Quicksort sorting algorithm based on a binary tree (see figure 4).
- In both cases, both in the adding numbers problem and in the sorting numbers problem, 50000 random integers were generated; cach number generated in a range between 0 and 50000, allowing make a sufficient charge for processors and thereby observe the performance improvement CPAN TreeDV.
- In the N-body problem, we work with a simulation of 50000 particles with electrostatic charge moving randomly in space. The calculations were: find

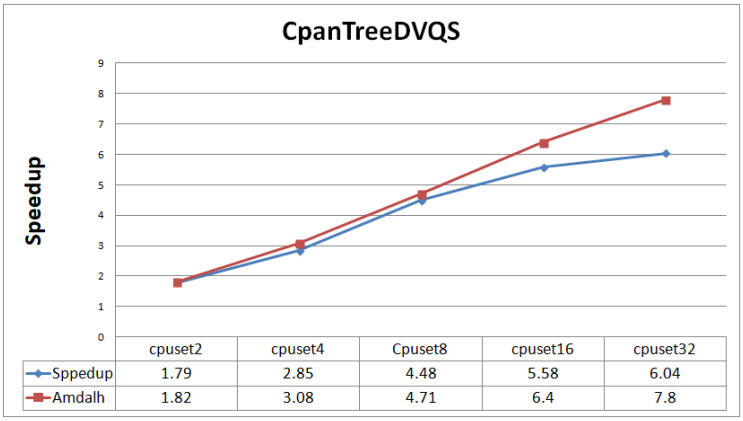


**Fig. 7.** Speedup scalability found for CpanTreeDVQS in solution of sorting numbers problem for 2, 4, 8, 16 and 32 exclusive processors.

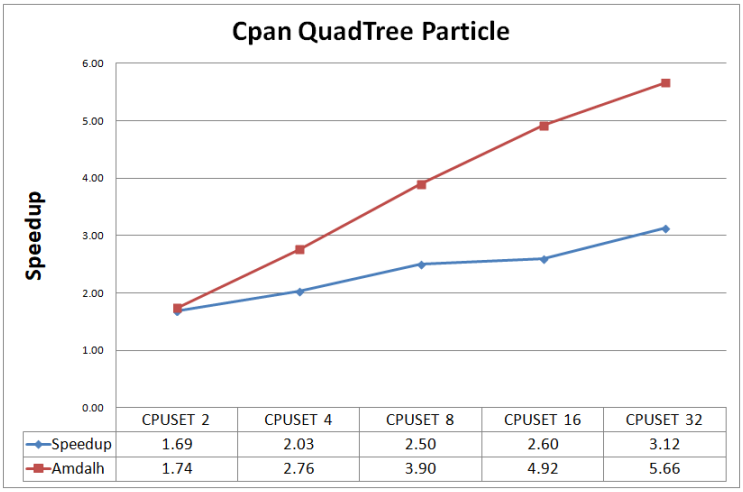


**Fig. 8.** Speedup scalability found for CpanQuadTree in solution of N-body problem for 2, 4, 8, 16 and 32 exclusive processors.

the positions and movements of the particles in the space that are subject to electrostatic forces from other particles using Coulomb laws. For this, the CPAN QuadTree calculated with the sequential algorithms associated with the slave objects of the generated M-tree, the masses and the forces of each particle (see figure 5).

These execution conditions allow a sufficient load for the processors and show the good performance of the Cpan TreeDV when solving them. For all of them the execution was performed in 2, 4, 8, 16 and 32 exclusive processors and the

results are shown in the figure 6, figure 7 and figure 8. In them show the series of measurements obtained including their corresponding sequential versions for Cpans TreeDV, TreeDVQS and QuadTree, magnitude speedup found and the upper bound on the magnitude of speedup using for that Amdahl's law, moreover the runtime execution in seconds of the programs.

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.

## 6 Conclusions

We have presented a method for design of concurrent applications based on the construction of High Level Parallel Compositions or CPANS and which are usually used in different platforms, such as C ++ and POSIX Threads. We discuss the implementation of CPANs treeDV as generic and reusable patterns of communication/interaction between processes which implements the algorithm design technique called divide and conquer making use of an N-tree as a pattern of communication associated, which can even be used by inexperienced parallel application programmers to obtain efficient code by only programming the sequential parts of their applications.The CPAN TreeDV has been reused in the communication/interaction between the processes of three solved problems with different implementation strategies of their respective sequential algorithms of solution: the adding numbers problem, the sorting numbers problem and de N-body (particles) problem. This selected problems have been included to show speedup and low execution times about their best sequential version of the algorithms that solve these problems. We have also obtained good performance in their executions and speedup scalability compared to Amdahls law on the number of processors used to obtain the solution.

## References

1. Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley (2000)
2. Bacci, B., Danelutto, M., Pelagattii, S., Vaneschi, M.: SklE: A Heterogeneous Environment for HPC Applications Parallel Computing. Springer, Vol. 25, No. 13–14 (1999)
3. Birrell, B.: An Introduction to Programming with Threads. Digital Equipment Corporation, Systems Research Center, Palo Alto California, USA (1989)
4. Brassard, G., Bratley, P.: Fundamentals of Algorithmics. Prentice-Hall (1997)

5. Brinch, H.: Model Programs for Computational Science. A programming methodology for multicomputers, Concurrency, Practice and Experience, Volume 5, Number 5 (1993)
6. Corradi, A., Leonardi, L.: PO constraints as tools to synchronize active objects. Journal Object Oriented Programming, Vol. 4, No. 6, pp.41–53 (1991)
7. Corradi, A., Leonardo, L., Zambonelli, F.: Experiences toward an Object-Oriented Approach to Structured Parallel Programming. DEIS Technical Report No. DEISLIA-95-007 (1995)
8. Danelutto, M., Torquati, M: Loop parallelism: A new skeleton perspective on data parallel patterns. In Proceedings of Intl. Euromicro PDP: Parallel Distributed and Network-based Processing, Torino, Italy (2014)
9. Darlington, J.: Parallel programming using skeleton functions. In Proceedings PARLE93, Munich (1993)
10. Hansen, B.: Model programs for computational science: A programming methodology for multicomputers. Concurrency Practice and Experience, Vol. 5, No. 5 (1993)
11. Lavander, G., Kafura, D.: A Polymorphic Future and First-class Function Type for Concurrent Object-Oriented Programming in C++. Journal of Object-Oriented Systems (1995)
12. Liwu, L.: Java Data Structures and Programming. Springer Verlag, Germany (2002)
13. Roosta, S.: Parallel Processing and Parallel Algorithms. Theory and Computation, Springer (1999)
14. Rossainz, M., Capel, M.: A Parallel Programming Methodology using Communication Patterns named CPANS or Composition of Parallel Object. In Proceedings of 20TH European Modeling & Simulation Symposium, Campora S. Giovanni, Italy (2008)
15. Rossainz, M., Capel, M.: Design and implementation of communication patterns using parallel objects. International Journal of Simulation and Process Modelling, Volume 12, No. 1, Pp: 69–91 (2017)
16. Wilkinson, B., Allen, M.: Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice-Hall, USA (1999)