33rd European Modeling & Simulation Symposium
18th International Multidisciplinary Modeling & Simulation Multiconference

ISSN 2724-0029 ISBN 978-88-85741-57-7 © 2021 The Authors.
doi: 10.46354/i3m.2021.emss.004

Parallel Design Patterns vs Parallel Object Compositions. Two Proposals for Parallelization of the Divide & Conquer Technique

Mario Rossainz-López[1, *], Ivo Pineda-Torres[1], Bárbara Sánchez-Rinza[1], Manuel Capel-Tuñón[2]

[1] Faculty of Computer Science, Autonomous University of Puebla, Av. San Claudio and 14 Sur Street, San Manuel, Puebla, Mexico, C.P. 72570
[2]Software Engineering Department, College of Informatics and Telecommunications ETSIIT, University of Granada, Daniel Saucedo Aranda s/n, Granada 18071, Spain

*Corresponding author. Email address: mrossainzl@gmail.com

Abstract

The present work shows the parallelization of the algorithmic design technique Divide & Conquer in two different ways: As a Parallel Design Pattern (PDP) through Active Objects and as Composition High-Level Parallel (HLPC). The overall purpose is to provide the user and novice programmer with two approaches within the object-oriented programming environment, particularly within the programming of Parallel Objects (PO), so that they can develop their programs according to a sequential programming style, automatically obtaining, easy and without much effort, the parallel counterpart of your code with the help of a specific programming environment like the one proposed. It is common for parallel applications to follow predetermined patterns in communication between processes. That is why this proposal proposes two different methods that solve problems with the same parallel control structure. Both methods use Structured Parallel Programming and Parallel Objects. The proposal is specialized in the algorithmic technique of divide & conquer to solve ordering, search, and optimization problems. The default pattern used to communicate problem solving processes is the tree structure. The proposed methods are novel because they offer the programmer the communication pattern between tree-like processes that is already defined in its structure. The programmer is only concerned with implementing the sequential algorithms that solve the problem under the divide & conquer paradigm. Both approaches are effective because they show good speedup analysis, and their usefulness, programmability, and performance are demonstrated.

Keywords: Parallel Objects, Structured Parallel Programming, Divide and Conquer, PDP, HLPC

1. Introduction

The construction of parallel and concurrent systems has as part of its objectives to obtain efficiency in the data processing. The use of this type of system is not limited to Computer Science; furthermore, such systems have spread to a variety of areas of different disciplines. To increase the performance of certain systems, parallel processing is an alternative to sequential processing. This work focuses on proposing improvements in the design of parallel algorithms as well as proposing parallel programming methods and models on different types of architectures. We focus on the design and

implementation of two models that allow the programming of parallel applications under the paradigm of Object Orientation, Structured Parallel Programming, and the concept of Parallel Object (McCool, Robison and Reinders, 2012), to solve problems whose algorithms are capable of being parallelized according to the two proposed models and achieve a good degree of performance:

- The model of Parallel Design Patterns or PDPs
- The Model of High-Level Parallel Compositions or HLPC

In the first model, a Parallel Design Pattern (PDP) is defined as a class of algorithms that solve different problems with the same control structure (Collins, 2011). In other words, a PDP is a Generic Parallel Program that describes the common control structure shared by all the algorithms that solve a problem and correspond to the same pattern (Collins, 2011). Patterns are different algorithmic design techniques that exist for the implementation of algorithms and heuristics that solve problems: Backtrack, Voracious Algorithms, Branching and Bounding, Divide and Conquer, Dynamic Programming, Total Pairs, etc. Neither the data types nor the code of specific data-dependent procedures needs to be detailed at this level, as these depend on specific problems. The core of this proposal is that the PDP has a parallel component that implements the generality of the chosen algorithmic design technique and a sequential component for a specific application that is solved with a such technique (Collins, 2011; Ernsting and Kuchen, 2012).

In the second model, a High-Level Parallel Composition or HLPC is defined as the representation of a communication pattern between processes of an application or parallel algorithm through 3 types of Objects: a manager object, which controls the references of some stage objects, and a collector object, which act collaboratively for each request made by the HLPC client objects (Brinch Hansen, 1993). Also, for each stage of the HLPC, a third slave object will oversee implementing the sequential part of the computing system that is intended to be carried out. The most common process communication patterns are pipelines or channels, farms or farms, trees or trees, mesh or meshes, cubes, and hypercubes, among others.

In this work we analyze the algorithmic technique of Divide & Conquer to be implemented through the two models previously described to solve the ordering problem through the Quicksort algorithm using a binary tree as a structure for its solution. We find, then, similarities in the models. In both, a parallel program is proposed, this defines a specific structure in a generic way to be adapted to any problem that is solved with the same algorithmic technique. In the case of PDP, the divide-and-conquer technique itself uses a binary tree as the communication structure of the generated processes. In the case of HLPC, the Binary Tree communication pattern is based on the development of the divide and conquer technique. Both solve the same ordering problem through the Quicksort algorithm and are an alternative for the novice programmer so that the greatest effort is focused on the sequential implementation of the algorithm that solves the problem and uses the proposed models to create the parallel structure of the appropriate algorithmic technique in a semi-automatic way. The procedure for creating the PDP and HLPC Divide & Conquer is explained as follows, as well as the performance analysis of both separately, and a comparison of their accelerations and execution times in a parallel machine with a particular architecture.

2. Literature Review

The transformation of existing sequential applications into parallel ones for multiprocessors environments has been of great interest for decades. There is not however a solution of general application to solve the still pending issues regarding a sound parallelization of algorithms and programs. In (Collins, 2011), the effectiveness and applicability of automatic techniques has been explored. Six implementation parameters in the FastFlow parallel skeleton framework were tuned to obtain speed up of calculations. FastFlow is a C++ parallel programming framework intended to propitiate high-level, pattern-based parallel programming, as the research work of (Torquati, Aldinucci and Danelutto, 2014) pointed out. There are currently projects that develop frameworks and offer to users constructs, templates and parallel communication patterns between processes, such as the ParaPhrase project, (Torquati, Aldinucci and Danelutto, 2015).

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

MALLBA (Alba et-al, 2007) is another software tool intended for assisting in the solution of combinatorial optimization problems using generic algorithmic skeletons implemented in C++.

Some environments of parallel programming, as the one called SklECL (Steuwer et-al, 2011), are based on skeletons and wrappers that make up the fundamental constructs of a coordination language, defining modules that encapsulate code written in a sequential language and three classes of skeletons: control, stream parallel, and parallel data.

Examples of commonly used skeletons are farms, i.e., a selection of workers processes that carry out a set of

computation tasks; pipelines that are used to exploit the derived parallelism extracted from executing the different phases of a calculation simultaneously; and trees to which parallel divide-and-conquer techniques can be applied. Several parallel programming libraries and environments provide these skeletons. Regarding the latter ones, the advantage of SklE amounts to allows composing the skeletons freely, and building more complex structures, and it is also able to generate optimized code for specific architectures. The development of parallel applications in SklE is carried out through VisualSklE, which is a graphic windows system (Steuwer et-al, 2011).

3. Structured Parallel Programming

SSP is based on the use of predefined communication/interaction patterns (pipelines, farms, trees, etc.) between the processes of a user application (Danish and Farooqui, 2013). This approach starts from the abstraction of the interaction pattern that allows the design of applications capable of using and particularizing it to solve a specific problem. The encapsulation of a communication pattern between processes must follow the principle of modularity and must provide a basis for obtaining the effective reusability of the parallel behavior of the software entity that this implements. Once this is achieved, a generic parallel pattern is created to provides a possible representation of the interaction between processes which is independent of their functionality. The approach presented in this work proposes a programming environment for both the PDP and the HLPC through program libraries (Darlington et al, 1993 and De Simone, et al, 199) which, in this case, represents the Tree communication pattern (tree of processes) in the parallelization of the algorithmic technique of Divide & Conquer. The contribution of this scheme is that, instead of programming a parallel application from the beginning, it is necessary only to identify the appropriate inter-process communication pattern for the parallelization of the problem, or to identify the Parallel Design Pattern that uses such pattern. For this work, we specify the proposal in the PDP-Divide & Conquer and the HLPC-Divide & Conquer and we use them to solve ordering problems. However, the identification and unambiguous definition of a complete set of communication patterns between processes of a parallel application are still far from being a solved problem, since there is not an agreement general enough that allows defining in a formal way their semantics (Corradi and Zambonelli, 1995). What this research shows is the definition and use of a PDP and a HLPC, both generic and adapted through the inheritance, composition, and/or aggregation mechanisms of Object Orientation, to the specific needs of each application. In this way, it is the user applications themselves which finally specify the semantics of the patterns and compositions according to the requirements of the software to be developed.

4. Parallel Objects (PO)

Parallel Objects are active object. that is, objects that can execute themselves. The applications within the PO model can exploit both the parallelism between objects (inter-object) and their internal parallelism (intra-object) (Corradi and Leonardi, 1991). A PO object has a structure like any object in any programming language oriented to objects. It also includes a scheduling policy determined a priori that specifies how to synchronize one or more operations of the object type that can be invoked in parallel (Theelen, Florescu, et al., 2007). Synchronization policies are expressed in terms of restrictions; for example, a restriction to ensure mutual exclusion between reader/writer processes can be specified, the maximum number of reader processes that will run in parallel, or simply the necessary synchronization between processes that access shared resources. All parallel objects are then derived from the classical definition of "class" and the incorporation of the process planning policy (synchronization restrictions, mutual exclusion, and maximum parallelism). Objects of the same class share the same specification of the behavior in it, from which they are instantiated. Parallel objects support multiple inheritance, which allows a new complete PO specification to be derived from an existing one (Corradi and Leonardi, 1991; Theelen, Florescu, et al., 2007). When there are parallel requests for service in a PO, it is necessary to have synchronization mechanisms so that they can concurrently manage several executions flows, and at the same time, the consistency of the data that is being processed can be garantee. To achieve this, within any PO, the following restrictions can be used:

- MaxPar: The maximum parallelism or MaxPar is the maximum number of processes that can be executed at the same time. The MAXPAR applied to a function represents the maximum number of processes that can execute that function concurrently.
- MutEx: The restriction of synchronization mutex carries out a mutual exclusion among processes that are needed to have access to a shared object. The mutex preserves critical sections of code and obtains exclusive access to the resources.
- Sync: Synchronization of producer/consumer type is used to program the methods or functions of the POs with the the processes that use them are synchronized so that a process can execute a method as long as other process confirms that this can be carried out before that the latte finishes using a shared resource; otherwise, the process will be blocked until the notification of the next execution of the other process is notified.

In addition, all PO provides different types of communication:

- The synchronous communication mode stops the client activity until it receives the answer of its request from the active server object.
- The asynchronous communication does not delay the client activity. The client simply sends the request to the active object server and the execution continues afterwards. Its use in application programming is also easy, it is only necessary to create a process and start it to carry out the communication independently from the client.
- The asynchronous future mode delays the client's 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, it requires of a special care to get a syntactical construct with the correct required semantics (Lavander and Kafura, 2010).

5. Parallel Design Patterns (PDP)

A Parallel Design Pattern or PDP is defined as a class of algorithms that solve different problems and that have the same control structure. Examples of this are the PDPs shown in table 1.

Table 1. Parallel paradigms and their model programs

PDP	Model Programs	Communication Pattern
Total Pairs	1. Householder 2. N-Body	Pipeline
Tuples Multiplication	1. Product-Matrices 2. Paths-Graphs	Pipeline
Divide and Conquer	1. Sort 2. Search	Tree
Cellular Automata	1. Laplace 2. Simulation	Matrix

A Generic Program is created for each PDP that defines the common control structure for those problems that can be solved with the same algorithmic design technique. The Generic Program is commonly referred to as the Algorithmic Skeleton (Ernsting and Kuchen, 2012).

Subsequently, from a general parallel program, two or more Model Programs are derived which show the use of the PDP to solve specific problems. A Generic Program includes some types of data that are not specified as well as procedures that vary from one application to another.

A Model Program is obtained by replacing these types of data and procedures with the corresponding types of data and procedures of a sequential program that solves a specific problem. In other words, the essence of this proposal is that a model program has a parallel component that implements a PDP and a sequential component for a specific application (Figure 1).

5.1. Derivation of a PDP

1. Identify one, two, or more computational problems with the same control structure.
2. For the problem (s) identified, write a tutorial that explains your computational theory and includes a complete program.
3. Write a parallel program for PDP programming.
4. Test the parallel program on a sequential computer.
5. Derive a parallel program for the problem(s) to be solved by substituting data types, variables, procedures, etc., and analyze the complexity of the programs.
6. Rewrite the parallel programs in an implementation language and measure their performance on a multicomputer.
7. Write clear descriptions of parallel programs.
8. Publish the programs and their descriptions in their entirety.

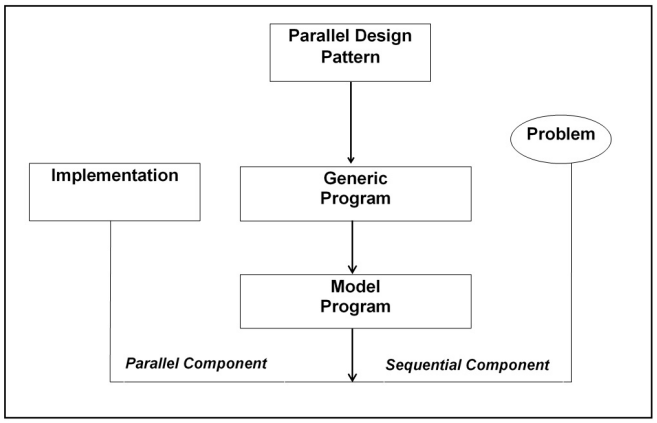


Figure 1. Abstract Model of a Parallel Design Pattern (PDP)

6. High Level Parallel Composition

A HLPC comes from the composition of a set three object types: an object manager that represents the HLPC itself and makes an encapsulated abstraction out of it that hides the internal structure. The object manager controls a set of objects references, which addresses the object collector and several stage objects and represent the HLPC components whose parallel execution is coordinated by the object manager (see Figure 2), (Danelutto and Torquati, 2014).

The objects stage are objects of a specific purpose, in charge of encapsulating a 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 HLPC but are considered external entities that contain the sequential algorithm that constitutes the solution of a given problem. Additionally, they provide the necessary interconnection to implement the semantics of the communication pattern in which definition is sought. In other words, each stage should

act as a node of the graph representing the pattern that operates in parallel with the other nodes. Depending on the pattern that the implemented HLPC follows, any stage can be directly connected to the manager and/or to the other component stages. In a 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 HLPC composition. During a service request, the control flow within the stages of a HLPC depends on the implemented communication pattern. When the composition finishes its execution, the result does not return to the manager directly. They return to an instance of the collector class that is in charge of storing these results and send them to the manager, which will finally send the results to the environment, and to a collector object as soon as they arrive. It is not necessary to wait for all the results that are being obtained. For implementation details see (Rossainz and Capel, 2014; Rossainz and Capel, 2017).

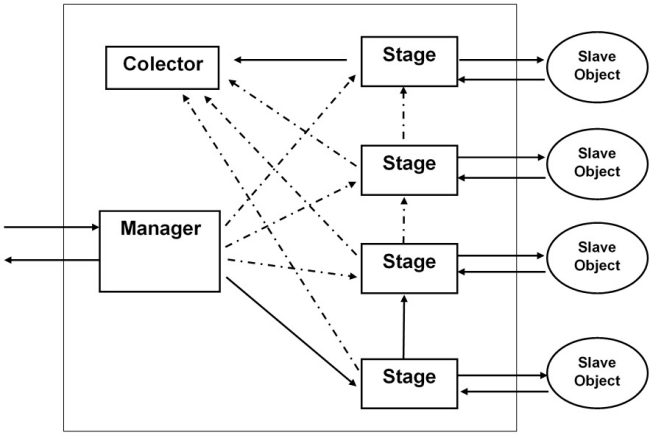


Figure 2. Abstract model of an HPLC

6.1. Derivation of a HLPC

1. An instance of the manager class is created, that is, one that implements the required parallel behavior according to the following steps:
   1.1. Initialize the instance with the reference to the slave objects that will be controlled by each stage and the solution algorithm associated with the slave object.
   1.2. The internal stages are created and an association "slave object-solution algorithm" is passed to each one, which will be executed by each stage.
2. The user asks the manager to start a calculation through the execution of the HLPC that is carried out as follows:
   2.1. The collector object referring to the request is created.
   2.2. The input data (without type checking) and the reference to the collector are transferred to the stages.
   2.3. The results are obtained from the collector object.
   2.4. The collector returns the results to the outside, again without type checking.
3. A manager object that represents the HLPC has been created and initialized and execution requests can be dispatched in parallel.

7. Divide & Conquer

The Divide & Conquer technique is characterized by dividing a problem into subproblems that have the same characteristics as the whole problem. The division of the problem into smaller subproblems is carried out using recursion. The recursive method continues dividing the problem until the divided parts can no longer be divided, then the partial results of each subproblem are progressively combined in ascending order until the solution to the initial problem is obtained (Brassard and Bratley, 1997). In this technique, the division of each problem is often done into two subproblems; therefore, we can assume a recursive formulation of the Divide and Conquer method with a division scheme in the form of a binary tree, whose nodes will be processes.

The root node of the tree receives as input a complete problem that is divided into two parts, one is sent to the left child node, the other is sent to the node representing the right child (Figure 3). The division process is repeated recursively until reaching the lowest levels of the tree. After certain time, all leaf nodes receive a subproblem as input from their parent node. Then, they solve the problem and return the solutions. Any parent node in the tree will get two partial solutions from its child nodes and combine them to provide a single solution that will be the output of the parent node. Finally, the root node will provide the complete solution of the initial problem (Brassard and Bratley, 1997). Figure 3 shows a complete binary tree, which is a perfectly balanced tree with leaf nodes at the same level, however, one or more leaf nodes could appear at different levels of the tree if the number of subproblems is not a power of two.

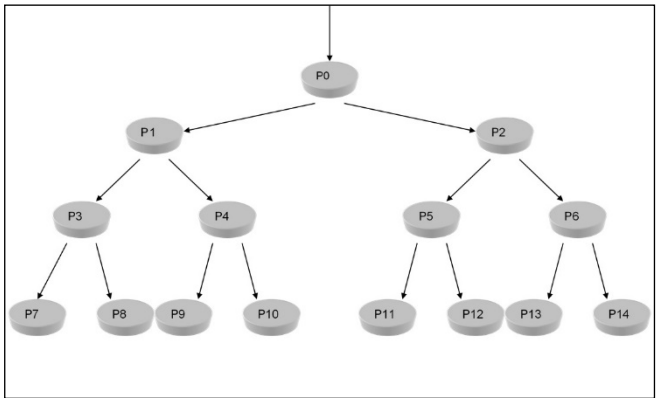


Figure 3. Model of a Binary Tree

8. Quicksort

The quicksort algorithm created by Hoare is based on the divide & conquer paradigm (Brassard and Bratley, 1997). As a first step, the algorithm selects one of the elements of the data set to be ordered as a pivot. The assembly is then split into both sides of the pivot. The elements are moved in such a way that those that are greater than the pivot is to the right, while those that are less are to the left. Subsequently, the parts of the set that remain on both sides of the pivot are ordered in a parallel, recursive, and independent manner. The result is a completely ordered set.

9. Quicksort parallelization using PDP

Figure 4 shows the graphical model of the Parallel Design Pattern (PDP) that is developed to implement the Divide & Conquer algorithmic design technique and that represents the parallel component used to solve the sorting problem with Quicksort. The PDP is made up of a binary tree whose root node has as input a complete problem (the user's sequential component). Both the problem (unordered data) and its solution (ordered-data) are defined by an array of n-elements of the same type. The type of elements and procedures for the division of the problem and combination of solutions is part of the parallel algorithm that depends on the nature of the specific program or model program (which in this case is the Divide & Conquer technique). This is the main feature that makes the PDP a device to solve specific problems in parallel in a simple way (Quicksort algorithm) (Roosta and Séller, 1999).

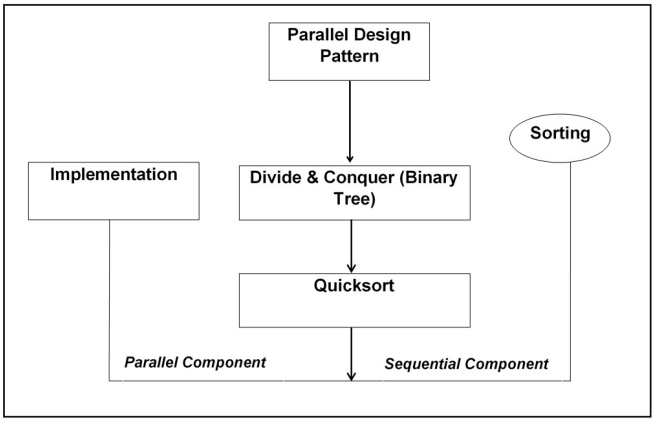


Figure 4. Divide & Conquer PDP Model for the Quicksort Sorting Algorithm

The usefulness of the proposal presented here is that different sequential problems such as binary searches or summation of numbers, to name a few, are solved using the same parallel component, the divide & conquer technique designed as PDP. The implementation of the Divide & Conquer PDP is made up of the components shown in the diagram in Figure 5.

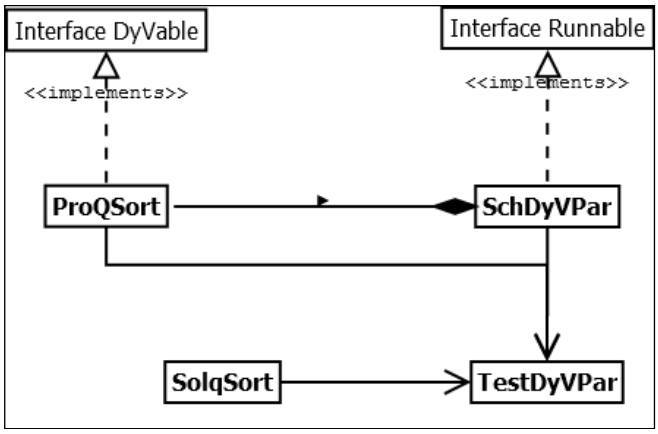
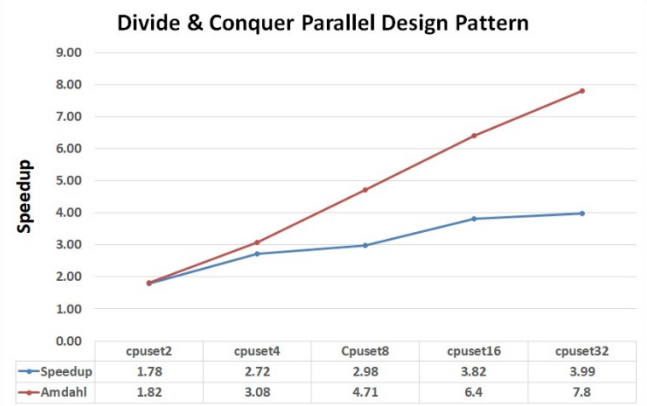


Figure 5. PDP Divide and Conquer Class Diagram

- The DyVable Interface: All problems that are solved through this PDP must implement this interface. In doing so, it Is possible to guarantee that these problems are compatible with it and, therefore, its abstract methods can be implemented. In this way, the class that implements the paradigm called SchDyVPar will be able to solve the problems generically. The SchDyVPar class then receives references to objects compatible with the interface and can, therefore, invoke its abstract methods:
  `public boolean base ()`: Returns a value TRUE if the object data represents a base or indivisible problem, FALSE otherwise
  `public Object solve ()`: Returns a solution to a base subproblem.
  `public Interface [] divide ()`: Divide a non-base problem into a vector of sub-problems.
  `public Object combine (Object [])`: Receives a vector of subproblem solution objects, combines them, and returns a solution to the problem.
- The SchDyVPar class: It is used to create active objects that implement the divide and conquer technique in parallel, using a process tree with the original problem as the root and the subproblems as nodes and leaves of said tree. The implementation is independent of the specific problem to be solved.
- The ProQSort class: Creates instances of the problem to be solved using a sequential algorithm (quicksort). This class must implement the interface.
- The SolqSort class: Provides instances that contain a solution to the ordering problem.
- The TestDyVPar class: It is the main program in this class a vector of elements is created to later obtain an instance of the ProQSort class. Then, a process is launched with the initial problem to solve it with an instance of the SchDyVPar class, whose parameter will be the object containing the original problem, and later it is expected to receive a solution object of type SolqSort

9.1. PDP-Quicksort Performance Analysis

This section shows the performance obtained from the execution of the Quicksort algorithm such as PDP-Divide & Conquer (Figure 6), which was carried out on a parallel computer with 32 processors, 8 GB of main memory, high-speed buses, and distributed in shared memory architecture. The input data were 50,000 random integers in a range between 0 and 50,000 to provide the processors of the machine with enough load so that the improvement in the performance of the PDP concerning its acceleration (speedup) and Amdahl's law could be observed.



	cpuset2	cpuset4	Cpuset8	cpuset16	cpuset32
Speedup	1.78	2.72	2.98	3.82	3.99
Amdahl	1.82	3.08	4.71	6.4	7.8

Figure 6. Scalability of the speedup found for the PDP Divide & Conquer that Quicksort implements in solving the number sorting problem for 2,4, 8, 16, and 32 processors

10. Quicksort parallelization using HLPC

The representation of the High-Level Parallel Composition or HLPC that defines the Divide & Conquer technique is shown in Figure 7. This parallel proposal offers the perspective of executing several parts of the tree simultaneously. The root node of the tree receives as input a complete problem that is divided into two parts, which are processed simultaneously by executing the sequential algorithm contained in the associated slave object in each node of the tree that is built. The division process is recursively repeated until reaching the lowest levels of the tree and until all leaf nodes receive as input a subproblem from their parent node, then they solve the problem and return the solutions simultaneously (Wilkinson and Allen, 1999). Any parent node in the tree will get two partial solutions from its child nodes in parallel and combine them to provide a single solution as its output. Finally, the root node will provide to the Collector object of the HLPC the complete solution of the initial problem, and this in turn, will send it to the Manager process to be delivered to the user.

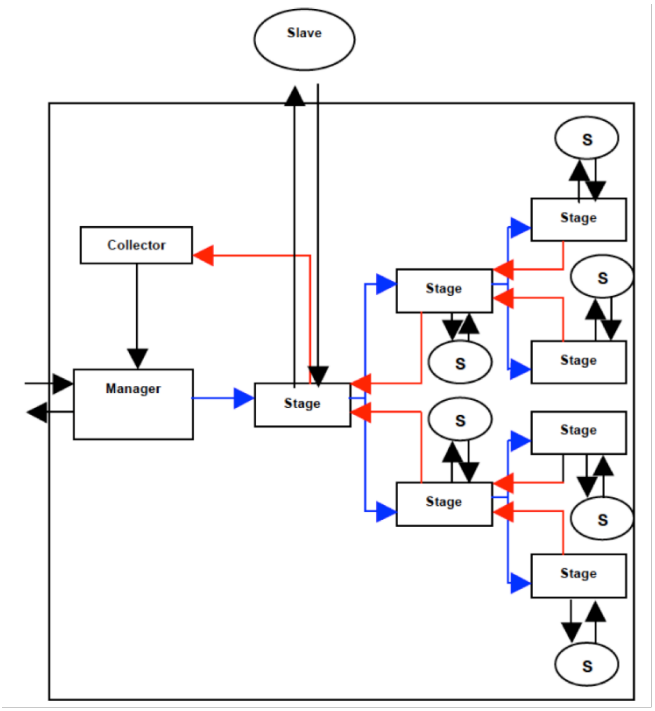


Figure 7. HLPC Divide & Conquer Abstract Model

By parallelizing the Quicksort algorithm using the generic HLPC of Figure 7, we obtain concrete and HLPC that solves the number ordering problem shown in Figure 8. In this last model, the input data provided by the user through the manager object flows from the root, which will be a dynamically created stage object to which a slave object is associated to have the user's sequential algorithm that solves the problem and repeats itself. This procedure of creation of tree nodes is done until reaching the stage leaves and vice versa in the backtracking of the inherent recursion. Peer nodes (stages) run in parallel. The initial or root stage of the HLPC will obtain the final solution, which is the completely ordered data set, and will be sent to the Collector object, which in turn will deliver such solution to the Manager for completion and delivery to the user. In the model in Figure 7, only one slave object is statically predefined and associated with the first stage of the tree. The following slave objects will be created internally by the stages themselves dynamically since the tree levels depend on the problem to be solved and not to the number of nodes that the tree may have is not known a priori, nor its depth level.

10.1. HLPC-Quicksort Performance Analysis

The proposed HLPC performance analysis is shown by sorting a list of integers using the Quicksort algorithm (Figure 9). At least, for this problem, the performance obtained is "good" according to the HLPC model. As in the PDP model, 50,000 random integers were generated, each number generated in a range between 0 and 50,000, which allowed obtaining a sufficient load for the processors of the machine on which the proposed implementation was executed,

and which was the same for the case of PDP. Therefore, observe the improvement of the performance of the HLPC was observed in the same way they are measuring the speedup and Amdahl's Law was measured

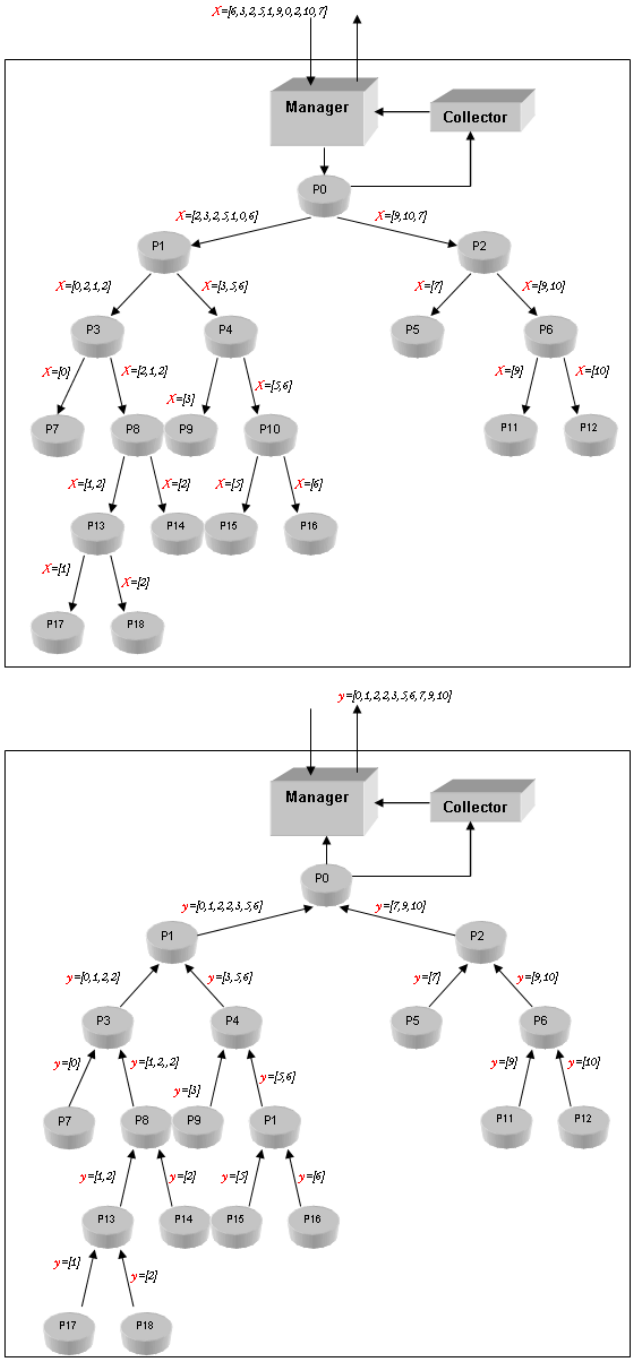


Figure 8. Quicksort sort algorithm sequence using the HLPC Divide & Conquer

11. PDP vs HLPC performance comparison

The execution of the two models proposed in this work: PDP and HLPC Divide & Conquer were carried out in 2, 4, 8, 16, and 32 exclusive processors, whose scalability was shown in Figure 6 and Figure 9, respectively. Figure 10 shows a comparison of the accelerations found and the upper bound or Amdahl's Law of the magnitude of these accelerations, which, as observed, is the same for both proposals. The acceleration obtained when executing the HLPC-Divide & Conquer is better than that found in the PDP-Divide & Conquer.

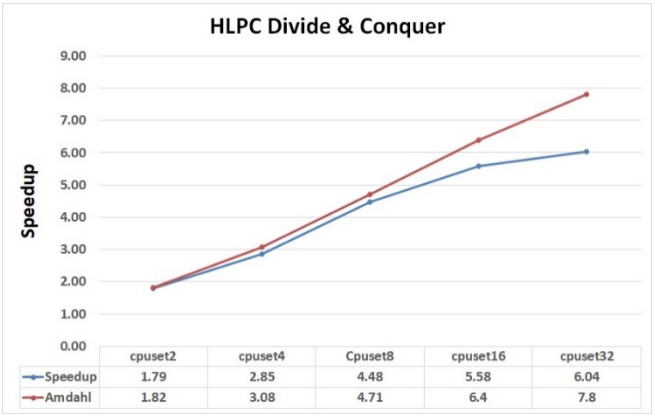


Figure 9. Scalability of the speedup found for the HLPC Divide & Conquer that Quicksort implements in solving the number sorting problem for 2,4, 8, 16, and 32 processors

The error range between said acceleration concerning Amdahl's Law is smaller in the execution of the HLPC than in the execution of the PDP. This occurs because HLPC execution times are better compared to PDP execution times as the number of processors is increased. In other words, as we increase the number of processors in the HLP and PDP executions, the former decreases its execution time faster. This is illustrated in Figure 11. Even so, the value of the magnitude known as Speedup, is always appreciated upwards as the execution times of both models improve in relation to their sequential counterparts, always below Amdahl's Law, which gives us "good" returns.

12. Conclusions

We have presented within Object-Oriented Programming, Structured Parallel Programming and within Parallel Objects two different models for the development of parallel applications: The Parallel Design Patterns or PDP model and the High-Level Parallel Compositions model or HLPC. Both proposals are based on the idea of generating generic constructs that contain the parallel structure that communicates to the different processes generated in each model according to an algorithmic design technique, and a common communication pattern for the "semi-automatic" parallelization of a sequential problem. In the case of PDP, the method of development of the Divide & Conquer Parallel Design Pattern is shown to solve the sorting problem through the parallelization as PDP of the Quicksort algorithm.

In the case of the HLPC, in the same way, the development of the Divide & Conquer High-Level Parallel Composition as a generic and reusable inter-

process communication pattern that implements the Divide & Conquer algorithmic technique using a binary tree as a pattern. associated communication is also illustrated.

Both proposals can be used by programmers without experience in the development of parallel applications to obtain efficient code, programming only the sequential part of their applications and using the parallel structure of the models described as libraries in their codes. Finally, the analysis of the performance of the models when used in the solution of the sorting problem with Quicksort was presented. This analysis shows the accelerations found (speedup) and the calculated run times, which also demonstrates the good performance in the executions on a 32-processor parallel machine and the good scalability of the accelerations compared to Amdahl's Law.

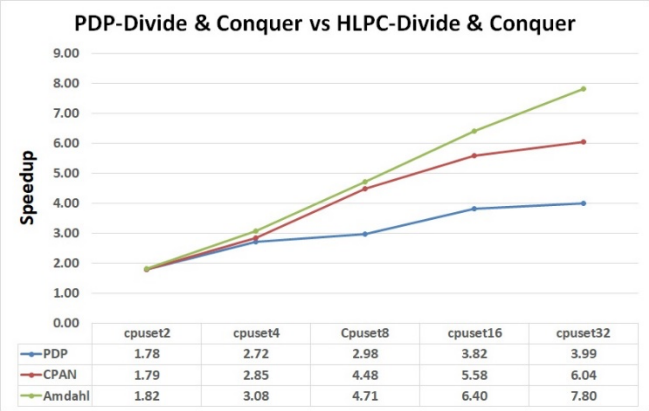


Figure 10. Comparison of the scalability of the PDP vs HLPC speedup in the implementation of Quicksort for the solution of the sorting problem with 2, 4, 8, 16 and 32 processors

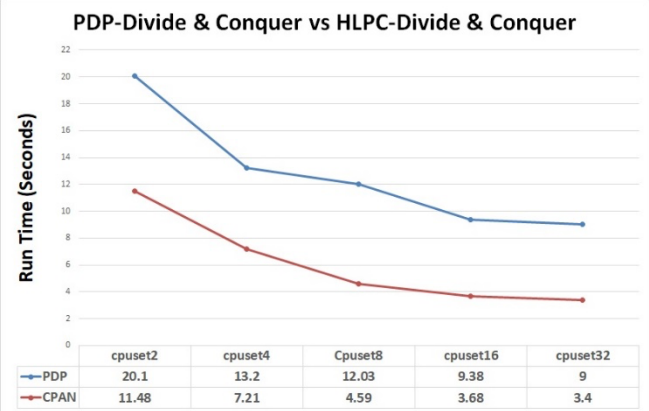


Figure 11. Run time in seconds of the PDP vs HLPC in the solution of the sorting problem applying Quicksort on 2, 4, 8, 16 and 32 processors.

References

Alba, E., Luque, G., Garcia, J. and Ordonez, G. (2007). MALLBA: a software library to design efficient optimization algorithms. International Journal of Innovative Computing and Applications, Vol. 1, No. 1, pp.74–85.

Brassard G. and Bratley P. (1997). Fundamentals of Algorithmics, Prentice-Hall, USA.

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

Collins A.J. (2011). Automatically Optimizing Parallel Skeletons. MSc thesis in Computer Science, School of Informatics University of Edinburgh, UK.

Corradi A. and Leonardi L. (1991). PO Constraints as tools to synchronize active objects. Journal Object Oriented Programming 10:42-53.

Corradi A. and Zambonelli I. (1995). Experiences toward an Object-Oriented Approach to Structured Parallel Programming. DEIS technical report no. DEIS-LIA-95-007.

Danelutto M. and Torquati M. (2014). Loop parallelism: a new skeleton perspective on data parallel patterns, in Proc. Of Intl. Euromicro PDP 2014. Parallel Distributed and Network-based Processing, Torino, Italy.

Danish S.A. and Farooqui Z. (2013). Approximate multiple pattern string matching using bit parallelism: a review, International Journal of Computer Applications, 74:47–51.

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

De Simone, et al. (1997). Design Patterns for Parallel Programming. PDPTA'96 International Conference.

Ernsting S. and Kuchen H. (2012). Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. of High-Performance Computing and Networking, Vol. 7:129–138.

Lavander G.R. and Kafura D.G. (2010). A Polymorphic Future and First-class Function Type for Concurrent Object-Oriented Programming. Journal of Object-Oriented Systems.

McCool M., Robison A.D. and Reinders J. (2012). Structured Parallel Programming. Patterns for Efficient Computation. Morgan Kaufmann Publishers Elsevier. USA.

Roosta and Séller. (1999). Parallel Processing and Parallel Algorithms. Theory and Computation. Springer.

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

Rossainz M. and Capel M. (2017). Design and implementation of communication patterns using parallel objects. Especial edition, Int. J. Simulation and Process Modelling, 12:1.

Steuwer, M., Kegel, P. and Gorlatch, S. (2011). SkelCL a portable skeleton library for high-level GPU programming. Proceedings of the 16th IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments, May, Anchorage, AK, USA.

Theelen B.D., Florescu O., et al. (2007). Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. IEEE/ACM International Conference on Formal Methods and Models for Codesign. Doi: 10.1109/MEMCOD.2007.371231. Nice, France.

Torquati, M., Aldinucci, M. and Danelutto, M. (2014). FastFlow documentation, Parallel programming in FastFlow, Computer Science Department, University of Pisa, Italy, [online] http://calvados.di.unipi.it/storage/refman/doc/html/index.html

Torquati, M., Aldinucci, M. and Danelutto, M. (2015) FastFlow Testimonials, Computer Science Department. University of Pisa, Italy. [online] https://alpha.di.unito.it/

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