Applying Design Patterns for Communication Components: Communicating CSE components for the Laplace Equation

Jorge L. Ortega-Arjona
jloa@ciencias.unam.mx
Departamento de Matemáticas
Facultad de Ciencias, UNAM
MEXICO

Abstract

The Design Patterns for Communication Components is a group of patterns within a method for developing the communication sub-systems of parallel software systems. They take as input: *(a)* the architectural pattern of the overall parallel software system, *(b)* the memory organization of the parallel hardware platform, and *(c)* the type of synchronization required.

This paper presents the application of the Design Patterns for Communication Components as part of the method for designing the communication components for the CSE solution to the Laplace Equation. The method used takes the information from the previous steps, Problem Analysis and Coordination Design, applying a design pattern for the communication components, and providing elements about its implementation.

CCS Concepts: • **Software and its engineering** → **Designing software**.

Keywords: Design Patterns, Laplace Equation, Parallel Software Design

ACM Reference Format:
Jorge L. Ortega-Arjona. 2022. Applying Design Patterns for Communication Components: Communicating CSE components for the Laplace Equation. In *Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP'22), October 18, 2022*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SugarLoafPLoP'22, October 18, 2022,
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Parallel programming is characterized by evolving parallel hardware architectures, programming paradigms, and parallel languages. This makes difficult to propose a single approach containing all the details to design and implement communication components for all parallel software systems. Hence, the Design Patterns for Communication Components [14, 16] have been proposed as an effort to aid programmers to design the communication components, depending on particular characteristics and features of the communication to be carried out when designing a parallel program.

The Design Patterns for Communication Components describe and refine the communication components of a parallel program by describing common programming structures used for communicating, exchanging data, or requesting operations between processing components. Their application directly depends on the Architectural Pattern for Parallel Programming [12, 16] which they are part of, detailing a communication and synchronization function as a local problem, and providing a form as a local solution of software components for such a communication problem.

When designing the communication components, it is important to carefully think how communication and synchronization are to be actually performed. From the many descriptions about how to organize the communication components of a parallel program [1, 7, 11] the Design Patterns for Communication Components are proposed having the following advantages [14, 16]:

- The Design Patterns for Communication Components describe some common structures used to refine and detail the communications and synchronization required by an Architectural Pattern for Parallel Programming [12, 16]. From this point of view, their objective is to help the software designer or programmer by providing medium-scale descriptions of software compounds that are used to communicate parallel processing components.

There has been an extensive research and development of such software compounds [1, 7, 11], but unfortunately, normally they have not been related or linked with overall structures of parallel programs.

- The Design Patterns for Communication Components, as any software pattern, are descriptions about how to relate a communication function (in run-time terms) with a coded form (in compile-time terms). In many parallel applications, communication components are designed so their run-time communication function is organized having little resemblance to the compile-time organization of code that performs it. In fact, both organizations are by far very independent from each other, making it difficult to notice how communication is performed by coded components. The Design Patterns for Communication Components attempt to connect both descriptions into a single description that provides both, dynamic and static information about those communication components.
- Design Patterns for Communication Components describe software sub-systems or sub-structures for data exchange and/or function call. As such, they are a guidance about how to find the set of software component that perform communication and synchronization between processing components. The communication between processing components of a parallel software system is a key for the success or failure of such a parallel software system. Hence, the Design Patterns for Communication Components are developed and classified based on *(a)* the architectural pattern used for the overall parallel software system, *(b)* the memory organization of the parallel hardware platform, and *(c)* the type of synchronization in the parallel programming language available. These three features strongly affect the design and implementation of communication software components. Thus, the Design Patterns for Communication Components describe and document several generic and different software structures for communication components, referring to different kinds of architectural patterns (Parallel Pipes and Filters, Parallel Layers, Communicating Sequential Elements, Manager-Workers, and Shared Resource) [12, 16], the hardware platform memory organization (shared memory or distributed memory), and the different types of synchronization available is common programming languages (synchronous or asynchronous).
- Normally, the Design Patterns for Communication Components are described so their components are explained in Object-Oriented terms. This does not prevent that they can be developed using other different paradigms and programming languages (such as process or task oriented). Nevertheless, such a description allows to take into consideration and advantage Object-Oriented principles, such as the separation between interfaces and implementation, reuse, and modifiability of code [6, 17].
- The Design Patterns for Communication Components introduce communication structures as forms in which software components are assembled or arranged together in order to perform communication. The communication structures are represented by forms as regular organizations of software components, aiming to allow software designers to understand complex communication software sub-systems, and therefore, reducing their cognitive burden.
- Design Patterns for Communication Components are based on the common concepts and terms originally used for inter-process communication [2–5, 8–10]. As such, these design patterns make use and provide elements to develop a terminology for designing communication components for parallel programs.

This paper is the second in a series of three papers: in the first paper [15] the Communicating Sequential Elements (CSE) has been applied as an architectural pattern for solving the Laplace Equation; the present paper is the second, presenting the application of Design Patterns for Communication Components for solving the communication issues between the components of the CSE pattern, and; there should be another paper which has to cover the application os idioms for synchronization issues within the communication components, but due to space restrictions, has to be presented later. The three papers, taken together, should present a complete picture of a parallel implementation for the Laplace Equation.

2 Specification of the System

In the paper *"Applying Architectural Patterns for Parallel Programming. Solving the Laplace Equation"* [15], the Communicating Sequential Elements (CSE) Architectural Pattern has been selected as a solution for the Laplace Equation. Here, in order to apply the Design Patterns for Communication Components to develop the communication components for this example, it si required some information about the CSE Pattern, the available parallel hardware platform, and the programming language. This information is summarized in the following sections.

2.1 The Communicating Sequential Elements pattern

The algorithmic solution for the Laplace Equation is defined in terms of calculating the next temperature of all elements that compose a two-dimensional surface, as ordered data. Each temperature is obtained almost autonomously. The exchange of data or communication should be between neighboring elements of the surface. So, the CSE pattern has been chosen as an adequate solution for the Laplace Equation. The design of the parallel software system continues, based on the Solution section of the CSE pattern, as briefly described as follows [13, 15].

- **Description of the coordination**. The CSE pattern describes multiple **Elements** as concurrent processing software components, each one applying the same operation, whereas **Channels** act as communication software components, allowing the exchange of temperature values between processing software components [13, 15].
- **Structure and dynamics**
  1. *Structure*. The structure of the actual solution involves a regular, two-dimensional, logical structure, conceived from the surface of the original problem. The solution is a two-dimensional network of elements that follows the shape of the surface. An Object Diagram taken from [15] as background information, represents the network of elements that follows the two-dimensional shape of the surface and its division into elements, as shown in Figure 1.

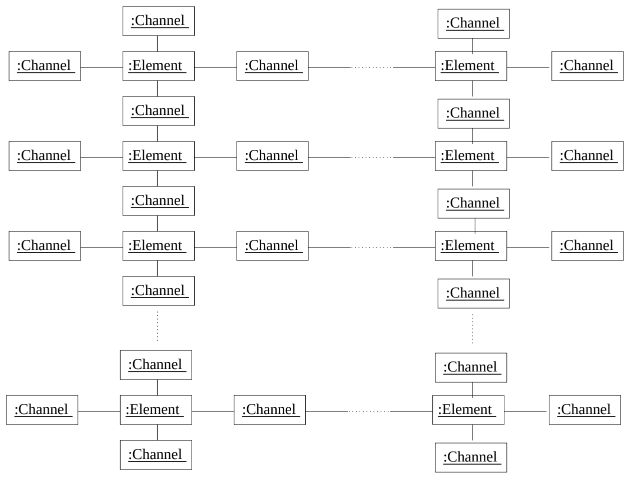

Figure 1. Object Diagram of CSE for solving the Laplace Equation [15].

  2. *Dynamics*. A scenario to describe the basic runtime behavior of the CSE pattern for solving the Laplace Equation is shown. Notice that all the elements, as basic processing software components, are active at the same time. Every element performs the same operation, as a piece of a processing network. However, for the two-dimensional case here, each element object communicates with its four neighbors as shown in Figure 2 [15].

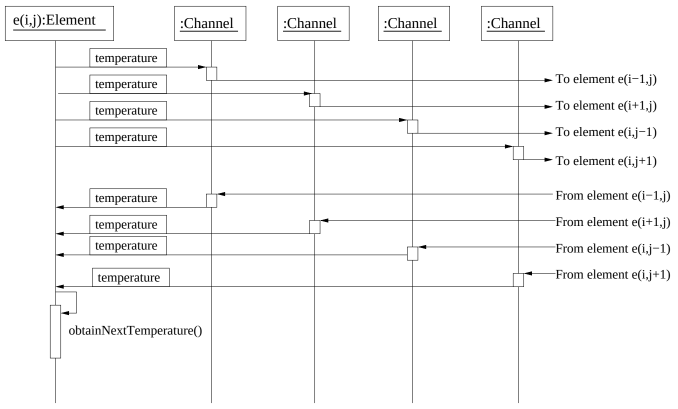

Figure 2. Sequence Diagram of CSE for communicating temperatures through channel components for the Laplace Equation [15].

2.2 Information about parallel platform and programming language

The parallel system available for this example is a SUN SPARC Enterprise T5120 Server. This is a multi-core, shared memory parallel hardware platform, with 1×8-CoreUltraSPARC T2, 1.2 GHz processors (capable of running 64 threads), 32 Gbytes RAM, and Solaris 10 as operating system. Applications for this parallel platform can be programmed using the Java programming language [12].

3 Communication Design – Specification of Communication Components

3.1 The scope

Let us consider the basic information about the parallel hardware platform and the programming language used, as well as the CSE pattern as the selected coordination for solving the Laplace Equation. The objective is to look for the relevant information when choosing a particular design pattern as a communication structure. Hence, the information about the parallel platform (a shared memory multi-core computer), the programming language (Java) and the description of channels as communication software components for the CSE pattern presented in the previous section, the procedure for applying a Design Pattern for the Communication Components for the Laplace Equation problem is presented here [14, 16]:

1. *Consider the architectural pattern selected in the previous step.* From the CSE pattern, the design patterns which provide communication components and allow the behavior are the Shared Variable Channel pattern and the Message Passing Channel pattern [14, 16].
2. *Select the nature of the communicating components.* The parallel hardware platform has a shared memory organization, and thus, the nature of the communicating components is considered to be shared variable.
3. *Select the type of synchronization required for the communication.* Normally, the communication between software components that compose an array communicated through point to point communication components makes use of an asynchronous communication. If a synchronous communications would be used, it is very likely that the processing software components may block, waiting for receiving temperature values from their counterpart. As every software component would be waiting, none would be receiving, leading to a deadlock situation. In this case, using a relaxation method such as Gauss-Seidel or the successive over-relaxation (SOR) at the coordination level would sort this out. Nevertheless, in the Problem Analysis it is stated that a Jacobi relaxation is to be used [15]. So, this makes it more important to make use of an asynchronous communication, which avoids every sender to wait for its receivers.
4. *Selection of a design pattern for communication components.* Considering (a) the use of the CSE pattern, (b) the shared memory organization of the parallel platform, and (c) the use of asynchronous communications, therefore the **Shared Variable Channel pattern** is proposed here as the base for designing the communications. Let us review the Context and Problem sections of the CSE pattern [14, 16]:
   - **Context:** 'A parallel program is being developed using the Communicating Sequential Elements architectural pattern as a domain parallelism approach in which the data is partitioned among autonomous processes (elements) as the processing components of the parallel program. The parallel program is developed for a shared memory computer. The programming language to be used counts with synchronization mechanisms for process communication such as semaphores, critical regions, or monitors'
   - **Problem:** 'A sequential element needs to exchange values with its neighboring elements. Every data is locked inside each sequential element, which is responsible for processing that data and only that data'

   From these descriptions, the selection for the communication components for this example is the **Shared Variable Channel pattern** [14, 16]. The use of a shared memory parallel platform implies using shared variables, and it is known that the Java programming language counts with the elements for developing semaphores or monitors. Moreover, the channels consider an asynchronous communication scheme between sender and receiver. Therefore, this completes the selection of the Design Pattern for Communication Components for the Laplace Equation. The design of the parallel software system continues, using the Shared Variable Channel pattern's Solution section as a starting point for communication design and implementation.

3.2 Functional description of software components

Each software component of the Shared Variable Channel pattern, as the participant of the communication sub-system, is described establishing its responsibilities, input, and output [14, 16].

1. **Synchronization Mechanisms.** These components are used to synchronize the access to the `Double` shared variables. They should allow the translation of `send()` and `receive()` operations into adequate operations for writing to, and reading from, the shared variables. Normally, synchronization mechanisms keep the order and integrity of the shared data.
2. **Shared Variables.** The shared variables store the `Double` variables that hold the temperature values exchanged by sequential elements. They are simple variables used as buffers for actually achieving an asynchronous communication.

3.3 Structure and dynamics

This section takes information of the Shared Variable Channel design pattern, expressing the interaction between its software components that carry out the communication between parallel software components for the actual example [14, 16].

1. *Structure.* The Shared Variable Channel pattern is applied for designing and implementing channel

communication components of the CSE pattern, as shown in Figure 3. Notice that the channel component structure allows an asynchronous, bidirectional communication between two sequential elements. The asynchronous feature is achieved by allowing an array of temperatures to be stored, so the sender does not wait for the receiver [14, 16].

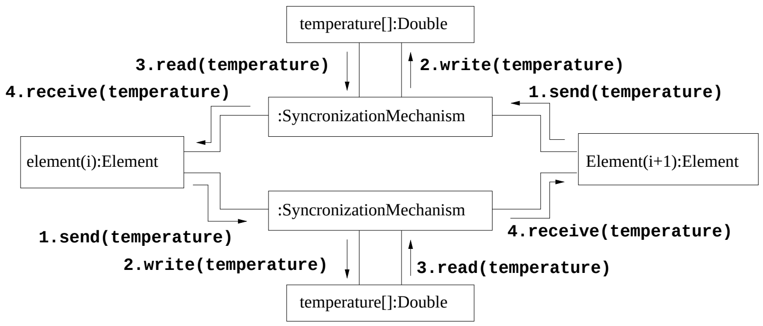

Figure 3. UML Collaboration Diagram of the Shared Variable Channel pattern used for asynchronously exchange temperature values between sequential components of the CSE solution to the Laplace Equation.

2. *Dynamics.* The Shared Variable Channel pattern performs the operation of a channel component within the available shared memory, multi-core parallel platform. Figure 4 shows the behavior of the participants of this pattern for the actual example [14, 16].

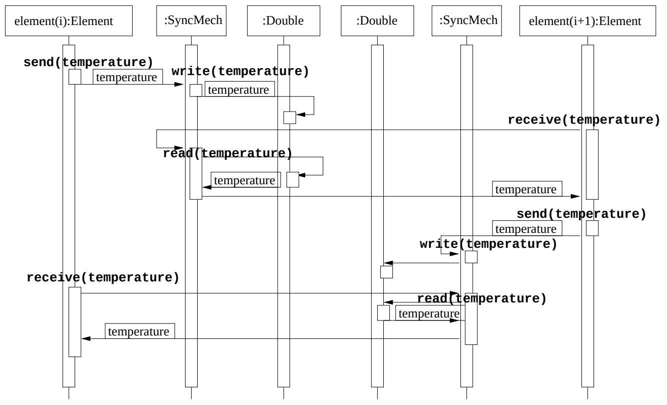

Figure 4. UML Sequence Diagram for the Shared Variable Channel pattern applied for exchanging temperature values between two neighboring sequential elements of the CSE solution for the Laplace Equation.

Here, a point to point, bi-directional, asynchronous communication exchange of temperature values of type `Double` is carried ou as follows:

- The `element(i)` sequential element sends its local `temperature` value by issuing a `send(temperature)` operation to the sending Synchronization Mechanism.
- This Synchronization Mechanism verifies if the `element(i+1)` sequential element is not reading the `temperature` shared variable. If this is the case, then it translates the sending operation, allowing a `write(temperature)` operation of the data item on `temperature`. Otherwise, it blocks the operation until the `temperature` can be safely written.
- When the `element(i+1)` attempts to receive the temperature value, it does so by issuing a `receive(temperat` request to the Synchronization Mechanism. This function returns a `double` type representing the temperature value stored in the shared variable `temperature`. Again, only if its counterpart sequential element (here, `element(i)`) is not writing on `temperature`, the Synchronization Mechanism grants a `read(temperature)` operation from it, returning the requested `temperature` value. This achieves the send and receive operations between neighboring elements.
- On the other hand, when data flows in the opposite direction, a similar procedure is carried out, but from `element(i+1)` to `element(i)`.

3.4 Description of the communication

The channel communication component acts as a single entity, allowing the exchange of information between processing software components. Given that the available parallel platform is a multi-core, shared memory system, the behavior of a channel component is modelled using shared variables. A couple of shared variables are used to implement the channel component as a bidirectional, shared memory communication means between elements. Such shared variables require to be safely modified by synchronizing read and write operations from the elements. Hence, the Java programming language provides the basic elements for developing synchronization mechanisms (such as semaphores or monitors). This is required to preserve the order and integrity of the transferred temperature values.

3.5 Communication Analysis

This section describes the advantages and disadvantages of the Shared Variable Channel pattern as a base for the communication structure proposed.

1. **Advantages**
   - The communication sub-structure allows to keep the precise order of the exchanged temperature values by considering a two directional FIFOs

for its implementation, as well as synchronizing the access to both Double type shared variables.

- The communication sub-structure allows for a point to point, bidirectional communication component.
- The use of Synchronization Mechanisms grants keeping the integrity of transferred temperature values, assuring that at any given moment only one element actual accesses any Double type shared variables.
- The use of shared variables implies that the implementation is particularly developed for a shared memory parallel platform.
- The Shared Variable Channel pattern allows the use of asynchronous communications between sequential elements by using the two Double type shared variables as two communication buffers.

2. **Liabilities**
   - As the available parallel platform is a shared memory one, the communication speed tends to be similar to simple assignation operations over shared variable addresses. Communications are only delayed by the synchronization actions taken by the Synchronization Mechanisms in order to keep the integrity of the temperature values. Nevertheless, it is very little what can be done to improve communication performance in terms of programming. The only programming action that can be taken is to change the amount of processing of the sequential elements, which modify the granularity, tuning the communication speed.
   - The implementation based on shared variables and synchronization mechanisms such as semaphores, conditional regions, or monitors, makes these communication sub-systems only suitable for shared memory platforms. If the parallel software system is considered to be ported to a distributed memory parallel platform, this would require replacing each Shared Variable Channel pattern by a Message Passing Channel pattern [14], and design and implement the communication sub-systems as indicated by this pattern.

4 Implementation

All the software components described in the Communication Design step are implementated using the Java programming language. However, it is only presented here the implementation of the communication sub-system, which interconnects processing components that implement the actual computation that is to be executed in parallel [15]. So, the implementation is presented for developing the channel as communication and synchronization components. Nevertheless, this design and implementation of the whole parallel software system goes beyond the actual purposes of the present paper.

4.1 Synchronization Mechanism

Based on the Java programming language, a basic description of a synchronization mechanism that controls the access to the temperature array is presented as follows:

```
import java.util.Vector;
class SynchronizationMechanism {
   private int numMessages = 0;
   private final Vector temperatures = new Vector();
   public final synchronized void write(double temp){
      ...
      numMessages++;
      temperatures.addElement(temp);
      ...
   }
   public final synchronized double read(){
      double temp = 0.0d;
      numMessages--;
      ...
      temp = temperatures.firstElement();
      temperatures.removeElementAt(0);
      return temp;
   }
}
```

The class SynchronizationMechanism presents two synchronized methods, write() and read(), which allow to safely modify the temperatures buffer and allows an asynchronous communication between element components. This class is used in the following implementation stage as the basic element of the channel components.

4.2 Communication components – Channels

Using the class SynchronizationMechanism from the previous section, here it is used as the synchronization mechanism component as described by the Shared Variable Channel pattern, in order to implement the class Channel, as follows:

```
public final class Channel {
   private SynchronizationMechanism m0 = null;
   private SynchronizationMechanism m1 = null;
   public Channel(){
      m0 = new SynchronizationMechanism();
      m1 = new SynchronizationMechanism();
   }
   public void send0(Channel c, double temp){
      if(temp == null) throw new NullPointerException();
      m0.write(temp);
   }
   public void send1(Channel c, double temp){
      if(temp == null) throw new NullPointerException();
         m1.write(temp);
   }
   public double receive0(Channel c){
      return m0.read();
   }
   public double receive1(Channel c){
      return m1.read();
   }
}
```

Each channel component is composed of two synchronization mechanisms, which allow the bi-directional flow of data through the channel. In order to keep the direction of each message flow, it is necessary to define two methods for sending and other two methods for receiving, and keep attention about the flow of messages. Each method distinguishes on which synchronization mechanism of the channel the message is written. This allows that the channel is capable of allowing a simultaneous bi-directional flow. In the present example, this is used to enforce the use of the Jacobi relaxation [15]. In fact, using (a) a channel communication structure with two-way flow of data, (b) making each one of them asynchronous, and later, (c) taking care on the communication exchanges between element components, are all design previsions for avoiding any potential deadlock. In parallel programming, it is generally advised that during design, all previsions should be taken against the possibility of a deadlock.

5 Summary

The Design Patterns for Communication Components are applied here within a method for selecting them, in order to show how to apply a design pattern that copes with the requirements of communication present in the Laplace Equation problem. The main objective of this paper is to demonstrate, with a particular example, the detailed design and implementation that may be guided by a selected design pattern. Moreover, the application of the Design Patterns for Communication Components and the method for selecting them is proposed to be used during the Communication Design and Implementation for other similar problems that involve the calculation of differential equations for a two-dimensional problem, executing on a shared memory parallel platform.

References

[1] G.R. Andrews *Foundation of Multithreaded, Parallel and Distributed Programming.*, Addison-Wesley Longman, Inc., 2000.

[2] Brinch-Hansen, P., *Structured Multiprogramming.* Communications of the ACM, Vol. 15, No. 17. July, 1972.

[3] Brinch-Hansen, P., *The Programming Language Concurrent Pascal.* IEEE Transactions on Software Engineering, Vol. 1, No. 2. June, 1975.

[4] P. Brinch-Hansen *Distributed Processes: A Concurrent Programming Concept.*, Communications of the ACM, Vol.21, No. 11, 1978.

[5] E.W. Dijkstra *Co-operating Sequential Processes*, In Programming Languages (ed. Genuys), pp.43-112, Academic Press, 1968.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Systems. Addison-Wesley, Reading, MA, 1994.

[7] S. Hartley *Concurrent Programming. The Java Programming Language.*, Oxford University Press Inc., 1998.

[8] Hoare, C.A.R., *Towards a theory of parallel programming.* Operating System Techniques, Academic Press, 1972.

[9] Hoare, C.A.R., *Monitors: An Operating System Structuring Concept.* Communications of the ACM, Vol. 17, No. 10. October, 1974.

[10] C.A.R. Hoare *Communicating Sequential Processes.* Communications of the ACM, Vol.21, No. 8, August 1978.

[11] S. Kleiman, D. Shah, and B. Smaalders *Programming with Threads*, 3rd ed. SunSoft Press, 1996.

[12] J.L. Ortega-Arjona and G.R. Roberts *Architectural Patterns for Parallel Programming*, Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP98), Kloster Irsee, Germany, 1998.

[13] J.L. Ortega-Arjona *The Communicating Sequential Elements Pattern. An Architectural Pattern for Domain Parallelism*, Proceedings of the 7th Conference on Pattern Languages of Programming (PLoP2000), Allerton Park, Illinois, USA, 2000.

[14] J.L. Ortega-Arjona *Design Patterns for Communication Components*, Proceedings of the 12th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2007), Kloster Irsee, Germany, 2007.

[15] J.L. Ortega-Arjona *Applying Architectural Patterns for Parallel Programming. Solving the Laplace Equation*, Proceedings of the 23rd 8th European Conference on Pattern Languages of Programs (EuroPLoP2018), Kloster Irsee, Germany, 2018.

[16] J.L. Ortega-Arjona *Patterns for Parallel Software Design*, John Wiley & Sons, 2010.

[17] Shalloway, A., and Trott, J.R., *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Software Pattern Series. Addison-Wesley, 2002.