# Applying Architectural Patterns for Parallel Programming

## Solving the Laplace Equation

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


*EuroPLoP '18, July 4–8, 2018, Irsee, Germany*

ACM ISBN 978-1-4503-6387-7/18/07.
https://doi.org/10.1145/3282308.3282335

## ABSTRACT

The Architectural Patterns for Parallel Programming is a collection of patterns related with a method for developing the coordination structure of parallel software systems. These architectural patterns take as input information *(a)* the available parallel hardware platform, *(b)* the parallel programming language of this platform, and *(c)* the analysis of the problem to solve, in terms of an algorithm and data.

In this paper, it is presented the application of the architectural patterns along within the Coordination stage, as part of the Pattern-based Parallel Software Design Method, which aims for developing a coordination structure for solving the Laplace Equation. The Coordination stage here takes the information from the Problem Analysis presented in Section 2, selects an architectural pattern for the coordination in Section 3, and provides some elements about its implementationin Section 4.

## CCS CONCEPTS

• **Software and its engineering** → **Cooperating communicating processes**; **Cooperating communicating processes**; **Object oriented architectures**;

## KEYWORDS

Architectural Patterns, Parallel Programming, Laplace Equation

**ACM Reference Format:**
Jorge L. Ortega-Arjona. 2018. Applying Architectural Patterns for Parallel Programming: Solving the Laplace Equation. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3282308.3282335

## 1 INTRODUCTION

A parallel program is *the specification of a set of processes executing simultaneously, and communicating among themselves in order to achieve a common objective.* This definition is obtained from the original research work in parallel programming provided by E.W. Dijkstra [2], C.A.R. Hoare [5], P. Brinch-Hansen [1], and many others, who have established the main basis for parallel programming today. Specifically, obtaining a parallel program from an algorithmic description is the main objective of the area of Parallel Software Design.

Practitioners in the area of parallel programming recognize that the success of a parallel program is able to achieve – commonly, in terms of performance– is affected by three main factors [8, 9]: *(a)* the hardware platform, *(b)* the programming language, and *(c)* the problem to solve. Nevertheless, from a review of literature, it is noticeable that most of research in parallel systems and parallel programming has normally been devoted to the first two factors: the hardware platform and the programming language. It is just until a few years ago that several researchers have focused on parallel programming from the point of view of the problem to solve, proposing the area of Parallel Software Design in order to study how and at what point the organization of a parallel program affects the development and performance of a parallel system.

Parallel Software Design proposes programming techniques to deal with the parallelization of a problem, described in algorithmic terms. The research in the area covers several approaches that provide forms to organize software with relatively independent parts which efficiently make use of multiple processors. As stated before, the goal is to obtain a parallel program from an algorithmic description. Nevertheless, designing parallel programs can be frustrating [9]:

- There are lots of issues to consider when parallelizing an algorithm. How to choose a coordination structure that is not too hard to program and that offers substantial performance compared to uniprocessor execution?
- The overheads involved in synchronization among multiple processors may actually reduce the performance of a parallel program. How to anticipate and mitigate this problem during testing and evaluation?
- Like many performance improvements, parallelizing increases the complexity of a program. How to manage such a complexity?

These are tough problems: we do not yet know how to solve an arbitrary problem efficiently on a parallel system of arbitrary size. Hence, Parallel Software Design, at its actual stage of development, does not (cannot) offer universal solutions, but tries to provide some simple ways to get started.

Simply put, architectural patterns allow software designers and developers to understand complex software systems in larger conceptual blocks and their relations, thus reducing the cognitive burden. Furthermore, architectural patterns provide several "forms" in which software components of a parallel software system can be structured or arranged, so the overall structure of such a software system arises. Architectural patterns also provide a vocabulary that may be used when designing the overall structure of a parallel software system, to talk about such a structure, and feasible implementation techniques. As such, the Architectural Patterns for Parallel Programming refer to concepts that have formed the basis of previous successful parallel software systems. Examples of Architectural Patterns for Parallel Programming are Parallel Pipes and Filters, Parallel Layers, Communicating Sequential Elements, Manager-Workers, and Shared Resource [6, 8, 9].

## 2 PROBLEM ANALYSIS – THE LAPLACE EQUATION

The present paper attempts to demonstrate the use of the Architectural Patterns for Parallel Programming for designing a coordination structure that solves the Laplace Equation. The objective is to show how an architectural pattern can be selected so it deals with the functionality and requirements present in this problem.

### 2.1 Problem Statement

Partial differential equations are commonly used to describe physical phenomena that continuously change in space and time. One of the most studied and well known of such equations is the Laplace Equation, which mathematically models the steady-state heat flow in a region that exposes certain dimensionality, with certain fixed temperatures on its boundaries. In the present example, the region is represented by a two-dimensional entity, for example, a plate of homogeneous material and uniform thickness. The surroundings of the plate are perfectly insulated, and on the extremes, each point keeps a known, fixed temperature. As heat flows through the plate, the temperature of each point eventually reaches a value or state in which such a point has a steady, time-independent temperature maintained by the heat flow. Thus, the problem of solving the Laplace Equation is to define the equilibrium temperature $u(x, y)$ for each point $(x, y)$ on the two-dimensional plate. Normally, the heat is studied as a flow through an elementary piece of the plate, a finite element. This element is represented as a small, two-dimensional element of the plate, with an area of $\Delta x \times \Delta y$ (Figure 1).

Given the insulation surrounding the plate, there could only be a flow through its two dimensions. At every point $(x, y)$, the velocity of the heat flow is considered to have a horizontal flow component, $v_x$, and a vertical flow component $v_y$, which are represented in terms of its temperature $u(x, y)$ by the equation:

$$v_x = -k\frac{\partial u}{\partial x} \tag{1}$$

$$v_y = -k\frac{\partial u}{\partial y} \tag{2}$$

This equation means that heat flow is proportional to the temperature gradient, towards decreasing temperatures. Moreover, in equilibrium, the element holds a constant amount of heat, making its temperature $u(x, y)$ a constant. Thus, in the steady-state, this is expressed as:

$$\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} = 0 \tag{3}$$

Combining this equation with the previous equation for the velocity of flow, thus Laplace's law for equilibrium temperatures arises:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{4}$$

Known as the Laplace equation or equilibrium equation, this equation is abbreviated and expressed in general terms (and dimensions) as:

$$\nabla^2 u = 0 \tag{5}$$

A function $u(x, y)$ that satisfies this equation is known as a "potential function", and it is determined by boundary conditions. By now, for the actual purposes, the Laplace Equation allows to mathematically model the heat flow through a plate. Nevertheless, in order to develop a program that numerically solves this equation, it is still required to perform a series of further considerations. Let us consider by now a thin plate, for which temperatures are considered fixed at each extreme (Figure 2).

In order to develop a program that models the Laplace Equation, first it is necessary to obtain its discrete form. So, the plate in Figure 2 is divided into elements, each element with a size of $a \times a$. This size is relatively very small regarding the size of the whole plate, so the element can be considered as a single point within the plate. So, this results on a divided plate, in which three types of elements can be considered (Figure 3).

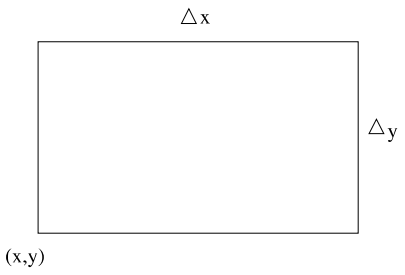


**Figure 1: A small two-dimensional element.**

(1) Interior elements, which require computing their temperatures, each one having to satisfy the Laplace equation.
(2) Extreme elements, which have fixed and given temperatures.
(3) Corner elements, which are not used.

The discrete solution of the Laplace Equation is based on the idea that the heat flow through interior elements is due to the temperature differences between an elements and all its neighbors. Let us suppose the temperature of a single interior element $u(i,j)$, whose four adjacent neighboring elements are $u(i-1,j)$, $u(i+1,j)$, $u(i,j-1)$ and $u(i,j+1)$ (Figure 4).

Notice that for the case, $a$ should be small enough so each neighboring element's temperature can be approximated. So, the discrete heat equation is reduced to a difference equation. Rearranging it, it is noticeable that for thermal equilibrium, the temperature of a single element $u(i,j)$ in time, from one thermal state to another, is:

$$u(t+1,i,j) \approx \frac{1}{4}[u(t,i-1,j)+u(t,i+1,j)+u(t,i,j-1)+u(t,i,j+1)] \quad (6)$$

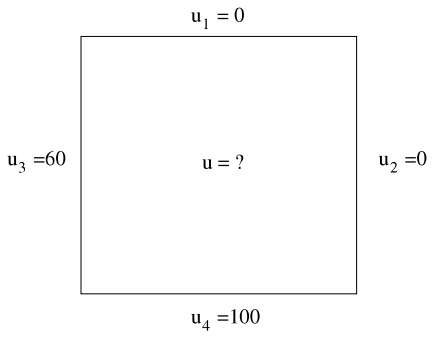


**Figure 2: A plate with fixed temperatures at each extreme.**

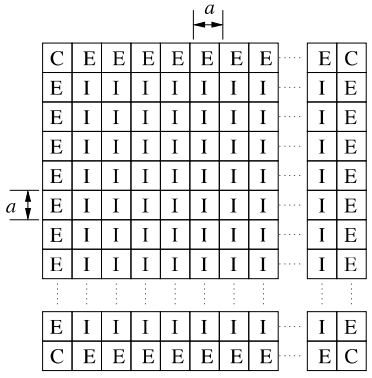


**Figure 3: A plate divided in three types of elements: interior (I), extreme (E), and corner (C).**

This is the discrete equation to be used in order to obtain a parallel numerical solution for the Laplace Equation.

## 2.2 Specification of the Problem

From the previous section, it is noticeable that using a plate divided into $n \times n$ elements, the discrete form of the Laplace Equation implies a computation for each discrete elements of the plate. Moreover, taking into consideration the time as another dimension so the evolution of temperatures through time can be observed, and solving it using a direct method on a sequential computer, requires something like $O(n^3)$ units of time. Suppose a numerical example: for a plate with, for example, $n = 65536$, it is required to solve about the same number of average operations, involving floating point coefficients. Using a sequential computer with a clock frequency of about 1MHz, it would take about eight years for the computation. Furthermore, notice that naive changes to the requirements (which are normally requested when performing this kind of simulations) produce drastic (exponential) increments of the number of operations required, which at the same time affects the time required to calculate this numerical solution.

- *Problem Statement.* The Laplace Equation, in its discrete representation, and for a relatively large number of elements in which a plate is divided, can be computed in a more efficient way by:
  (1) using a group of software components that exploit the two-dimensional logical structure of the plate, and
  (2) allowing each software component to simultaneously calculate the temperature value for all elements of the plate at a given time step.

  The objective is to obtain a result in the best possible time-efficient way.
- *Descriptions of the data and the algorithm.* The relatively large number of elements in which a plate is divided and the discrete representation of the Laplace Equation is described in terms of data and an algorithm. The divided region is normally represented as a large plate in terms of a $(n+2) \times (n+2)$ array of elements which represent every discrete element of the plate, and encapsulate some floating point data which

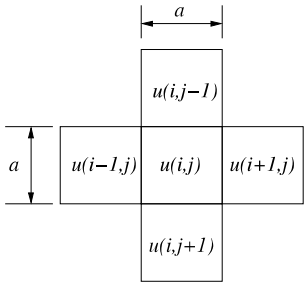


**Figure 4: An element $u(i,j)$ and its four neighboring elements.**

represents temperature, as shown as follows. Thus, a whole plate consists of $n \times n$ interior elements and $4n+4$ extreme elements. For example, in Java elements are represented as objects of a class `Element`, as follows:

```
class Element implements Runnable{
   ...
   private int i = -1;
   private int j = -1;
   ...
   private Element(int i, int j){
      this.i = i;
      this.j = j;
      new Thread(this).start();
   }
   ...
}
```

Each **Element** object is able to compute a local discrete heat equation as a single thread. Thus, it exchanges messages with its neighboring elements (whether interior or extreme) and computes its local temperature, as follows:

```
class Element implements Runnable{
   ...
   private int i = -1;
   private int j = -1;
   ...
   public void run(){
      double temperature, received, total;
      for (int i = 0; i < iterations; i++) {
         // Here the actual element exchanges data with
         // its neighboring elements
         total = 0.0;
         for (i = 0; i < 4; i++) {
            // Receive from neighboring elements
            // and put it in the variable 'received'
            total += received;
         }
         temperature = total/4;
      }
   }
   ...
}
```

Each time step, a new temperature for the local **Element** object is obtained from the previous temperature and the temperatures of the neighboring elements (whether interior or extreme), until a total number of `iterations` is achieved. Notice that the term "time step" implies an iterative method in which the operation requires four coefficients. The algorithm described takes into consideration an iterative solution of operations, known as *relaxation*. The simplest relaxation method is the Jacobi relaxation, in which the temperature of each and every interior element is simultaneously approximated using its local temperature and the temperatures of its neighbors (and it is the one presented here). Other relaxation methods include the Gauss-Seidel relaxation and the successive over-relaxation (SOR). Iterative methods tend to be more efficient than direct methods.

- *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 $1 \times 8$-Core UltraSPARC T2, 1.2 GHz processors (capable of running 64 threads), 32 Gbytes RAM, and Solaris 10 as operating system [10]. Applications for this parallel platform can be programmed using the Java programming language [4].
- *Quantified requirements about performance and cost.* This application example has been developed in order to test the parallel system described in the previous point. The idea is to experiment with the platform, testing its functionality in time, and how it maps with a domain parallel application. So, the main objective is simply to test and characterize performance (in terms of execution time) regarding the number of processes/processors involved in solving a fixed size problem. Thus, it is important to retrieve information about the execution time considering several configurations, changing the number of processes on this parallel, shared memory platform.

## 3 COORDINATION DESIGN

In this section, the *Architectural Patterns for Parallel Programming* [6, 8, 9] are used along with the the information from the Problem Analysis, in order to apply an architectural pattern for developing a coordination that solves the Laplace Equation.

### 3.1 Specification of the System

- **The scope**. This section aims to describe the basic operation of the parallel software system, considering the information presented in the Problem Analysis step about the parallel system and its programming environment. Based on the problem description and algorithmic solution presented in the previous section, the procedure for applying an architectural pattern for a parallel solution to the Laplace Equation problem is presented as follows [6, 9]:

(1) *Analyze the design problem and obtain its specification.* Analyzing the problem description and the algorithmic solution provided, it is noticeable that the calculation of the Laplace Equation is a step-by-step, iterative process. Such a process is based on calculating the next temperature of each element of the surface through each time step. The calculation uses as input the temperatures of the four neighbor elements of the surface, and provides the temperature at the next time step.

(2) *Select the category of parallelism.* Observing the form in which the algorithmic solution partitions the problem, it is clear that the surface is divided into elements, and computations are executed simultaneously by different elements. Hence, the algorithmic

solution description implies the category of **Domain Parallelism**.

(3) *Select the category of the nature of the processing components.* Also, from the algorithmic description of the solution, it is clear that the temperature of each element of the surface is obtained using exactly the same calculations. Thus, the nature of the processing components of a probable solution for the Laplace Equation, using the algorithm proposed, is certainly a **Homogeneous** one.

(4) *Compare the problem specification with the architectural pattern's Problem section.* An Architectural Pattern that directly copes with the categories of domain parallelism and the homogeneous nature [6, 8, 9] of processing components is the **Communicating Sequential Elements (CSE) pattern** [7, 9]. In order to verify that this architectural pattern actually copes with the Laplace Equation problem, let us compare the problem description with the Problem section of the CSE pattern. From the CSE pattern description, the problem is defined as [7, 9]:

> *"A parallel computation is required that can be performed as a set of operations on regular data. Results cannot be constrained to a one-way flow among processing stages, but each component executes its operations influenced by data values from its neighboring components. Because of this, components are expected to intermittently exchange data. Communications between components follow fixed and predictable paths".*

Observing the algorithmic solution for the Laplace Equation, it can be defined in terms of calculating the next position of the surface elements as ordered data. Each element is operated almost autonomously. The exchange of data or communication should be between neighboring elements of the surface. So, the CSE is chosen as an adequate solution for the Laplace Equation, and the architectural pattern selection is completed. The design of the parallel software system should continue, based on the Solution section of the CSE pattern.

- **Structure and dynamics**. Based on the information of the Communicating Sequential Elements architectural pattern, it is used here to describe the solution to the Laplace Equation in terms of this architectural pattern's structure and behavior.

(1) *Structure.* Using the Communicating Sequential Elements architectural pattern for the Laplace Equation, the same operation is applied simultaneously to obtain the next position values of each element. However, this operation depends on the partial results in its neighboring elements. Hence, the structure of the actual solution involves a regular, two-dimensional, logical structure, conceived from the surface of the original problem. Therefore, the solution is presented as a two-dimensional network of elements that follows the shape of the surface. Identical components simultaneously exist and process during the execution time. An Object Diagram, representing the network of elements that follows the two-dimensional shape of the surface and its division into elements, is shown in Figure 5.

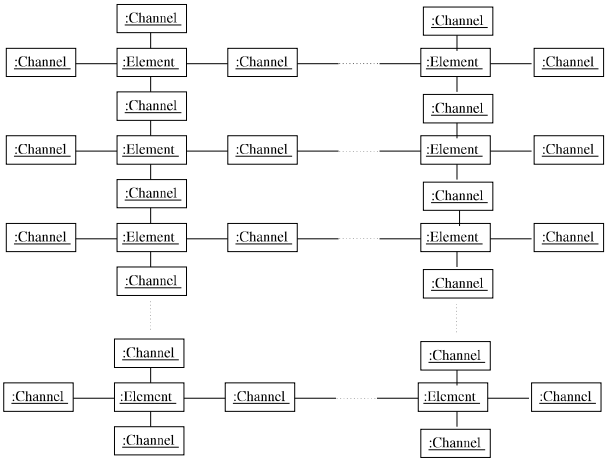


**Figure 5: Object Diagram of Communicating Sequential Elements for the solution to the Laplace Equation.**

(2) *Dynamics.* A scenario to describe the basic run-time behavior of the Communicating Sequential Elements pattern for solving the Laplace Equation is shown as follows. Notice that all the elements, as basic processing software components, are active at the same time. Every element performs the same position operation, as a piece of a processing network. However, it is diffcult to present in a single figure all the interactions carried out for the whole parallel computation. Thus, for the two-dimensional case here, Figure 6 only shows the interactions between only one element object, which communicates with its neighbors thorugh four dedicated channels.

The processing and communicating scenario is as follows:

– Initially, consider only a single **Element** object, **e(i,j)**. At first, it exchanges its local temperature value with its neighbors **e(i-1,j)**, **e(i+1,j)**, **e(i,j-1)**, and **e(i,j+1)** though the adequate communication **Channel** components. After this, **e(i,j)** counts with the different positions from its neighbors.

– The position operation is simultaneously started by the **e(i,j)** component and all the other components of the surface.

– In order to continue, all components iterate as many times as required, exchanging their partial position values through the available communication channels.

– The process repeats until each component has finished iterating, and thus, finishing the whole Laplace Equation computation.

(3) *Functional description of components.* This section describes each processing and communicating software components as participants of the Communicating Sequential Elements architectural pattern, establishing its responsibilities, input and output for solving the Laplace Equation.
   – **Element**. The responsibilities of an element, as a processing component, are to obtain the next local temperature from the temperature values it receives, and make available its own temperature value so its neighboring components are able to proceed.
   – **Channel**. The responsibilities of every channel, as a communication component, are to allow sending and receiving temperature values, synchronizing the communication activity between neighboring sequential elements. Channel components are developed as the main design objective of a following step, called "Communication Design", which is not addressed in this paper.

(4) *Description of the coordination.* The Communicating Sequential Elements pattern describes a coordination in which multiple **Element** objects act as concurrent processing software components, each one applying the same temperature operation, whereas **Channel** objects act as communication software component which allow exchanging temperature values between sequential components. No temperature values are directly shared among **Element** objects, but each one may access only its own private temperature values. Every **Element** object communicates by sending its temperature value from its local space to its neighboring **Element** objects, and receiving in exchange their temperature values. This communication is normally asynchronous, considering the exchange of a single temperature value, in a one to one fashion. Therefore, the data representing the whole two-dimensional surface represents the regular logical structure in which data of the problem is arranged. The solution, in terms of a divided surface, is presented as a network that actually reflects this logical structure in the most transparent and natural form [7, 9].

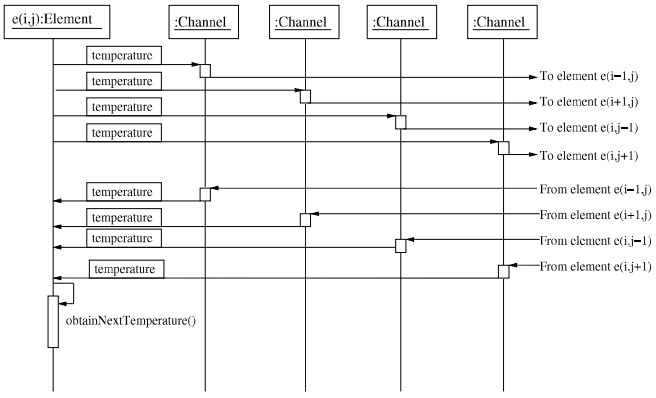


**Figure 6: Sequence Diagram of the Communicating Sequential Elements for communicating positions through channel components for the Laplace Equation.**

(5) *Coordination analysis.* The use of the Communicating Sequential Elements patterns as a base for organizing the coordination of a parallel software system for solving the Laplace Equation has the following advantages and disadvantages:

– **Advantages**
   (a) The order and integrity of temperature results is granted because each **Element** object accesses only its own local temperature value, and no other data is directly shared among components.
   (b) All **Element** objects have the same structure and behavior, which normally can be modified or changed without excessive effort.
   (c) The solution is easily structured in a transparent and natural form as a two-dimensional array of components, reflecting the logical structure of the two-dimensional surface in the problem.
   (d) All **Element** objects perform the same temperature operation, and thus, granularity is independent of functionality, depending only on the size and number of the elements in which the two-dimensional surface is divided. Changing the granularity is normally easy, by just adjusting the number of **Element** objects in which the surface is divided, thus obtaining a better resolution or precision.
   (e) The Communication Sequential Elements pattern can be easily mapped into the shared memory structure of the parallel platform available.

– **Liabilities**
   (a) The performance of a parallel application for solving the Laplace Equation based on the Communicating Sequential Elements pattern is heavily impacted by the communication strategy used. For the present example, the threads available in the parallel platform have to take care of a large number of **Element** objects, so each thread has to operate on a subset of the data rather than on a single value. Due to this, dependencies between data, expressed as communication exchanges, could be a cause of a slow down in the program execution.
   (b) For this example, load balancing is kept by allowing only a fixed number of **Element** objects per thread, which tends to be larger than the number of threads available. Nevertheless, if data would not be easily divided into same-size subsets, then the computational intensity varies on different

processors. Even though every processor is virtually equal to the others, maintaining the synchronization of the parallel application means that any thread that slows down should eventually catch up before the computation can proceed to the next step. This builds up as the computations proceeds, and could impacts strongly on the overall performance.

(c) Using synchronous communications implies a significant amount of effort required to get a minimal increment in performance. On the other hand, if the communications are kept asynchronous, it is more likely that delays would be avoided. This is taken into consideration in the next step, "Communication Design" (not described here).

# 4 IMPLEMENTATION

In this section, all the software components described in the Coordination Design step are considered for their implementation using the Java programming language. Once programmed, the whole system is evaluated by executing it on the available hardware platform, measuring and observing its execution through time, and considering some variations regarding the granularity.

Here, it is only presented the implementation of the coordination structure, in which the processing components are introduced, implementing the actual computation that is to be executed in parallel. Further design work is required for developing the channel as communication and synchronization components. Nevertheless, this design and implementation goes beyond the actual purposes of the present paper.

The distinction between coordination and processing components is important, since it means that, with not a great effort, the coordination structure may be modified to deal with other problems whose algorithmic and data descriptions are similar to the Laplace Equation, such as the Poisson Equation [3].

## 4.1 Coordination

Considering the existence of a class `Channel` for defining the communications between **Element** objects, the Communicating Sequential Elements architectural pattern is used here to implement the main Java class of the parallel software system that solves the Laplace Equation. The class `Element` is presented as follows. This class represents the Communicating Sequential Elements coordination for the Laplace Equation example.

```
class Element implements Runnable{
   private static int M = 65536, N = 65536, iterations = 10;
   private static Channel[][][] element = null;
   private int i = -1;
   private int j = -1;
   public Element(int i, int j){
      this.i = i;
      this.j = j;
      new Thread(this).start();
   }
   public void run(){
      double temperature, received, total;
      temperature = random(10*M);

      for (int iter = 0; iter < iterations; iter++) {
        // Send local temperatures to neighbors
        if (i < M-2 && j > 0 && j < N-1) send(element[i+1][j][0], temperature);
        if (i > 1   && j > 0 && j < N-1) send(element[i-1][j][1], temperature);
        if (j < N-2 && i > 0 && i < M-1) send(element[i][j+1][2], temperature);
        if (j > 1   && i > 0 && i < M-1) send(element[i][j-1][3], temperature);
        total = 0.0;
        // Receive temperature from neighbors
        if(i > 0 && j > 0 && i < M-1 && j < N-1){
           for(int x = 0; x < 4; i++){
              received = receive(element[i][j][x]);
              total += received;
           }

        }
       // Insert processing here
      }
   }
   public static void main(String[] args){
      segment = new Channel[M][N][2];
      for(int m = 0; m < M; m++){
         for(int n = 0; n < N; n++){
            for(int p = 0; p < 4; p++){
               element[m][n][p] = new Channel();
            }
         }
      }
      for(int m = 0; m < M; m++){
         for(int n = 0; n < N; n++){
            new Element(m,n);
         }
      }
      System.exit(0);
   }
}
```

This class only creates two adjacent, two-dimensional arrays of `Channel` components and `Element` components, which represents the coordination structure of the whole parallel software system, developed for executing on the available parallel hardware platform. `Channel` components are used for exchanging temperature values between neighboring `Element` components, each one first sending its own temperature value (which is an asynchronous, non-blocking operation), and later retrieving the temperature values of the four neighboring surface components. Using this data, now it is possible to sequentially process to obtain the new temperature of the present component. This communication-processing activity repeats as many times as iterations defined.

## 4.2 Processing components

At this point, all what properly could be considered "parallel design and implementation" has finished: data is initialized (here, randomly, but it can be initialized with particular temperature values) and distributed among a collection of `Element` components. It is now the moment to insert the sequential processing which corresponds to the algorithm and data description found in the Problem Analysis, This is done in the class `Element`, where it is commented `Insert processing here`, by simply adding the following code, and considering the particular declarations for its computation:

```
temperature = total/4;
```

The simple, sequential Java code allows that each `Element` component obtains a local temperature based on the Laplace

Equation. Modifying this code implies modifying the processing behavior of the whole parallel software system, so the class `Element` can be used for other parallel applications, as long as they are two-dimensional and execute on a shared memory parallel computer.

## 5 SUMMARY

The Architectural Patterns for Parallel Programming are applied here along with a method in order to show how to apply an architectural pattern that copes with the requirements of order of data and algorithm 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 architectural pattern. Moreover, the application of the Architectural Patterns for Parallel Programming and the method for selecting them is proposed to be used during the Coordination 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.

## 6 ACKNOWLEDGEMENTS

The author would like to thank Christian Kohls, my shepherd for EuroPLoP 2018, for his valuable comments, as well as the attendants to the Writers' Workshop group, for their greatly helpful suggestions.

## REFERENCES

[1] P. Brinch-Hansen *Distributed Processes: A Concurrent Programming Concept.*, Communications of the ACM, Vol.21, No. 11, 1978.
[2] E.W. Dijkstra *Co-operating Sequential Processes*, In Programming Languages (ed. Genuys), pp.43-112, Academic Press, 1968.
[3] J. Gazdag and H.H. Wang *Concurrent computing by sequential staging of tasks*, In IBM Systems Journal, pp.646-660, IBM Co., 1989.
[4] S. Hartley *Concurrent Programming. The Java Programming Language.*, Oxford University Press Inc., 1998.
[5] C.A.R. Hoare *Communicating Sequential Processes.* Communications of the ACM, Vol.21, No. 8, August 1978.
[6] 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.
[7] 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.
[8] J.L. Ortega-Arjona *Architectural Patterns for Parallel Programming: Models for Performance Evaluation*, PhD Thesis, Department of Computer Science, University College London, UK, 2007. `http://www.sigsoft.org/phdDissertations/theses/JorgeOrtega.pdf`
[9] J.L. Ortega-Arjona *Patterns for Parallel Software Design*, John Wiley & Sons, 2010.
[10] Sun Microsystems. *Sun SPARC Enterprise T5120 Server.* `http://www.sun.com/servers/coolthreads/t5120/`.