Applying Architectural Patterns for Parallel Programming: Solving a Matrix Multiplication

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

ABSTRACT

The Architectural Patterns for Parallel Programming is a set of patterns along with a method for designing the coordination of parallel software systems. Their application takes as input: *(a)* the available parallel hardware platform, *(b)* the available parallel programming language, and *(c)* the analysis of the problem as an algorithm and data. This paper presents the application of the architectural patterns within the method for solving the Matrix Multiplication. The method takes information from the problem analysis, selects an architectural pattern for the coordination, and provides some elements about its implementation.

CCS CONCEPTS

• **Software and its engineering → Designing software**.

KEYWORDS

Architectural Patterns, Matrix Multiplication, Parallel Software Design

ACM Reference Format:
Jorge L. Ortega-Arjona. 2021. Applying Architectural Patterns for Parallel Programming: Solving a Matrix Multiplication. In *European Conference on Pattern Languages of Programs (EuroPLoP'21), July 7–11, 2021, Graz, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3489449.3490011

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 consistent with the original research by E.W. Dijkstra [4], C.A.R. Hoare [7], P. Brinch-Hansen [2], and many others, who established the main basis for parallel programming. Specifically, obtaining a parallel program from an algorithmic description is the main objective of the area of Parallel Software Design [17].

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.
EuroPLoP'21, July 7–11, 2021, Graz, Austria
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8997-6/21/07…$15.00
https://doi.org/10.1145/3489449.3490011

1.1 Architectural Patterns for Parallel Programming

The *Architectural Patterns for Parallel Programming* are fundamental organizational descriptions of common top-level structures observed in parallel software systems [10, 16, 17]. They specify properties and responsibilities of their sub-systems, and the particular form in which they are assembled together.

The most important step in designing a parallel program is to think carefully about its overall structure. The Architectural Patterns for Parallel Programming provide descriptions about how to organize a parallel program, having the following advantages [16, 17]:

(1) The Architectural Patterns for Parallel Programming (as any Software Pattern) provide a description that links a problem statement (in terms of an algorithm and the data to be operated on) with a solution statement (in terms of an organization structure of communicating software components).
(2) The partition of the problem to solve is a key for the success or failure of a parallel program. Hence, the Architectural Patterns for Parallel Programming have been developed and classified based on the kind of partition applied to the algorithm and/or the data present in the problem statement.
(3) As a consequence of the previous two points, the Architectural Patterns for Parallel Programming can be selected depending on characteristics found in the algorithm and/or data, which drive the selection of a potential parallel structure by observing and studying the characteristics of order and dependence among instructions and/or datum.
(4) The Architectural Patterns for Parallel Programming introduce parallel structures as forms in which software components can be assembled or arranged together, considering the different partitioning ways of the algorithm and/or data.

Nevertheless, even though the Architectural Patterns for Parallel Programming have these advantages, they also present the disadvantage of not describing, representing, or producing a complete parallel program in detail. Anyway, the Architectural Patterns for Parallel Programming are proposed as a way of helping a software designer to select a parallel structure as a starting point when designing a parallel program. For a complete exposition of the Architectural Patterns for Parallel Programming, refer to [10, 17], and futher work on each particular architectural pattern in [11–15].

1.2 Parallel Software Design

Parallel Software Design proposes forms, descriptions, and programming techniques to solve the parallelization of a problem

described as an algorithm and data. The research provides forms to organize parallel software as relatively independent parts which attempt to efficiently make use of multiple processors. As stated before, the goal is to obtain a parallel program from an algorithm and data description. Nevertheless, designing parallel programs can be frustrating [17]:

(1) 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?
(2) The overheads involved in synchronization among multiple processors may actually reduce the performance of a parallel program. How to anticipate and mitigate this problem?
(3) Like many performance improvements, parallelizing increases the complexity of a program. How to manage such a complexity?

These are difficult problems: yet, we do not know how to efficiently solve an arbitrary problem for a parallel system of arbitrary size. Hence, Parallel Software Design, at its actual stage of development, cannot offer universal solutions, but tries to provide some simple forms to get started [17].

Nevertheless, by sticking with some common parallel "coordination structures", it is possible to avoid a lot of errors and aggravation. Many approaches to Parallel Software Design have been presented up to date, proposing organizational descriptions of top-level, coordination structures observed in parallel programming. Some of these descriptions are: *Outlines of the Program* [3], *Programming Paradigms* [8], *Parallel Algorithms* [6], *High-level Design Strategies* [9], and *Paradigms for Process Interaction* [1]. All these descriptions provide common overall coorcination structures in parallel programming (such as, for example,"master-slave", "pipeline", "workpile", and others) that represent assemblies of parallel software components which are allowed to simultaneously execute and communicate. Furthermore, these descriptions support the design of parallel programs since all of them introduce coordination structures or forms that such assemblies exhibit. All these forms are the base for the *Architectural Patterns for Parallel Programming* [10–15], as a Software Patterns approach for parallel programming. These architectural patterns attempt to save the transformation "jump" between algorithm and program.

2 PROBLEM ANALYSIS – THE MATRIX MULTIPLICATION

The present paper demonstrates the application of Architectural Patterns for Parallel Programming for designing a coordination structure to solves the Matrix Multiplication. The objective is to show how an architectural pattern can be selected, so it copes with the functionality and requirements present in this problem.

2.1 Problem Statement

The matrix multiplication is a common known mathematical operation. The algorithm is relatively simple, although as it is discussed later, it may take a long time to compute if the size of the involved matrices is large enough. Consider, for example, the multiplication of two square matrices (Figure 1). Both matrices have been chosen to be square just for sake of simplicity.

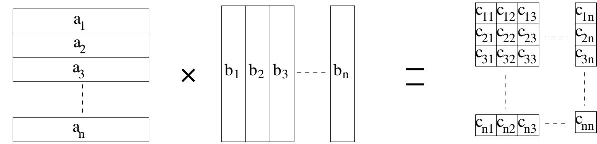

Figure 1: Multiplying two square matrices.

Notice that each one of the elements of the product matrix is obtained from applying a dot product between each one of the rows of the first matrix and each one of the columns of the second matrix. This is:

$$c_{ij} = \mathbf{a}_i \cdot \mathbf{b}_j = \sum_{k=0}^{n} a_{ik} \times b_{kj} \tag{1}$$

where:

- c_{ij} represents each element of the product matrix,
- $\mathbf{a}_i$ is an horizontal vector representing each row of the first matrix,
- $\mathbf{b}_j$ is a vertical vector representing each column of the second matrix,
- a_{ik} represents each element of the ith row, and
- b_{kj} represents each element of the jth column.

2.2 Specification of the Problem

Analyzing the problem, it is noticeable that a sequential process can perform all computations in $O(n^3)$ basic steps (since each n^2 entries of the result matrix requires n multiply-add operations). Suppose this numerically: the multiplication of two square matrices with, say, $n = 65,536$, may be solved in about $281,474,976,710,656$ time steps.

Moreover, naive changes to the requirements (such as, for instance, the size of the matrices), which are normally requested when performing this kind of processes, produce drastic increments of the number of required operations, which affects the time to calculate such a numerical solution.

- *Problem Statement.* A *Matrix Multiplication* of two relatively large matrices can be obtained in a more efficient way by:
  (1) using a group of software components that exploit the independent dot products, and
  (2) allowing each software component to simultaneously work on one or several dot products.

  A parallel version of this simultaneously computes products between all row-column pairs in $O(n)$, for a certain large n. The objective is to obtain a result in the best possible time-efficient way. Two parallel solutions based on this approach for matrix multiplication are the Fox Algorithm and the Cannon Algorithm [5].

- *Descriptions of the data and the algorithm.* The parallel program that carries out the matrix mutiplication takes as its

input two matrices: the first divided into rows, and the second divided into columns. Since all the dot products can be simultaneously performed, the only issue is to obtain the result matrix by ordering each one of its elements.

Hence, every dot product could be obtained independently from the other dot products:

```
class MatrixMultiply {
    private static int L = 0, M = 0, N = 0;
    private static double[][] a = null,
        b = null, c = null;
    ...
    public double dotProd(double[] a, double[] b) {
        double innerProduct = 0.0;
        for (int m = 0; m < M; m++)
            innerProduct += a[l][m]*b[m][n];
        return innerProduct;
     }
        ...
}
```

3 COORDINATION DESIGN

Here, the architectural patterns for parallel programming [10, 16, 17] are applied along with the the information from the problem analysis, in order to propose an architectural pattern for developing a coordination structure that carries out a parallel Matrix Multiplication.

3.1 Specification of the System

Based on the problem description and algorithmic solution presented in the previous section, the procedure for proposing an architectural pattern for a parallel solution to the Matrix Multiplication is presented as follows [17]:

(1) *Analyze the design problem and obtain its specification.* From analyzing the problem description and the algorithmic solution, a Matrix Multiplication yields a group of dot products that can be simultaneously performed on different vectors, which can be distributed and independently operated.
(2) *Select the category of parallelism.* Observing each dot product, the parallel solution should operate on different data while distributing such data. Hence, the solution description implies the category of **Activity Parallelism** [12, 13, 16, 17].
(3) *Select the category of the nature of the processing components.* Also, from the description of the solution, all dot products represent the same algorithm. Thus, the nature of the processing components of a probable solution for the Matrix Mutiplication is a **Homogeneous** one [12, 13, 16, 17].
(4) *Compare the problem specification with the architectural pattern's Problem section.* An Architectural Pattern that directly copes with the categories of activity parallelism and the homogeneous processing components is the **Manager-Workers (MW) pattern** [15–17]. Let us compare the problem description with the Problem section of the MW pattern [13, 16, 17]:

   'The same operation is required to be repeatedly performed on all the elements of some ordered data. Data can be operated without a specific order. However, an important feature is to preserve the order of data. If the operation is carried out serially, it should be executed as a sequence of serial jobs, applying the same operation to each datum one after another. Generally, performance as execution time is the feature of interest, so the goal is to take advantage of the potential simultaneity in order to carry out the whole computation as efficiently as possible'.

   Observing the algorithmic solution for the Matrix Multiplication, it can be defined in terms of a dot product performed over a row vector and a column vector. The information about which row and which column is important, since this actually refers to the position of the inner product within the result matrix. Each dot product can be operated completely and autonomously. The data or communication should be between an organizing component and several processing components, distributing the information of all the rows and columns so each dot product can be obtained by each processing component. So, the MW is chosen as an adequate solution for the Matrix Multiplication, and the architectural pattern selection is completed. The design of the parallel software system should continue, based on the Solution section of the MW pattern.

3.2 Functional description of components

Each processing and communicating software components, as participants of the Manager-Workers architectural pattern, is designed considering its responsibilities, and input/output when solving the Matrix Multiplication [13, 16, 17].

- **Manager**. The responsibilities of a manager are to create a number of workers, to distribute work among them, to start up their execution, and to assemble the overall matrix result from the sub-results from the workers.
- **Worker**. The responsibility of each worker is to seek for two vectors, to perform the dot product, and to return a scalar result.

3.3 Structure and dynamics

The information of the Manager-Workers architectural pattern is used to describe the solution to the Matrix Multiplication in terms of this architectural pattern's structure and behavior [13, 16, 17].

(1) *Structure.* Using the Manager-Worker architectural pattern for a Matrix Multiplication, each row and column pair is distributed by the manager, and operated by workers as conceptually-independent components. Each worker simultaneously performs a dot product. An object diagram representing the manager and worker components on which the bodies information is distribuited is shown in Figure 2.

   Notice that this organization effectively allows to distribute each row and column among worker components, as previously described in the problem analysis, so each dot product can be computed independently from the others.
(2) *Dynamics.* A typical scenario is used here to describe the basic run-time behavior of this pattern when applied to the

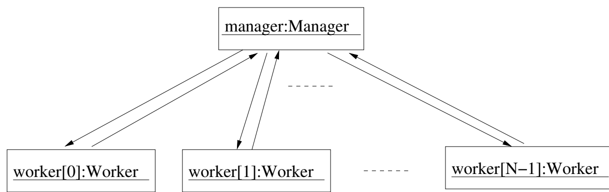

Figure 2: Object diagram of the Manager-Workers pattern applied for solving the Matrix Multiplication.

Matrix Multiplication. All components, whether manager or workers, are active at the same time, distributing and processing the information of different rows and columns, and assembling an overall resulting matrix as described in Figure 3:

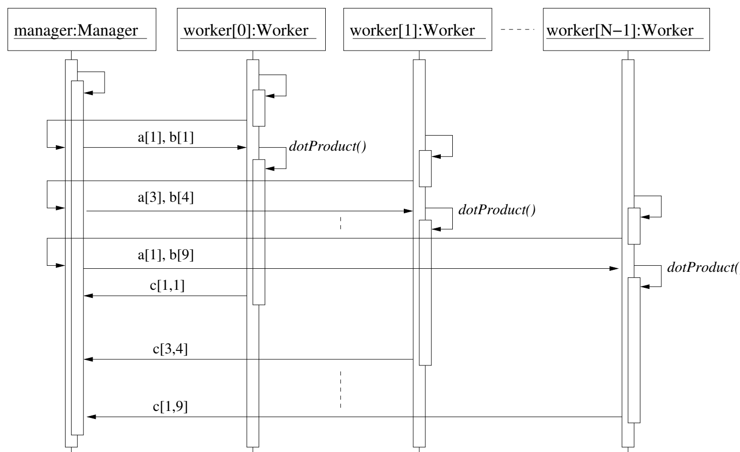

Figure 3: Sequence diagram of the Manager-Workers pattern for solving the Matrix Multiplication.

- All participants are created, and wait until two matrices `a` and `b` are provided to the manager. When such data is available to the manager, this distributes it, sending row-column pairs `a[i]` and `b[j]` by request to each waiting worker.
- Each worker receives a row-column pair. Notice that the *i*-th worker is associated with obtaining the dot product with the two vectors, and producing an scalar value `c[i,j]` as part of the result matrix `c`. Each operation is independent of the operations of other workers. Once all results are received by the manager, each worker may request again for more work representing another row-column pair, and the process repeats.
- The manager is usually replying to requests from the workers or receiving their partial results. Once all row-column pairs have been processed, the manager assembles a total result from the partial results and the program finishes. Any non-serviced requests of data from the workers are ignored.

3.4 Description of the coordination

The Manager-Workers architectural pattern uses activity parallelism to execute the *Matrix Multiplication*, allowing the simultaneous existence and execution of more than one worker components through time. Each one of these instances at the same time obtain a dot product for each row-column pair. In a parallel system like this, the *Matrix Multiplication* involves the distribution and execution of data. Each processing starts by distributing the row-column data among all workers, and finish only when all workers provide the manager with the result of each dot product.

3.5 Coordination analysis

The use of the Manager-Workers pattern as a base for organizing the coordination of a parallel software system for solving the *Matrix Multiplication* has the following advantages and disadvantages:

- **Advantages**

(1) The order and integrity of the result matrix is preserved and granted due to the defined behavior of the manager component. The manager takes care of what part of both matrices has been operated on by which worker, and what remains to be obtained by the rest of the workers.

(2) An important characteristic of the Manager-Workers pattern is due to the independent nature of operations that each worker performs. Each worker requests for a row-column pair during execution and obtains a dot product. Such an independence makes that the structure presents a natural load balance, and easily scale for larger matrices.

(3) Synchronization is simply achieved because communications are restricted to only between manager and each worker. The manager is the component in which the synchronization is stated.

(4) Using the Manager-Worker pattern, the parallelizing task is relatively straightforward, and it is possible to achieve a respectable performance. If designed carefully, the Manager-Worker pattern enables the performance to be increased without significant changes.

- **Liabilities**

(1) The Manager-Workers systems may present poor performance if the size of matrices is excesively large. In such a case, workers may remain idle for periods of time while the manager is busy trying to serve all their requests. Granularity should be modified in order to balance the amount of work, by allowing that more than one dot product is operated by each worker.

(2) Manager-Worker architectures may also have poor performance if the manager activities – data distribution, receive worker requests, send data, receive partial results, and assembling the final result – take a longer time compared with the processing time of the workers. The overall performance depends mostly on the manager, so programming the manager should be done taking special consideration to the time it takes to perform its activities. A poor performance of the manager impacts heavily on the performance of the whole system.

(3) Many different issues must be carefully considered, such as strategies for work distribution, manager and worker

collaboration, and assemble of final result. In general, the issue is to find the right combination of worker number, active or passive manager, and data size in order to get the optimal performance, but experience shows that this still remains a research issue.

(4) Moreover, it is necessary to provide error handling strategies for failure of worker execution, failure of communication between the manager and workers, or failure to start-up parallel workers.

4 IMPLEMENTATION

Here, all software components described in the coordination design section 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, for the purposes of measuring and observing its execution through time.

Nevertheless, here it is only presented the implementation of the coordination, 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 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 Matrix Multiplication [6].

4.1 Coordination

In order to support the communication exchange of data, let us consider the class `Message` as follows, which establishes a serializable set of fields used to communicate data from the manager to the workes, as well as from each worker to the manager.

```
class Message implements Serializable {
    public int workerID = -1;
    public boolean containsResult = false;
    public double result = -1;
    public boolean containsWork = false;
    public int N = -1;
    public int inRow = -1;
    public int inColumn = -1;
    public double [] Row = null;
    public double [] Column = null;

    public Message(int workerID, boolean
        containsResult, double result, boolean
        containsWork, double Row[], double Column[],
        int inRow, int inColumn, int N) {
         this.workerID = workerID;
         this.containsResult = containsResult;
         this.result = result;
         this.containsWork = containsWork;
         this.Row = Row;
         this.Column = Column;
         this.inRow = inRow;
         this.inColumn = inColumn;
         this.N = N;
        }
    ...
}
```

The first element of a `Message` type is `workerID` which allows to identify which worker is to perform the dot product for the data present in this message. Next, two variables, `containsResult` and `result`, represent the information from any worker to the manager; `containsResult` states that this message has a result from the dot product of the sending worker, which is the scalar value stored in the variable `result`. Finally, the next five fields represent the information from the manager to the worker which is required to perform de dot product: `inRow` and `inColumn` identify the position within the first matrix and the second matrix respectively; both these are used later, when sending a result of the dot product to the manager, so this could place such result in the proper position within the product matrix. The two arrays `Row` and `Column` are the actual data vectors to dot product. And `N` is the length of the vectors, used in the dot product as a limit for the operation.

Now, the Manager-Workers architectural pattern is used here to implement the main Java class of the parallel software system that solves the *Matrix Multiplication*. The class `MatrixMultManager` is presented as follows. This class represents the Manager-Workers coordination for the *Matrix Multiplication*.

```
class MatrixMultManager extends MyObject {
    private static int N = 8; //size of square matrices
    private static double[][] a = null,
        b = null, c = null;
    private static int numProducts = 0;
    int numWorkers = 8;
    private static int portNum = 9999;
    ...
    public static void main(String[] args) {
        ...
        // create the remote workers
        if (numWorkers > 0) {
            ...
            for (int i = 0; i < numWorkers; i++)
                new MatrixMultWorker(i, er);
        }
        ...
        // send out all the "work" (initial configurations)
        // a dot product of row i column j
        Message m = null;
        int numResultsReceived = 0;
        for (int i = 1; i <= N*N; i++) {
            for (int j = 1; i <= N*N; j++) {
                ...
                m = (Message) r.serverGetRequest();
                if (m.containsResult) {
                    c[m.inRow][m.inColumn] = m.result;
                    numResultsReceived++;
                    r.serverMakeReply(new Message(-1, false,
                        0, true, a[i], b[j], i, j, N));
                    r.close();
                }
                // tally up the returning results
```

```
                while (numResultsReceived < N*N) {
                    ...
                    m = (Message) r.serverGetRequest();
                    if (m.containsResult) {
                        c[m.inRow][m.inColumn] = m.result;
                        numResultsReceived++;
                        r.serverMakeReply(new Message(-1,
                            false, 0, false, null, null,
                            0, 0, 0));
                        r.close();
                    }
                }
            }
        }
        ...
        System.exit(0);
    }
}
```

This class makes use of the class `Message` as the basic data communication structure. This class requires the creation of MatrixMultWorker as worker components, to perform the dot product, and which together with the class `MatrixMultManager` represent the coordination of the whole parallel software system, developed for executing on the available parallel hardware platform. Notice that both classes rely on a class `Rendezvous` for communication exchange, to send and receive messages. Nevertheless, this class goes beyond the scope of this paper. By now, let us suppose that such a class is available, so communications are performed just as indicated.

4.2 Processing components

At this point, all what properly could be considered "parallel design and implementation" has finished: data is initialized and distributed among a collection of worker components. It is now the moment to present the sequential processing which corresponds to the dot product found in the problem analysis. This is done in the class `MatrixMultWorker`, which considers the particular declarations for the *Matrix Multiplication* computation:

```
class MatrixMultWorker implements Runnable {
    private int N = -1;
    private String masterMachine = null;
        // Server where manager executes
    private int portNum = -1;
    ...
    private int id = -1;

    public MatrixMultWorker(int id,
        String masterMachine, int portNum) {
        ...
        this.id = id;
        this.masterMachine = masterMachine;
        this.portNum = portNum;
        ...
        new Thread(this).start();
    }

    public void run() {
        Message m = new Message(id, false, 0,
            false, null, null, 0, 0, 0);
        while (true) {
            ...
            m = (Message)
                r.clientMakeRequestAwaitReply(m);
            r.close();
            if (!m.containsWork) {
                if (masterMachine != null) {
                // remote workers
                    System.exit(0);
                } else return;
            }
        }
        N = m.N;
        int inRow = m.inRow;
        int inColumn = m.inColumn;
        double result = 0.0d;
        for (int i = 0; i < N; i++) result += a[i]*b[i];
        m = new Message(id, true, result, false, null,
            null, inRow, inColumn, 0);
        }
    }
    ...
    public static void main(String[] args) {
     new MatrixMultWorker(id, masterMachine, portNum);
    }
}
```

This simple, sequential Java code allows that each `MatrixMultWorker` component to obtain a local dot product from two row-column vectors, for the position provided. Modifying this code implies modifying the processing behavior of the whole parallel software system, so the class `MatrixMultManager` can be modified and used for other parallel applications, as long as they are independent computations, and execute on a cluster or a distributed memory parallel computer.

The utility of the coordination presented here goes beyond of a parallel *Matrix Multiplication*. By modifying the sequential processing section, each worker component is capable of processing other problems, such as the N- Queens problem [6].

5 SUMMARY

The architectural patterns for parallel programming are applied here along with a method for selecting them, in order to show how to select an architectural pattern that copes with the requirements of order of data and algorithm present in the *Matrix Multiplication*. 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 a distribution of work, executing on a distributed memory parallel platform.

ACKNOWLEDGMENTS

The author wishes to acknowledge the important contribution to the development of this paper to my shepherd for EuroPLoP 2021, Ruslan Batdalov, as well as all the participants of the attendants to the Writers' Workshop Group 3, for their invaluable comments.

REFERENCES

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

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

[3] K.M. Chandy, and S. Taylor *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Inc., Boston, 1992.

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

[5] G.C. Fox, S.W. Otto, and A.J.G. Hey *Matrix algorithms on a hypercube I: matrix multiplication.*, In Parallel Computing 4, pp.17-31, 1987.

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

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

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

[9] B. Lewis and D.J.. Berg *Multithreade Programming with Java Technology*, Sun Microsystems, Inc., 2000.

[10] 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.

[11] 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.

[12] J.L. Ortega-Arjona *The Shared Resource Pattern. An Activity Parallelism Architectural Pattern 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 Manager-Workers Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming*, Proceedings of the 9th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2004), Kloster Irsee, Germany, 2004.

[14] J.L. Ortega-Arjona *The Parallel Pipes and Filters Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming*, Proceedings of the 10th European Conference on Pattern Languages of Programming and Computing (EuroPLoP2005), Kloster Irsee, Germany, 2005.

[15] J.L. Ortega-Arjona *The Parallel Layers Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming*, Proceedings of the 6th Latin American Conference on Pattern Languages of Programming and Computing (SugarLoafPLoP2007), Porto de Galinhas, Pernambuco, Brasil, 2007.

[16] J.L. Ortega-Arjona *Architectural Patterns for Parallel Programming: Models for Performance Evaluation*, VDM Verlag, 2009.

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