Applying Architectural Patterns for Parallel Programming

The Fast Fourier Transform

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

Abstract

The Architectural Patterns for Parallel Programming is a system of patterns, related with a method for applying them for the development of the coordination of parallel software systems. These Architectural Patterns make use of input information: *(a)* the available parallel hardware platform, *(b)* the parallel programming language used by 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 an Architectural Pattern, along with the method, for developing a coordination to solve the Fast Fourier Transform. The method used here starts taking the information from the Problem Analysis, proposes an Architectural Pattern for designing the coordination, and provides some elements about its partial implementation.

1 Introduction

A parallel program is defined as *the specification of a set of processes executing simultaneously, and communicating among themselves in order to achieve a common objective* [19]. This definition is obtained from the original work by E.W. Dijkstra [6], C.A.R. Hoare [9], P. Brinch-Hansen [2], and many others, who have established the main basis for parallel programming. In general, the success of a parallel program is able to achieve –commonly, in terms of performance– is affected by three main factors [19]: *(a)* the hardware platform, *(b)* the programming language, and *(c)* the problem to solve.

Nevertheless, parallel programming still yields a hard problem to the software designer and programmer: up to date, we do not yet know how to efficiently solve an arbitrary problem of arbitrary size in parallel. Hence, parallel programming, at its actual stage of development, does not (cannot) offer universal solutions, but tries to provide some simple ways to get started. By sticking with some common parallel *patterns* of coordination, it is possible to avoid a lot of errors and aggravation.

Many approaches have been proposed, describing top-level patterns of coordination observed in many parallel programs. Some of these descriptions are: *Outlines of the Program* [4], *Programming Paradigms* [10], *Parallel Algorithms* [7], *High-level Design Strategies* [11], and *Paradigms for Process Interaction* [1]. These descriptions provide common overall patterns of coordination, such as for example,"master-slave", "pipeline", "work-pile", and others. They represent assemblies of parallel software components, which are allowed to simultaneously execute and communicate. Furthermore, these descriptions are expected to support the design of parallel programs, since all of them introduce common forms that such assemblies exhibit.

The Architectural Patterns for Parallel Programming [13, 14, 15, 16, 17, 18, 19, 20] represent a software patterns approach for designing the coordination of parallel programs. These Architectural Patterns attempt to save the transformation "jump" between algorithm description and program design and implementation. They are defined as *fundamental organizational descriptions of common top-level structures observed in parallel software systems* [13, 20], specifying properties and responsibilities of their sub-systems, and the particular form in which they are assembled together into a coordination.

Architectural Patterns allow software designers and developers to understand complex software systems in larger conceptual blocks and their relations, thus reducing the cognitive burden. Further, 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.

The most important step in designing a parallel program is to think carefully about its overall coordination. The Architectural Patterns for Parallel Programming provide descriptions about how to coordinate the components of a parallel program, having the following advantages [13, 14, 15, 16, 17, 18, 20]:

- The Architectural Patterns for Parallel Programming provide a description that connects a problem, in terms of an algorithm and the data to be operated on, with a solution, in terms of an organization or coordination of communicating software components.
- The partition of the problem 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.
- As a consequence of the previous two points, the Architectural Patterns for Parallel Programming can be proposed depending on characteristics found in the algorithm and/or data, which drive the selection of a potential parallel coordination, by observing and studying the characteristics of order and dependence among instructions and/or datum.

- The Architectural Patterns for Parallel Programming introduce coordinations as forms in which software components can be assembled or arranged together, and thus, execute simultaneously, considering the different partitioning ways of the algorithm and/or data.

Nevertheless, the Architectural Patterns for Parallel Programming also present the main disadvantage of not describing, representing, or producing a complete parallel program in detail. Other software patterns are still needed for achieving this. Anyway, the Architectural Patterns for Parallel Programming are proposed as a way of helping a software designer to select a parallel coordination as a starting point when designing a parallel program. For a complete exposition of the Architectural Patterns for Parallel Programming, refer to [13, 20], and further work on each particular architectural pattern in [14, 15, 16, 17, 18].

The present paper is organized as follows: Section 2 presents a brief Problem Analysis of the Fast Fourier Transform (FFT), in mathematical, algorithmical, and programming terms. The objective is to take an idea about what the parallel program has to achieve. Section 3 presents de Coordinatio Design, this is, the discussion from the information of the Problem Analysis in Section 2, driving the selection of an Architectural Pattern for Parallel Programming, proposed as the base coordination for solving the FFT in parallel. Section 4 presents some elements of the very implementation of this coordination in Java, attempting to expose the most important issues to consider during programming of the coordination. Finally, Section 5 presenta a summary of the aims and objectives of this paper.

2 Problem Analysis – The Fast Fourier Transform

The present paper attempts to demonstrate the application of the Architectural Patterns for Parallel Programming for designing the coordination of a parallel program that obtains the Fast Fourier Transform (FFT) [5]. The objective here is to show how such an Architectural Pattern can be selected and applied, so it deals with the functionality and requirements present in the FFT.

2.1 Problem Statement

2.1.1 The Fourier Transform and the Discrete Fourier Transform

The Fourier Transform is one of the best well known mathematical operations, applied in many fields of science and technology. It is commonly defined for a function in time $a(t)$ as:

$$A(f) = \int_{-\infty}^{\infty} a(t) e^{2\pi i f t} dt$$

where $i = \sqrt{-1}$. However, the Fourier Transform defines the frequency components of a continuous function. When a function is sampled and analyzed using a computer, it is necessary to obtain the Discrete Fourier Transform (DFT). Hence, supposing that function $a(t)$ is sampled at discrete points with intervals

of fixed length Δt, the DFT is obtained for an array of n sampled points that keep the values of $a(t)$:

$$a = [a_0 a_1 \dots a_{n-1}]$$

where $a_k = a(t_k)$ and $t_k = k\Delta t$ for $k = 0, 1, \dots, n-1$. The DFT, as an approximation of the Fourier Tranform $A(f)$ of function $a(t)$, is stored in another array A at n discrete points. Thus, the DFT array A is:

$$A = [A_0 A_1 \dots A_{n-1}]$$

where $A_j = A(f_j)$ and $f_j = \frac{j}{n\Delta t}$ for $j = 0, 1, \dots, n-1$. Each discrete frequency f_j is a multiple of f_1, the inverse of the total sampling time $n\Delta t$.

The relation between each element of the arrays a and A is approximated by a discrete sum:

$$A_j = \sum_{k=0}^{n-1} a(t_k) e^{2\pi i f_j t_k} \Delta t$$

This expression can be simplified, without loosing generality, by considering that $\Delta t = 1$, and introducing a complex number $w(n)$ that is the nth root of unity in the complex plane ($w(n)^n = e^{2\pi i} = 1$). Hence, the relation between A and a is simplified to:

$$A_j = \sum_{k=0}^{n-1} a_k w(n)^{jk} \quad j = 0, 1, \dots, n-1$$

2.1.2 The Fast Fourier Transform

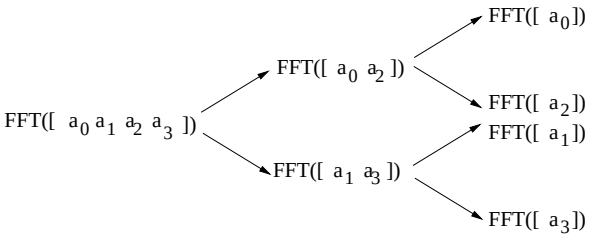
The FFT can be obtained as Cooley and Tukey proposed in 1965 [5]. If n is a power of two, the DFT can be computed by a FFT in $O(nlogn)$ time. Nevertheless, the complete description of FFT tends to be somehow long. Hence, for the purposes of this paper, FFT is described here using, for example, four points $a = [a_0 a_1 a_2 a_3]$. First, let us consider the n sampled points split into two halves of even and odd numbered points. Each half is then divided into two samples of one point each. Since the transform of a single point is the point itself, the recursion stops here. Now the FFT algorithm continues by combining the four FFTs of length 1 into two FFTs of length 2, and finally, into a single FFT of length 4. Both, the division and the combination, are shown in Figure 1.

Several further considerations may be taken, and other operations on the samples may be perfomed (like, for example, permutations, iterations, and some operations over arrays), in order to continue simplifying the obtention of the FFT. For our purposes, let us consider the following simplified and recursive sequential algorithm for the FFT in Java:

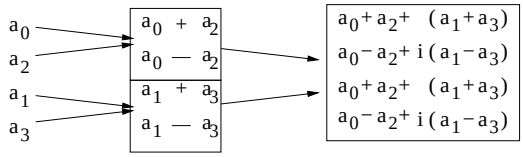
```
class FFT{
   private double [] a = null;
   private int first = -1;
   private int last = -1;
   public FFT(int n){
```

```
      ...
      first = a[0];
      last = a[n];
   }
   public void combine(int first, int last){
      const double pi = 3.1415926536d;
      int even,half,odd,k;
      complex w, wj, x;

      half = (last - first + 1) / 2;
      w = pair(Math.cos(pi/half), Math.sin(pi/half));
      wj = pair(1,0);
      for(k =0; k < half; k++){
         even = first + k;
         odd = even + half;
         x = product(wj,a[odd]);
         a[odd] = difference(a[even],x);
         a[even] = sum(a[even],x);
         wj = product(wj,w);
      }
   }
   public void FFT(double [] a, int first, int last){
      int middle;
      ...
      middle = (first + last) / 2;
      FFT(a,first,middle);
      FFT(a,middle+1,last);
      combine(a,first,last);
      }
   }
}
```



Dividing the samples



Combining the transforms

Figure 1: Dividing the samples and combining the transforms.

The run-time of this partitioning is $O(n)$, yielding an $O(n \log n)$ average run-time, and an $O(n^2)$ run-time for the worst case.

2.2 Specification of the Problem

From the previous section, considering an array of n `double` numbers, it is possible to analyze T_1 as the average sequential runtime required to solve the problem into a single node. Input and output of the array of n `double` numbers is performed in $O(n)$ time. The processing between input and output is carried out in $O(n \log n)$ time. Thus:

$$T_1 = n(a \log n + b)$$

where a and b are constants depending on communication and processing.

Taking into consideration this time analysis, solving the FFT problem on a sequential computer requires something like T_1 units of time. Let us suppose a numerical example: for an array with, for example, $n = 65,536$, it is required to solve about $1,048,576$ operations. Further, notice that naive changes to the requirements (which are normally requested when performing this kind of computations) produce drastic increments of the number of operations required, which at the same time affects the time required to calculate this numerical solution.

- *Problem Statement.* The FFT, for a relatively large number of array elements, can be computed in a more efficient way by:

  1. using a group of software components that exploit the hierarchical logical structure of the algorithm, and
  2. allowing each software component to simultaneously operate on its local array.

  The objective is to obtain a result in the best possible time-efficient way.

- *Descriptions of the data and the algorithm.* The whole parallel program that performs the FFT takes as its input the very array and its bounds, which can be received through a communication call.

```
class Node implements Runnable{
   ...
   private int first = -1;
   private int last = -1;
   private channel c = null;
   private double [] a = null;
   ...
   public void run(){
      ...
      receive(c,first,last);
      for(int k = first; k < last; k++){
         receive(c, a[k]);
      }
      ...
      FFT(a,first,last);
```

```
      ...
      for(int k = first; k < last; k++){
         send(c, a[k]);
      }
      ...
   }
}
```

Once it has received its part of the array from channel `c`, each **Node** object is able to compute a local FFT as a single thread. When the local result is obtained, it sends the result to the neighboring nodes again through channel `c`.

- *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-Core UltraSPARC 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 [7, 8].

- *Quantified requirements about performance and cost.* This application example has been developed as a course exercise and for experimenting with the platform, testing its functionality in time, and how it maps with a parallel application. So, the main objective is simply to 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 platform for further later studies.

3 Coordination Design

Here, the Architectural Patterns for Parallel Programming [13, 19, 20] are applied, along with the information from the Problem Analysis, in order to propose an Architectural Pattern for developing a coordination structure that performs a parallel FFT.

3.1 Specification of the System

This section describes the basic operation of the parallel software system, considering the information presented in the Problem Analysis (Section 2) about the parallel system and its programming environment. Based on the problem description and algorithmic solution presented in the previous section, the method for proposing an Architectural Pattern for a parallel solution to the FFT problem is presented as follows [20]:

1. *Analyze the design problem and obtain its specification.* Analyzing the problem description and the provided algorithmic solution, it is noticeable that the FFT yields a hierarchical structure of operations. Such an structure is based on dividing the data of the original array into two

sub-arrays. This division is carried out over and over, until obtaning the FFT of an array with a single element becomes a trivial operation. Only then, the algorithm continues combining the data, now going back in the hierarchical structure.

2. *Select the category of parallelism.* Observing the form in which the algorithmic solution divides the problem, it is clear that the algorithm partitions the FFT operation into sub-FFT operations. So, these should be executed simultaneously on different array elements. Hence, the algorithmic solution description implies the category of **Functional Parallelism**.

3. *Select the category of the nature of the processing components.* Also, from the algorithmic description of the solution, it is clear that each FFT operation is obtained using exactly the same algorithm. Thus, the nature of the processing components of a probable solution for the FFT, using the algorithm proposed, is certainly **Homogeneous**.

4. *Compare the problem specification with the architectural pattern's Problem section.* An Architectural Pattern that directly copes with the categories of functional parallelism and the homogeneous nature of processing components is the **Parallel Layers (PL) pattern** [18, 19, 20]. In order to verify that this architectural pattern actually copes with the FFT problem, let us compare the problem description with the Problem section of the PL pattern. From the PL pattern description, the problem is defined as [18, 19, 20]:

   > 'An algorithm is composed of two or more simpler sub-algorithms, which can be divided into further sub-algorithms, and so on, recursively growing as an ordered tree-like structure until a level in which the sub-parts of the algorithm are the simplest possible. The order of the tree structure (algorithm, sub-algorithms, sub-sub-algorithms, etc.) is a strict one. Nevertheless, data can be divided into data pieces which are not strictly dependent, and thus, can be operated on the same level in a more relaxed order. If the whole algorithm is performed serially, it could be viewed as a chain of calls to the sub-algorithms, evaluated one level after another. Generally, performance as execution time is the feature of interest. Thus, how do we solve the problem (expressed as algorithm and data) in a cost-effective and realistic manner?'.

   Observing the algorithmic solution for the FFT, it can be defined in terms of an algorithm composed of two sub-algorithms, which is divided over and over, recursively growing as a tree-like structure. Each FFT sub-algorithm performs completely and autonomously. The exchange of data or communication should be between a root component and two children components, dividing the array into two sub-arrays. So, the PL is chosen as an adequate solution for the FFT problem, and the architectural pattern selection is completed. The design of the parallel software system should continue, based on the Solution section of the PL pattern.

3.2 Structure and dynamics

The information of the Parallel Layers pattern is used here to describe the design solution to the FFT, in terms of this Architectural Pattern's structure and behavior [18, 19, 20].

1. *Structure.* Using the Parallel Layers pattern for FFT, different data is operated by conceptually-independent components, ordered in the shape of layers. Each layer, as an implicit different level of abstraction, is composed of several components that perform the same FFT operation. To communicate, layers use calls, referring to each other. FFT is performed by different groups of functionally related layer components. These components simultaneously exist and process. An object diagram, representing the tree of layer components on which the FFT shape is mapped for dividing the operations is shown in Figure 2.

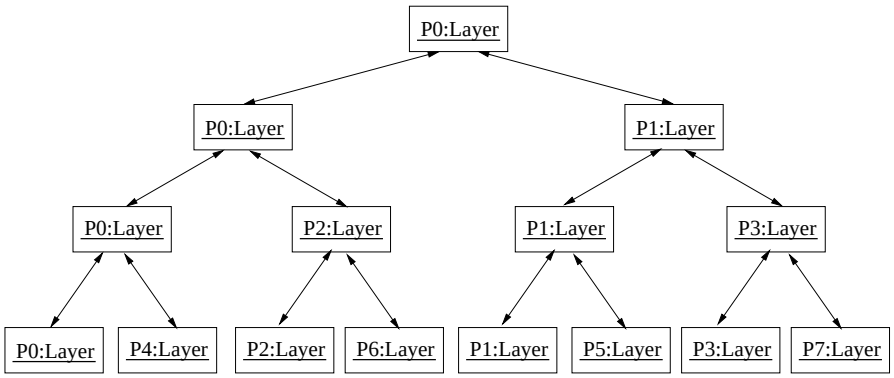


Figure 2: Object diagram of the Parallel Layers pattern applied for solving the FFT.

   Notice that this coordination effectively allow to distribute data among layer components, as described by the FFT algorithm, in the Problem Analysis (Section 2).

2. *Dynamics.* A typical scenario of three levels is used to describe the basic run-time behavior of this pattern, when applied to the FFT of n `double` numbers. All layer components are considered active at the same time, accepting a function call with its assigned `double` numbers, distributing them through two function calls to its child components in lower level layers, and once all `double` numbers are completely distributed, applying a FFT operation to the returned results from the child components.

   The PL pattern is used here to repeatedly perform a parallel FFT, as series of tree ordered FFT operations, as described in Figure 2. The parallel execution follows the description of the FFT (Figure 3):

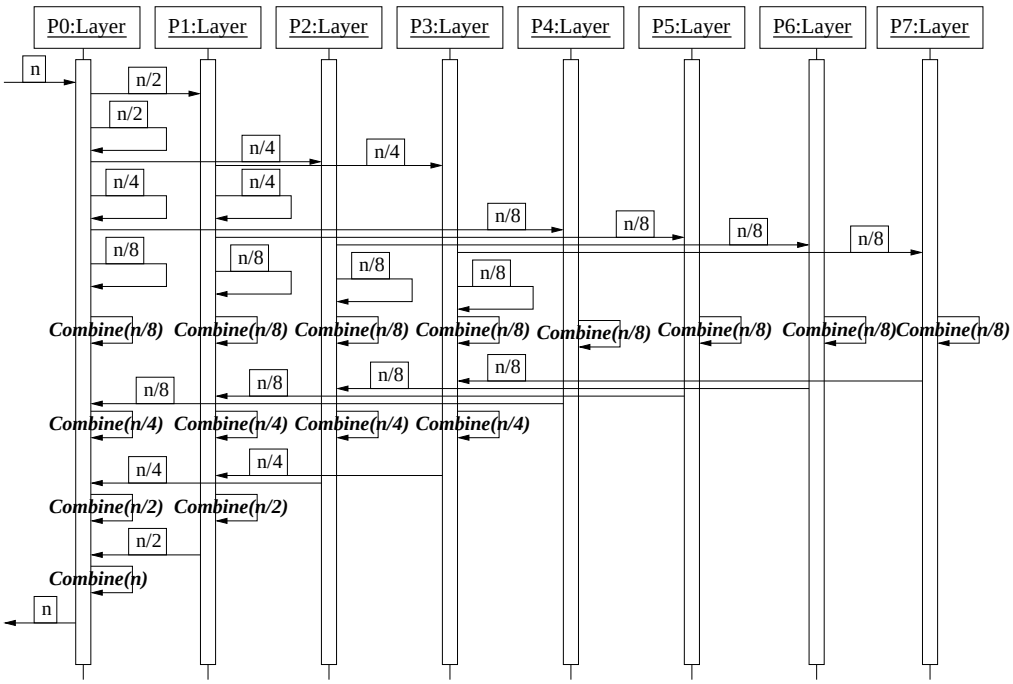


Figure 3: Sequence diagram of the Parallel Layers pattern for solving the FFT.

3.3 Functional description of components

Each processing and communicating software components are described as participants of the Parallel Layers architectural pattern, establishing its responsibilities, input, and output for solving the FFT.

- **Layer component.** The responsibilities of a layer component here are to allow the creation of the tree structure for solving the FFT. Hence, it provides a service to the layer component above, receiving a function call when distributing data, while delegating further distribution to the two or more layer components below. This allows the top-down flow of data, by receiving data from the layer component above, distributing it to the layers components below. Also, the layer component allows the bottom-up flow of results, by receiving partial results from the components below, and making its result available to the layer component above. Moreover, each layer component is able to independently perform a FFT operation over the results received from the components below it, making it easy to execute in parallel layer components belonging to the same layer [18, 19, 20].

3.4 Description of the coordination

The Parallel Layers pattern makes use of functional parallelism to execute the FFT, allowing the simultaneous existence and execution of more than one instance of a layer component through time. Each layer simultaneously divides the

data, for further applying the FFT. In a layered system like this, FFT involves the execution of FFT in several layers. These operations are usually triggered by a call, and data is vertically shared among layers in the form of arguments for these function calls. During the execution of the operations in each layer, usually the higher layers have to wait for the results from lower layers. However, if each layer is represented by more than one component, they can be executed in parallel. Therefore, at the same time, several ordered sets of operations can be carried out by the same system, by allowing several FFT overlapped in time.

3.5 Coordination analysis

The use of the Parallel Layers pattern as a base for organizing the coordination of a parallel software system for solving the FFT has the following advantages and disadvantages:

- **Advantages**

  1. The Parallel Layers pattern, as the original Layers pattern, is based on increasing levels of complexity. This allows the partitioning of the computation of a complex problem into a sequence of incremental, simple operations [23]. Each layer is presented as multiple components executing in parallel, performing the computation several times, enhancing performance.
  2. Changes in one layer do not propagate across the whole system, as each layer interacts at most with only the layers above and below, that can be affected. Furthermore, standardizing the interfaces between layers usually confines the effect of changes exclusively to the layer that is changed. [22, 23].
  3. Layers support reuse. If a layer represents a well-defined operation, and communicates via a standardized interface, it can be used interchangeably in multiple contexts. A layer can be replaced by a semantically equivalent layer without great programming effort [22, 23].
  4. Granularity depends on the level of complexity of the operation that the layer performs. As the level of complexity decreases, the size of the components diminishes as well.
  5. Due to several instances of the same computation are executed independently on different data, synchronization issues are restricted to the communications within just one computation. Relative performance depends only on the level of complexity of the operations to be computed, since all components are active [21].

- **Liabilities**

  1. Many times, a layered system is not as efficient as a structure of communicating components. If services in upper layers rely heavily on the lowest layers, all data must be transferred through the system. Also, if lower layers perform excessive or duplicate work, there is a negative influence on the performance. In certain cases, it is possible to consider a Pipe and Filter architecture instead [22].

2. If an application is developed as layers, a lot of effort must be expended in trying to establish the right levels of complexity, and thus, the correct granularity of different layers. Too few layers do not exploit the potential parallelism, but too many introduce unnecessary communications. The granularity and operation of layers is difficult, but related with the performance quality of the system [22, 23, 12].
3. If the level of complexity of the layers is not correct, problems can arise when the behavior of a layer is modified. If substantial work is required on many layers to incorporate an apparently local modification, the use of Layers can be a disadvantage [22].

4 Implementation

All the software components described in the coordination design section are considered for their implementation, using the Java programming language. 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 is required for developing the communication and synchronization components. Moreover, 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 FFT, such as the Quicksort [3].

4.1 Coordination

The Parallel Layers pattern is used here to implement the main Java class of the parallel software system that solves the FFT problem. The class `ParallelFFT` is presented as follows. This class represents the Parallel Layers coordination for the FFT problem.

```
class ParallelFFT{
    ...
    private static BinaryTree<ArrayList<Double>> tree;
    private Node<ArrayList<Double>> rootNode;
    private ArrayList<Double> combination;
    ...
    public ParallelFFT(Node <ArrayList<Double>> rootNode){
        this(rootNode,new Vector(), new Vector());
    }
    public Node<ArrayList<Double>> getOrdered(){
        return rootNode;
    }
    ...
    private ArrayList<Double> divide(ArrayList<Double> cont, boolean half1){
        ArrayList<Double> part = new ArrayList<Double>();
        if(half1){
            for(int x = 0; x < cont.size()/2.0; x++){
                part.add(cont.get(x));
            }
        }
```

```
        else{
            for(int x = cont.size()-1; x >= cont.size()/2.0; x--){
                part.add(0, cont.get(x));
            }
        }
        return part;
    }

    ...
    /* A tree is used as the data structure that composes the layers.
    *  The depth of the tree is provided by the user.
    *  The data structure is a binary tree with numNods = (2^deep)-1 nodes.
    * The number of leaves is numLeaves = 2^(deep-1).
    */
    int N;    // Number of double numbers
    int deep; // Dept of the tree
    ...
    // dependent variables
    int numLeaves;    // Number of leaves of the tree
    int numNods;      // Number of nodes of the tree

    numLeaves = (int)(Math.pow(2, deep-1));
    numNods = (int)(Math.pow(2, deep)-1);
    ...
    if(deep < 2) deep = 2;
    if(N < numLeaves) N = numLeaves + 50;
    ...
    // A Vector of nodes is contained in the tree
    Vector<Node<ArrayList<Double>>> nods =
            new Vector<Node<ArrayList<Double>>>(numNods);
    for(int x = 0; x < numNods; x++){
            nods.add(new Node<ArrayList<Double>>(new ArrayList<Double>()));
     }
     tree=new BinaryTree(nods);
     ...
     tree.getNode(0).setCont(nums);
     tree.getNode(0).getCont());
     new ParallelFFT(tree.getNode(0),v,v2).getOrdered().getCont();
    }
}
```

This class makes use of a binary tree as the basic data structure that represents the FFT as a layered coordination. Thus, this class creates a `tree` data structure of `Node` components, which represents the coordination of the whole parallel software system, developed for executing on the available parallel hardware platform. Each `Node` operates on `ArrayLists` in Java instead of `double` arrays, to take advantage of the many possible operations that the Java programming language has available for `ArrayLists`. So, the FFT algorithm is applied to `ArrayLists` in Java, as it is shown as follows.

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 `Node` components. It is now the moment to insert the sequential processing which corresponds to the FFT algorithm and data description found in the Problem Analysis, This is done in the class `FFT`, which considers the particular declarations for the FFT algorithm computation:

```
public class FFT{
    ...
    public static void FFT(ArrayList<Double> a){
        FFT(a, 0, a.size() - 1);
    }
    private static void FFT(ArrayList<Double> a, int left, int right){
        if (right <= left) return;
        int i = partition(a, left, right);
        FFT(a, left, i-1);
        FFT(a, i+1, right);
    }

    private static int combine(ArrayList<Double> a, int left, int right){
        int i = left - 1;
        int j = right;
        while (true) {
            while(less(a.get(++i),a.get(right)));// find item on left to swap
            while(less(a.get(right),a.get(--j))) // find item on right to swap
                if (j == left) break;            // do not go out-of-bounds
            if (i >= j) break;                   // check if pointers cross
            exch(a, i, j);                       // swap two elements into place
        }
        exch(a, i, right);                      // swap with partition element
        return i;
    }
    ...
    public static void main(String a[]){
        ArrayList<Double> v = new ArrayList<Double>();
        ArrayList<Double> v2 = new ArrayList<Double>();
        for(int x=0; x<1000; x++){
            v.add(new Random().nextDouble()*100);
        }
        ...
        FFT(v);
    }
}
```

This simple, sequential Java code allows that each `Node` component obtains a local FFT over its ArrayList provided. Modifying this code implies modifying the processing behavior of the whole parallel software system, so the class `ParallelFFT` can be modified and used for other parallel applications, as long as they are tree-like computations and execute on a shared memory parallel computer.

5 Summary

The Parallel Layers pattern is an Architectural Pattern for Parallel Programming applied here, along with part of 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 FFT 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 a tree-like algorithm, 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] P. Brinch-Hansen *Distributed Processes: A Concurrent Programming Concept.*, Communications of the ACM, Vol.21, No. 11, 1978.

[3] P. Brinch-Hansen *Studies in Computational Science. Parallel Programming Paradigms.*, Prentice-Hall, 1995.

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

[5] J.W. Cooley, and J.W. Tukey *An algorithm for machine calculation of complex Fourier series.* Mathematics of Computation, 19, pp. 297-301, 1965.

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

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

[8] Herlihy, M., and Shavit, N., *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers. Elsevier, 2008.

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

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

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

[12] Christopher H. Nevison, Daniel C. Hyde, G. Michael Schneider, Paul T. Tymann. Laboratories for Parallel Computing. Jones and Bartlett Publishers, 1994.

[13] 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.

[14] 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.

[15] J.L. Ortega-Arjona *The Shared Resource Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming*, Proceedings of the 10th Pattern Languages of Programming 2003 (PLoP2003)Allerton Park, Illinois, USA, 2003.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] Cherri M. Pancake. Is Parallelism for You? Oregon State University. Originally published in Computational Science and Engineering, Vol. 3, No. 2. Summer, 1996.

[22] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. Pattern-Oriented Software Architecture. John Wiley & Sons, Ltd., 1996.

[23] Mary Shaw and David Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall Publishing, 1996.