Applying Architectural Patterns for Parallel Programming

An Hypercube Sorting

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

Héctor Benítez-Pérez
DISCA, IIMAS, UNAM
hector@uxdea4.iimas.unam.mx

Abstract

The architectural patterns for parallel programming is a collection of patterns related with a method for developing the coordination 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 with the method for developing a coordination for solving an hypercube sorting. The method used here takes the information from the problem analysis, proposes an architectural pattern for the coordination, and provides some elements about its implementation.

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* [19]. This definition is obtained from the original research work in parallel programming provided by E.W. Dijkstra [5], C.A.R. Hoare [9], P. Brinch-Hansen [2], and many others, who have established the main basis for parallel programming today. 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: *(a)* the hardware platform, *(b)* the programming language, and *(c)* the problem to solve.

Nevertheless, parallel programming still represents a hard problem to the software designer and programmer: we do not yet know how to solve an arbitrary problem efficiently on a parallel system of arbitrary size. 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 *coordinations*, it is possible to avoid a lot of errors and aggravation. Many approaches have been presented up to date, proposing descriptions of top-level coordinations observed in parallel programs. Some of these descriptions are: *Outlines of the Program* [4], *Programming Paradigms* [10], *Parallel Algorithms* [6], *High-level Design Strategies* [11], and *Paradigms for Process Interaction* [1]. These descriptions provide common overall coordinations 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 and program. 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 subsystems, 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. 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.

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 a parallel program, having the following advantages [13, 14, 15, 16, 17, 18, 20]:

- The architectural patterns for parallel programming 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 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 statement.
- 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, 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. 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].

2 Problem Analysis – Quicksort and Hypercube Sorting

The present paper attempts to demonstrate the use of the architectural patterns for parallel programming for designing a coordination that solves an hypercube sorting, based on the Quicksort algorithm [8]. The objective is to show how an architectural pattern can be chosen and used so it deals with the functionality and requirements present in this problem.

2.1 Problem Statement

2.1.1 Quicksort

Quicksort is perhaps the best well known sequential algorithm for array sorting [8]. As it is defined, Quicksort is able to sort an array of n items in an $O(n\, log\, n)$ time. Nevertheless, in the worst case, it presents a sorting time of $O(n^2)$. This feature of Quicksort makes difficult to design and implement its parallel version, so it efficiently works on a multiprocessor computer.

The basic Quicksort algorithm relays on a partitioning to divide the sorting array of n items into two sorting arrays of $n/2$ items. This division is repeated recursively (obtaining sorting arrays of $n/4$, $n/8$, and so on) until the sorting array is composed of a single item, or an empty operation [8].

The partitioning is based on an algorithm which initially selects an arbitrary key from an slice of the array and splits this slice into two parts, keeping that no item in a “left” part is larger than the key, and no item in a “right” part is smaller than the key. The following algorithm, for example, uses the middle item as key for an array `a` of `int`:

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

```
        ...
        first = a[0];
        last = a[n];
    }
    public void partition(int first, int last){
        int i = first;
        int j = last;
        int temp, key;
        key = a[(i+j)/2];
        while(i <= j){
            while(a[i] < key) i = i + 1;
            while(key < a[j]) j = j - 1;
            if(i <= j){
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
                i++;
                j--;
            }
        }
    }
    public void Quicksort(int first, int last){
        int i, j;
        ...
        if(first < last){
            partition(first,last);
            Quicksort(first,j);
            Quicksort(i,last);
        }
    }
}
```

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. Nevertheless, the partitioning generates slices of unknown sizes. In the best case, slices have the same size, but in the worst case, a slice of a single item may result. This feature of Quicksort makes it difficult to produce a parallel version that distributes a similar amount of work among nodes of a parallel computer.

2.1.2 Hypercube Sorting

A parallel sorting based on Quicksort can be carried out by using an hypercube approach. By now, this parallel sorting is explained in terms of an hypercube structure with eight nodes (Figure 1). Even though the partitioning of the array into slices produces sub-arrays of unknown lengths (which cause unbalance problems for the parallel computer), let us assume that, somehow, the partitioning provides sub-arrays of equal, or nearly equal, size.

An eight-node cube sorts n `int` numbers in three main steps:

1. *Partitioning:* In a first step, node 0 receives the n `int` numbers, and divides them into two halves. One half is sent to node 1, whereas the other half is kept in node 0. A second step now divides each half within node 0 and node 1 into fourths. Node 0 and node 1 keep one fourth, sending the other fourths to node 2 and node 3, respectively. In a third and final partition step, nodes 0, 1, 2, and 3 divide their respective fourths into eighths, again keeping one eighth and respectively sending the other eighth to nodes 4, 5, 6, and 7. This partition procedure ends having an eighth of the number set assigned to each node.

2. *Sorting:* Simultaneously, each node sorts its assigned eighth of the problem.

3. *Combining:* Simultaneously, nodes 0, 1, 2, and 3 receive the sorted eighth from nodes 4, 5, 6, and 7, and combine them with their own eighth into a sorted forth sequence. Just after this, nodes 0 and 1 receive one fourth from nodes 2 and 3, respectively. Again these fourths are combined into a sorted half sequence on each node 0 and 1. Finally, node 0 receives the sorted half from node 1, combining it with its own half sequence, and outputting a complete sorted sequence of size n.

It is noticeable that larger hypercubes can be used, following a similar distribution of data, and dividing the problem each time into smaller problems which can be solved in parallel. In general, a hypercube has p nodes, where p is a power of 2. So considering d as the dimension of the hypercube, the relations between number of nodes and dimension are:

$$p = 2^d \qquad d = log\, p$$

As described previously, the hypercube solution is a hierarchical organization, in which each level can be seen as composed by a number of nodes. Hence, the sorting problem is distributed through the hypercube, one level (or dimension) at a time. In general, an hypercube with dimension d has $d + 1$ levels.

2.2 Specification of the Problem

From the previous section, considering an array of n `int` numbers, it is possible to analyze T_1 as the average sequential runtime required to solve the problem into

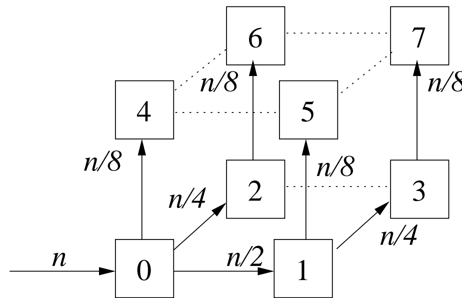

Figure 1: Distribution of data to be sorted into an 8 node cube.

a single node. Input and output of the array of n int 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, it can be observed that solving the sorting 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. Furthermore, 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.* Quicksort, 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 sort 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 sorts an array 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 int [] a = null;
   ...
   public void run(){
      ...
      receive(c,first,last);
      for(int k = first; k < last; k++){
         receive(c, a[k]);
      }
      ...
      Quicksort(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 Quicksort 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 platform available for this parallel program is a cluster of computers, specifically, a dual-core server (Intel dual Xeon processors, 1 Gigabyte RAM, 80 Gigabytes HDD) 16 nodes (each with Intel Pentium IV processors, 512 Megabytes RAM, 40 Gigabytes HDD), which communicate through an Ethernet network. The parallel application for this platform is programmed using the Java programming language [6, 7].
- *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

In this section, the architectural patterns for parallel programming [13, 19, 20] are used along with the the information from the problem analysis, in order to propose an architectural pattern for developing a coordination structure that performs a parallel hypercube sorting, based on Quicksort.

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 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 proposing an architectural pattern for a parallel solution to the hypercube sorting problem is presented as follows [20]:

1. *Analyze the design problem and obtain its specification.* Analyzing the problem description and the algorithmic solution provided, it is noticeable that hypercube sorting 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 sorting an array of a single element becomes a trivial operation. Only then, the algorithm continues retrieving the sorted data, now going back in the hierarchical structure.

2. *Select the category of parallelism.* Observing the form in which the algorithmic solution partitions the problem, it is clear that the algorithm partitions the sorting operation into sub-sorting 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 sorting operation is obtained using exactly the same algorithm, this is, Quicksort. Thus, the nature of the processing components of a probable solution for the hypercube sorting, 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 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 hypercube sorting 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 hypercube sorting, it can be defined in terms of a sorting algorithm composed of two sub-algorithms, which is divided over and over, recursively growing as a tree-like structure. Each sorting sub-algorithm performs completely 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 hypercube sorting 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 architectural pattern is used here to describe the solution to the hypercube sorting in terms of this architectural

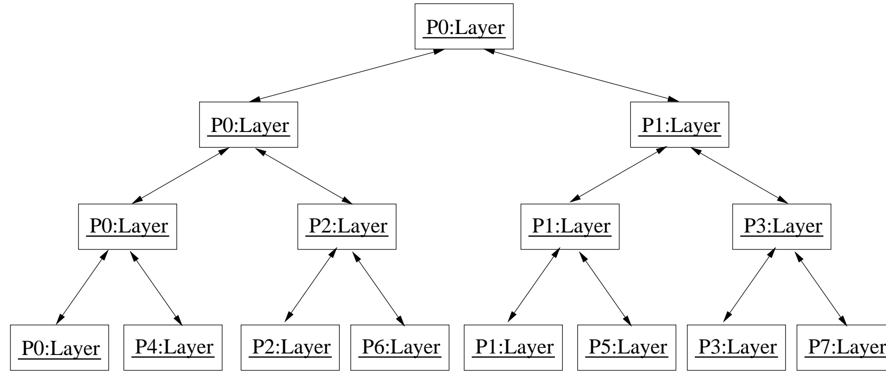

Figure 2: Object diagram of the Parallel Layers pattern applied for solving the hypercube sorting.

pattern's structure and behavior [18, 19, 20].

1. *Structure.* Using the Parallel Layers architectural pattern for hypercube sorting, different data is sorted 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 Quicksort operation. To communicate, layers use calls, referring to each other. Quicksort 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 hypercube shape is mapped for dividing the Quicksort operations is shown in Figure 2.

   Notice that this organization effectively allow to distribute data among layer components as described by the hypercube sorting, as previously described in the problem analysis.

2. *Dynamics.* A typical scenario of three levels is used to describe the basic run-time behavior of this pattern when applied to the hypercube sorting of n `int` numbers. All layer components are active at the same time, accepting a function call with its assigned `int` numbers, distributing them through two function calls to its child components in lower level layers, and once all `int` numbers are completely distributed, applying a Quicksort operation to the returned results from the child components. This pattern is used here to repeatedly perform a parallel hypercube sorting, as series of tree ordered Quicksort operations, as described in Figure 2. The parallel execution follows the description of the hypercube sorting (Figure 3):

3.3 Functional description of components

This section describes each processing and communicating software components as participants of the Parallel Layers architectural pattern, es-

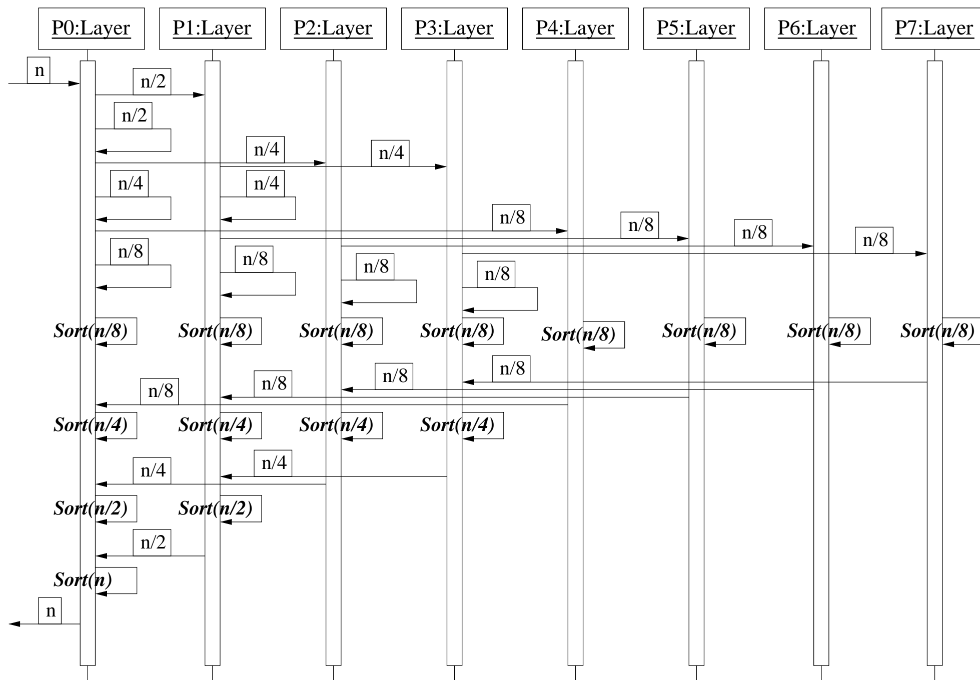

Figure 3: Sequence diagram of the Parallel Layers pattern for solving the hypercube sorting.

tablishing its responsibilities, input and output for solving the hypercube sorting.

- Layer component. The responsibilities of a layer component here are to allow the creation of the tree structure for solving the hypercube sorting. 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 Quicksort 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 architectural pattern uses functional parallelism to execute the Quicksort, allowing the simultaneous existence and execution of more than one instance of a layer component through time. Each one of these instances at the same time divides the data for further applying Quicksort. In a layered system like this, hypercube sorting involves the execution of Quicksort in several layers. These Quicksort 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 Quicksort 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 Quicksort operations can be carried out by the same system, by allowing several Quicksorts 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 hypercube sorting has the following advantages and disadvantages:

- **Advantages**
  (a) 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 [24]. Allowing each layer to be presented as multiple components executing in parallel allows to perform the computation several times, enhancing performance.

(b) 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, 24].

(c) 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, 24].

(d) 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.

(e) 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**

(a) Not every system computation can be efficiently structured as layers. Considerations of performance may require a strong coupling between high-level functions and their lower level implementations. Load balance among layers is also a difficult issue for performance [24, 21].

(b) 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].

(c) 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, 24, 12].

(d) 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

In this section, all the 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 hypercube sorting, such as the Fast Fourier Transform [3].

4.1 Coordination

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

```
class ParallelQS{
    ...
    private static BinaryTree<ArrayList<Int>> tree;
    private Node<ArrayList<Int>> rootNode;
    private ArrayList<Int> merge;
    ...
    public ParallelQS(Node <ArrayList<Int>> rootNode){
        this(rootNode,new Vector(), new Vector());
    }
    public Node<ArrayList<Int>> getOrdered(){
        return rootNode;
    }
    ...
    private ArrayList<Int> divide(ArrayList<Int> cont, boolean half1){
        ArrayList<Int> part = new ArrayList<Int>();
        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;
    }
    private ArrayList<Int> merge(ArrayList<Int> left,
        ArrayList<Int> right){
```

```
        ArrayList<Int> merge = new ArrayList<Int>();
        while(left.size() > 0 && right.size() > 0){
            if(left.get(0) < right.get(0)){
                merge.add(left.remove(0));
            }
            else{
                merge.add(right.remove(0));
            }
        }
        if(left.size() > 0 && right.size() == 0){
            merge.addAll(left);
        }
        else{
            merge.addAll(right);
        }
        return merge;
    }

    public static void main(String[] args){
        ...
        /*
        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 int numbers to order
        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<Int>>> nods =
            new Vector<Node<ArrayList<Int>>>(numNods);
        for(int x = 0; x < numNods; x++){
            nods.add(new Node<ArrayList<Int>>(new ArrayList<Int>()));
        }
        tree=new BinaryTree(nods);
        ...
        tree.getNode(0).setCont(nums);
        tree.getNode(0).getCont());
        new ParallelQS(tree.getNode(0),v,v2).getOrdered().getCont();
    }
}
```

This class makes use of a binary tree as the basic data structure that represents the hypercube sorting 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 `int` arrays, to take advantage of the many possible operations that the Java programming language has available for `ArrayLists`. So, the Quicksort algorithm is applied to `ArrayLists` in Java, as it is shown as follows.

The utility of the coordination presented here goes beyond of a parallel hypercube sorting application. By modifying the sequential processing section, each layer component is capable of processing other tree-like problems, such as the Fast Fourier Transform [3].

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 Quicksort algorithm and data description found in the problem analysis, This is done in the class `Quicksort`, which considers the particular declarations for the Quicksort algorithm computation [23]:

```
public class Quicksort {
    private static long comparisons = 0;
    private static long exchanges   = 0;
    public static void Quicksort(ArrayList<Int> a){
        Quicksort(a, 0, a.size() - 1);
    }
    private static void Quicksort(ArrayList<Int> a, int left, int right){
        if (right <= left) return;
        int i = partition(a, left, right);
        Quicksort(a, left, i-1);
        Quicksort(a, i+1, right);
    }
    private static int partition(ArrayList<Int> 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;
    }
    private static boolean less(double x, double y) {
        comparisons++;
        return (x < y);
    }
    private static void exch(ArrayList<Int> a, int i, int j) {
```

```
        exchanges++;
        double swap = a.get(i);
        a.set(i, a.get(j));
        a.set(j, swap);
    }
    private static void shuffle(ArrayList<Int> a) {
        int N = a.size();
        for (int i = 0; i < N; i++) {
            int r = i + (int)(Math.random()*(N-i)); // between i and N-1
            exch(a, i, r);
        }
    }
    public static void main(String a[]){
        ArrayList<Int> v=new ArrayList<Int>();
        ArrayList<Int> v2=new ArrayList<Int>();
        for(int x=0; x<1000; x++){
            v.add(new Random().nextInt()*100);
        }
        ...
        Quicksort(v);
    }
}
```

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

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 hypercube sorting 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 distributed memory parallel platform.

6 Acknowledgements

The author wishes to thank Veli-Pekka Eloranta and Ville Reijonen, my shepherds for EuroPLoP 2010, for his encouraging comments about the present paper. This work is part of an ongoing research, funded by project IN103310, PAPIIT-DGAPA-UNAM, 2010.

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

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

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

[8] C.A.R. Hoare *Algorithm 64: Quicksort.* Communications of the ACM, No. 4, 1961.

[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 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP98), Kloster Irsee, Germany, 1998.

[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] Robert Sedgewick. Algorithms in Java. Addison-Wesley Professional, 3 edition, 2002.

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