

Applying Architectural Patterns for Parallel Programming

Solving the One-dimensional Heat Equation

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

Abstract

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

In this paper, it is presented the application of the architectural patterns along with the method for developing a coordination structure for solving the One-dimensional Heat Equation. The method used here takes the information from the problem analysis, selects an architectural pattern for the coordination, and provides 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* [16]. This definition is obtained from the original research work in parallel programming provided by E.W. Dijkstra [4], C.A.R. Hoare [7], 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* [3], *Programming Paradigms* [8], *Parallel Algorithms* [5], *High-level Design Strategies* [9], and *Paradigms for Process Interaction* [1]. These descriptions provide common overall coordinations such as, for example, “master-slave”, “pipeline”, “workpile”, 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* [10, 11, 12, 13, 14, 15] 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* [10], 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. 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 structure. The Architectural Patterns for Parallel Programming provide descriptions about how to organize a parallel program, having the following advantages [10, 11, 12, 13, 14, 15]:

- 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 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 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.
- The Architectural Patterns for Parallel Programming introduce parallel structures or 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. 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], and further work on each particular architectural pattern in [11, 12, 13, 14, 15].

2 Problem Analysis – The One-dimensional Heat Equation

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

2.1 Problem Statement

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

elementary piece of the wire, a finite element. This element is represented as a small, one-dimensional segment of the wire, with a length of Δx (Figure 1).

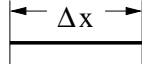


Figure 1: A small one-dimensional element.

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

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

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

$$\frac{\partial v_x}{\partial x} = 0$$

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

$$\frac{\partial^2 u}{\partial x^2} = 0$$

Known as the one-dimensional heat equation or equilibrium equation, this equation is abbreviated and expressed in general terms (and dimensions) as:

$$\nabla^2 u = 0$$

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

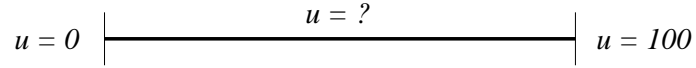


Figure 2: A wire with fixed temperatures at each extreme.

In order to develop a program that models the Heat Equation, first it is necessary to obtain its discrete form. So, the wire in Figure 2 is divided into segments, each segment with a size of h . This size is relatively very small regarding the size of the whole wire, so the segment can be considered as a single point within the wire. So, this results on a segmented wire, in which two types of segments can be considered (Figure 3).

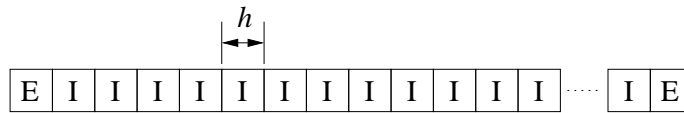


Figure 3: A segmented wire with two types of elements: interior (I) and extreme (E).

1. Interior segments, which require computing their temperatures, each one having to satisfy the heat equation.
2. Extreme segments, which have fixed and given temperatures.

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

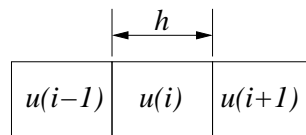


Figure 4: An element $u(i)$ and its two neighbouring elements.

Notice that for the case, h should be small enough so each neighbouring element's temperature can be approximated in terms of a Taylor expansion. So, the discrete heat equation is reduced to a difference equation. Rearranging it, it

is noticeable that for thermal equilibrium, the temperature of a single element $u(i)$ in time, from one thermal state to another, is:

$$u(t + 1, i) \approx u(t, i) + \frac{1}{h^2}(u(t, i - 1) + u(t, i + 1) - 2u(t, i))$$

This is the discrete equation to be used in order to obtain a parallel numerical solution for the One-dimensional Heat Equation.

2.2 Specification of the Problem

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

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

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

- *Descriptions of the data and the algorithm.* The relatively large number of segments in which a wire is divided and the discrete representation of the One-dimensional Heat Equation is described in terms of data and an algorithm. The divided region is normally represented as a long wire in terms of a $(n + 2)$ array of segments which represent every discrete element of the wire, and encapsulate some floating point data which represents temperature, as shown as follows. Thus, a whole wire consists of n interior segments and 2 extreme segments.

```

class Segment implements Runnable{
    ...
    private int i = -1;
    ...
    private Segment(int i){
        this.i = i;
        new Thread(this).start();
    }
    ...
}

```

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

```

class Segment implements Runnable{
    ...
    private int i = -1;
    ...
    public void run(){
        double temperature, received, total;
        for (int i = 0; i < iterations; i++) {
            // Here the actual segment exchanges data with
            // its neighbouring elements
            total = 0.0;
            for (i = 0; i < 2; i++) {
                // Receive from neighbouring elements
                // and put it in the variable 'received'
                total += received;
            }
            temperature += temperature + (1/h^2)*(total - 2*temperature);
        }
    }
    ...
}

```

Each time step, a new temperature for the local **Segment** object is obtained from the previous temperature and the temperatures of the neighbouring segments (whether interior or extreme). Notice that the term “time step” implies an iterative method in which the operation requires four coefficients. The algorithm described takes into consideration an iterative solution of operations, known as *relaxation*. The simplest relaxation method is the Jacobi relaxation, in which the temperature of each and every interior segment is simultaneously approximated using its local temperature and the temperatures of its neighbours (and it is the one

presented here). Other relaxation methods include the Gauss-Seidel relaxation and the successive overrelaxation (SOR). Iterative methods tend to be more efficient than direct methods.

- *Information about parallel platform and programming language.* The parallel system available for this example is a SUN SPARC Enterprise T5120 Server. This is a multi-core, shared memory parallel hardware platform, with 1×8 -Core UltraSPARC T2, 1.2 GHz processors (capable of running 64 threads), 32 Gbytes RAM, and Solaris 10 as operating system [17]. Applications for this parallel platform can be programmed using the Java programming language [5, 6].
- *Quantified requirements about performance and cost.* This application example has been developed in order to test the parallel system described in the previous point. The idea is to experiment with the platform, testing its functionality in time, and how it maps with a domain parallel application. So, the main objective is simply to test and characterise performance (in terms of execution time) regarding the number of processes/processors involved in solving a fixed size problem. Thus, it is important to retrieve information about the execution time considering several configurations, changing the number of processes on this parallel, shared memory platform.

3 Coordination Design

In this section, the *Architectural Patterns for Parallel Programming* [10] are used along with the the information from the problem analysis, in order to select an architectural pattern for developing a coordination structure that solves the One-dimensional Heat Equation.

3.1 Specification of the System

- **The scope.** This section aims to describe the basic operation of the parallel software system, considering the information presented in the Problem Analysis step about the parallel system and its programming environment. Based on the problem description and algorithmic solution presented in the previous section, the procedure for selecting an architectural pattern for a parallel solution to the One-dimensional Heat Equation problem is presented as follows [10]:

1. *Analyse the design problem and obtain its specification.* Analysing the problem description and the algorithmic solution provided, it is noticeable that the calculation of the One-dimensional Heat Equation is a step-by-step, iterative process. Such a process is based on calculating the next temperature of each segment of the wire through

each time step. The calculation uses as input the previous temperature, and the temperatures of the two neighbour segments of the wire, and provides the temperature at the next time step.

2. *Select the category of parallelism.* Observing the form in which the algorithmic solution partitions the problem, it is clear that the wire is divided into segments, and computations should be executed simultaneously on different segments. Hence, the algorithmic solution description implies the category of **Domain Parallelism**.
3. *Select the category of the nature of the processing components.* Also, from the algorithmic description of the solution, it is clear that the temperature of each segment of the wire is obtained using exactly the same calculations. Thus, the nature of the processing components of a probable solution for the One-dimensional Heat Equation, using the algorithm proposed, is certainly a **Homogeneous** one.
4. *Compare the problem specification with the architectural pattern's Problem section.* An Architectural Pattern that directly copes with the categories of domain parallelism and the homogeneous nature [10] of processing components is the **Communicating Sequential Elements (CSE) pattern** [11]. In order to verify that this architectural pattern actually copes with the One-dimensional Heat Equation problem, let us compare the problem description with the Problem section of the CSE pattern. From the CSE pattern description, the problem is defined as [11]:

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

Observing the algorithmic solution for the One-dimensional Heat Equation, it can be defined in terms of calculating the next temperature of the wire segments as ordered data. Each segment is operated almost autonomously. The exchange of data or communication should be between neighbouring segments of the wire. So, the CSE is chosen as an adequate solution for the One-dimensional Heat Equation, and the architectural pattern selection is completed. The design of the parallel software system should continue, based on the Solution section of the CSE pattern.

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

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

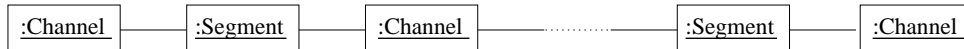


Figure 5: Object Diagram of Communicating Sequential Elements for the solution to the One-dimensional Heat Equation.

2. *Dynamics.* A scenario to describe the basic run-time behaviour of the Communicating Sequential Elements pattern for solving the One-dimensional Heat Equation is shown as follows. Notice that all the segments, as basic processing software components, are active at the same time. Every segment performs the same temperature operation, as a piece of a processing network. However, for the one-dimensional case here, each segment object communicates with its previous and next neighbours as shown in Figure 6.

The processing and communicating scenario is as follows:

- Initially, consider only a single **Segment** object, **segment(i)**. At first, it exchanges its local temperature value with its neighbours **segment(i-1)** and **segment(i+1)** through the adequate communication **Channel** components. After this, **segment(i)** counts with the different temperatures from its neighbours.
 - The temperature operation is simultaneously started by the **segment(i)** component and all the other components of the wire.
 - In order to continue, all components iterate as many times as required, exchanging their partial temperature values through the available communication channels.
 - The process repeats until each component has finished iterating, and thus, finishing the whole One-dimensional Heat Equation computation.
3. *Functional description of components.* This section describes each processing and communicating software components as participants

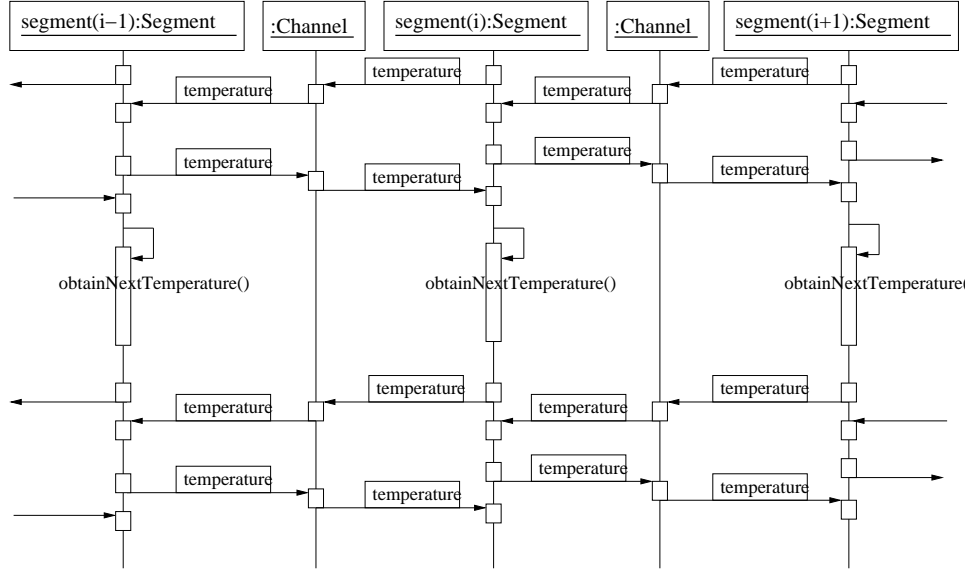


Figure 6: Sequence Diagram of the Communicating Sequential Elements for communicating temperatures through channel components for the One-dimensional Heat Equation.

of the Communicating Sequential Elements architectural pattern, establishing its responsibilities, input and output for solving the One-dimensional Heat Equation.

- **Segment.** The responsibilities of a segment, as a processing component, are to obtain the next temperature from the temperature values it receives, and make available its own temperature value so its neighbouring components are able to proceed.
 - **Channel.** The responsibilities of every channel, as a communication component, are to allow sending and receiving temperature values, synchronising the communication activity between neighbouring sequential elements. Channel components are developed as the main design objective of a following step, called “Communication Design”, which is not addressed in this paper.
4. *Description of the coordination.* The Communicating Sequential Elements pattern describes a coordination in which multiple **Segment** objects act as concurrent processing software components, each one applying the same temperature operation, whereas **Channel** objects act as communication software component which allow exchanging temperature values between sequential components. No temperature values are directly shared among **Segment** objects, but each one may access only its own private temperature values. Every **Seg-**

ment object communicates by sending its temperature value from its local space to its neighbouring **Segment** objects, and receiving in exchange their temperature values. This communication is normally asynchronous, considering the exchange of a single temperature value, in a one to one fashion. Therefore, the data representing the whole one-dimensional wire represents the regular logical structure in which data of the problem is arranged. The solution, in terms of a segmented wire, is presented as a network that actually reflects this logical structure in the most transparent and natural form [11].

5. *Coordination analysis.* The use of the Communicating Sequential Elements patterns as a base for organising the coordination of a parallel software system for solving the One-dimensional Heat Equation has the following advantages and disadvantages:

– **Advantages**

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

– **Liabilities**

- (a) The performance of a parallel application for solving the One-dimensional Heat Equation based on the Communicating Sequential Elements pattern is heavily impacted by the communication strategy used. For the present example, the threads available in the parallel platform have to take care of a large number of **Segment** objects, so each thread has to operate on a subset of the data rather than on a single value. Due to this, dependencies between data, expressed as

communication exchanges, could be a cause of a slow down in the program execution.

- (b) For this example, load balancing is kept by allowing only a fixed number of **Segment** objects per thread, which tends to be larger than the number of threads available. Nevertheless, if data would not be easily divided into same-size subsets, then the computational intensity varies on different processors. Even though every processor is virtually equal to the others, maintaining the synchronisation of the parallel application means that any thread that slows down should eventually catch up before the computation can proceed to the next step. This builds up as the computations proceeds, and could impacts strongly on the overall performance.
- (c) Using synchronous communications implies a significant amount of effort required to get a minimal increment in performance. On the otherhand, if the communications are kept asynchronous, it is more likely that delays would be avoided. This is taken into consideration in the next step, “Communication Design” (not described here).

4 Implementation

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

Here, it is only presented the implementation of the coordination structure, in which the processing components are introduced, implementing the actual computation that is to be executed in parallel. Further design work is required for developing the channel as communication and synchronisation 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 One-dimensional Heat Equation, such as the Wave Equation or the Poisson Equation.

4.1 Coordination

Considering the existence of a class **Channel** for defining the communications between **Segment** objects, the Communicating Sequential Elements architectural

pattern is used here to implement the main Java class of the parallel software system that solves the One-dimensional Heat Equation. The class `Segment` is presented as follows. This class represents the Communicating Sequential Elements coordination for the One-dimensional Heat Equation example.

```

class Segment implements Runnable{
    private static int M = 65536, iterations = 10;
    private static Channel[][] segment = null;
    private int i = -1;
    public Segment(int i){
        this.i = i;
        new Thread(this).start();
    }
    public void run(){
        double temperature, received, total;
        temperature = random(10*M);
        for (int iter = 0; iter < iterations; iter++) {
            // Send local temperature to neighbours
            if (i < M-2) send(segment[i+1][0], temperature);
            if (i > 1) send(segment[i-1][1], temperature);
            total = 0.0;
            // Receive temperature from neighbours
            if(i > 0 && i < M-1){
                received = receive(segment[i][0]);
                total += received;
                received = receive(segment[i][1]);
                total += received;
            }
            // Insert processing here
        }
    }
    public static void main(String[] args){
        segment = new Channel[M][2];
        for(int m = 0; m < M; m++){
            for(int i = 0; i < 2; i++){
                segment[m][i] = new Channel();
            }
        }
        for(int m = 0; m < M; m++){
            new Segment(m);
        }
        System.exit(0);
    }
}

```

This class only creates two adjacent, one-dimensional arrays of `Channel` components and `Segment` components, which represents the coordination structure

of the whole parallel software system, developed for executing on the available parallel hardware platform. **Channel** components are used for exchanging temperature values between neighbouring **Segment** components, each one first sending its own temperature value (which is an asynchronous, non-blocking operation), and later retrieving the temperature values of the two neighbouring wire components. Using this data, now it is possible to sequentially process to obtain the new temperature of the present component. This communication-processing activity repeats as many times as iterations defined.

The utility of the coordination presented here goes beyond the One-dimensional Heat Equation application. By modifying the sequential processing section, each wire component is capable of computing the discrete versions of other one-dimensional differential equations, such as the Wave Equation or the Poisson Equation.

4.2 Processing components

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

```
temperature += temperature + (1/h^2)*(total - 2*temperature);
```

The simple, sequential Java code allows that each **Segment** component obtains a local temperature based on the One-dimensional Heat Equation. Modifying this code implies modifying the processing behaviour of the whole parallel software system, so the class **Segment** can be used for other parallel applications, as long as they are one-dimensional and execute on a shared 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 One-dimensional Heat Equation problem. The main objective of this paper is to demonstrate, with a particular example, the detailed design and implementation that may be guided by a selected architectural pattern. Moreover, the application of the Architectural Patterns for Parallel Programming and the method for selecting them is proposed to be used during the Coordination Design and Implementation for other similar problems that involve the

calculation of differential equations for a one-dimensional problem, executing on a shared memory parallel platform.

6 Acknowledgements

The author wishes to thank Neil Harrison, my shepherd for EuroPLoP 2009, for his encouraging comments about the present paper.

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] S. Hartley *Concurrent Programming. The Java Programming Language.*, Oxford University Press Inc., 1998.
- [6] Herlihy, M., and Shavit, N., *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers. Elsevier, 2008.
- [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 *Multithreaded 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*, PhD Thesis, Department of Computer Science, University College London, UK, 2007. <http://www.sigsoft.org/phdDissertations/theses/JorgeOrtega.pdf>
- [17] Sun Microsystems. *Sun SPARC Enterprise T5120 Server*. <http://www.sun.com/servers/coolthreads/t5120/>.