Applying Architectural Patterns for Parallel Programming

An N-body Simulation

Jorge L. Ortega-Arjona
Departamento de Matemáticas
Facultad de Ciencias, UNAM
jloa@fciencias.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 an N-Body Simulation. The method used here takes the information from the problem analysis, selects 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*. 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. Specifically, obtaining a parallel program from an algorithmic description is the main objective of the area of Parallel Software Design [17].

1.1 Parallel Software Design

Parallel Software Design proposes programming techniques to deal with the parallelization of a problem, described in algorithmic terms. The research in the area covers several approaches that provide forms to organize software with

relatively independent parts which efficiently make use of multiple processors. As stated before, the goal is to obtain a parallel program from an algorithmic description. Nevertheless, designing parallel programs can be frustrating [17]:

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

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

1.2 Architectural Patterns for Parallel Programming

The *Architectural Patterns for Parallel Programming* [10, 11, 12, 13, 14, 15, 16, 17] represent an approach for parallel programming using Software Patterns. Architectural Patterns attempt to save the transformation "jump" between algorithm and program. These Software Patterns are *fundamental organizational descriptions of common top-level structures observed in parallel software systems* [10, 16, 17]. They specify properties and responsibilities of their sub-systems, and the particular form in which they are assembled together.

Simply put, architectural patterns allow software designers and developers to understand complex software systems in larger conceptual blocks and their relations, thus reducing the cognitive burden. Furthermore, architectural patterns provide several "forms" in which software components of a parallel software system can be structured or arranged, so the overall structure of such a software system arises. Architectural patterns also provide a vocabulary that may be used when designing the overall structure of a parallel software system, to talk about such a structure, and feasible implementation techniques. As such, the Architectural Patterns for Parallel Programming refer to concepts that have formed the basis of previous successful parallel software systems [16, 17].

The most important step in designing a parallel program is to think carefully about its overall structure. The Architectural Patterns for Parallel Programming provide descriptions about how to organize a parallel program, having the following advantages [10, 11, 12, 13, 14, 15, 16, 17]:

- The Architectural Patterns for Parallel Programming (as any Software Pattern) provide a description that links a problem statement (in terms

of an algorithm and the data to be operated on) with a solution statement (in terms of an organization structure of communicating software components).

- The partition of the problem to solve is a key for the success or failure of a parallel program. Hence, the Architectural Patterns for Parallel Programming have been developed and classified based on the kind of partition applied to the algorithm and/or the data present in the problem statement.

- 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 as forms in which software components can be assembled or arranged together, considering the different partitioning ways of the algorithm and/or data.

Nevertheless, even though the Architectural Patterns for Parallel Programming have these advantages, they also present the disadvantage of not describing, representing, or producing a complete parallel program in detail. Anyway, the Architectural Patterns for Parallel Programming are proposed as a way of helping a software designer to select a parallel structure as a starting point when designing a parallel program. For a complete exposition of the Architectural Patterns for Parallel Programming, refer to [10, 17], and futher work on each particular architectural pattern in [11, 12, 13, 14, 15].

2 Problem Analysis – The N-body Simulation

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

2.1 Problem Statement

An *N-body* simulation refers to compute the trajectories of n bodies present in a three-dimensional space, which interact through gravitational forces only. At discrete time intervals, the forces over each body are computed, adjusting its velocity and position [5]. Figure 1 attempts to show how a single body is affected by the gravitational forces produced by the existence of other bodies in a two-dimensional space.

Given all the attraction forces, the body modifies its velocity and position. So, the overall effect of all the forces is obtaned using a force summation. On the other hand, all bodies are affected by the interaction with the other bodies, so each one modifies its velocity and position through time. Hence, the whole behavior of the system is obtained using a simple time integration for each body.

2.1.1 Force Summation

Commonly, each body is considered as a point in a three-dimensional space. Its state is defined by its mass m and three vectors: position $\mathbf{r}$, velocity $\mathbf{v}$, and total force $\mathbf{f}$. The last one is a result of the attraction of the rest of the bodies on this particular body. Figure 2 shows two bodies, b_i and b_j, and their position vectors relative to the origin O.

The body b_j attracts b_i with a force $\mathbf{f_{ij}}$ along the vector $\mathbf{r_{ij}}$. By Newton's law, the magnitude of this force is proportional to the mass of each body and inversely proportional to the square of the distance:

$$\mathbf{f_{ij}} = \frac{Gm_i m_j}{|r_{ij}|^2}\mathbf{r_{ij}}$$

where G is the universal gravitational constant. Moreover, b_i attracts b_j with a force $\mathbf{f_{ji}}$ of the same magnitude, but opposite direction, regarding Newton's thrid law.

Hence, the total force $\mathbf{f_i}$ acting over a body b_i is the vectorial addition of all

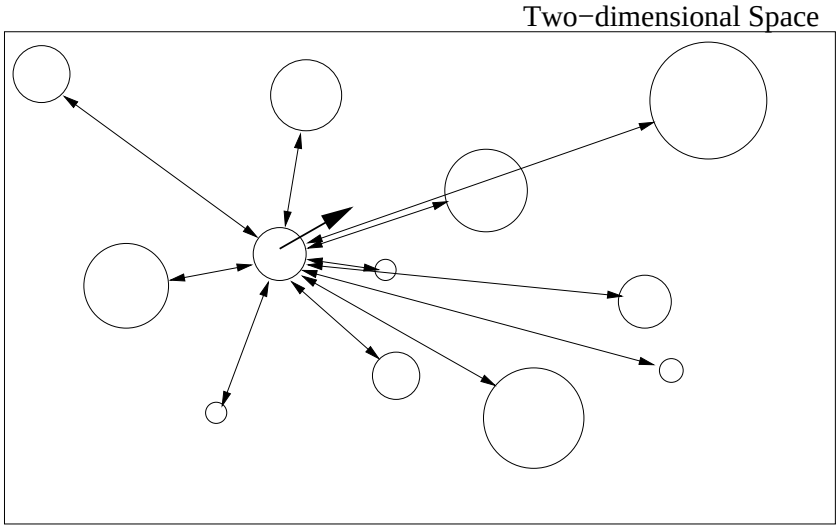

Figure 1: An example of a body, interacting with other bodies in a two-dimensional space.

the forces on b_i, resulting from the other $N-1$ bodies:

$$\mathbf{f_i} = \sum_{i=1}^{N-1} \mathbf{f_{ij}} \qquad (i \neq j)$$

2.1.2 Time integration

In order to obtain the new position of a body b_i due to force $\mathbf{f_i}$, it is necessary first to calculate the acceleration of the body, and with this, apply a simple integration method to obtain such a new position. By Newton's second law, it is known that the acceleration $\mathbf{a_i}$ of a body b_i is determined by its mass m_i and the total force $\mathbf{f_i}$ acting on it:

$$\mathbf{a_i} = \frac{\mathbf{f_i}}{m_i}$$

For a discrete time interval Δt, the acceleration can be considered approximately constant. Thus, the velocity $\mathbf{v_i}$ and position $\mathbf{r_i}$ of b_i increase by:

$$\begin{aligned} \Delta \mathbf{v_i} &= \mathbf{a_i} \Delta t \\ \Delta \mathbf{r_i} &= \int_0^{\Delta t} (\mathbf{v_i} + \mathbf{a_i}) dt = \mathbf{v_i} \Delta t + \frac{1}{2} \mathbf{a_i} \Delta t^2 \end{aligned}$$

From both these expressions, the new position of b_i is thus:

$$\Delta \mathbf{r_i} = (\mathbf{v_i} + \frac{1}{2} \Delta \mathbf{v_i}) \Delta t$$

If two bodies are too close, accelerations may become very large. So, in order to prevent numerical inestability, the time step should be made very small.

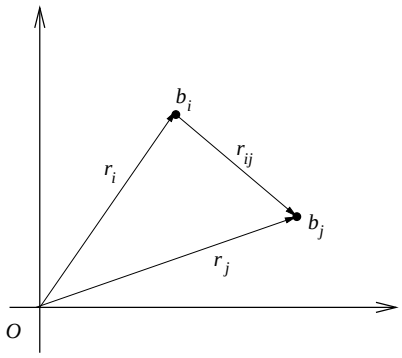

Figure 2: Two bodies considered as points in a space.

In order to avoid problems, all initial masses are considered suficiently small regarding the whole space. Nevertheless, a realistic simulation tends to be highly time-consuming.

2.2 Specification of the Problem

Analyzing the complexity of the whole problem, it is noticeable that a sequential computer can perform all $(n^2 - n)/2$ computations in $O(n^2)$ basic steps. Let us suppose a numerical example: for a problem with, for example, $n = 65,536$ bodies, it may be solved in about $4,294,967,296$ time steps. 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.* An *N-body* simulation, for a relatively large number of bodies, can be computed in a more efficient way by:

  1. using a group of software components that exploit the independent operations performed over each body, and
  2. allowing each software component to simultaneously work on a single body.

  A parallel version of this simulation may simultaneously compute forces between all pair of bodies in $O(n)$, for a certain large n. In this example, if $n \leq 10,000$ the direct method of force summation allows very accurate simulations. However, for larger systems, there are other approximations which may be faster. 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 simulation takes as its input an array of bodies, each body represented by its mass m and its vectors r, v, and f. It is important here to consider some basic vectorial operations, which may be useful when working with all the three-dimensional vectors. Finally, it seems necessary to consider some constant values, required for the computation, such as G.

  Hence, every body should be capable of obtaining its own force summation and time integration:

```
class Body implements Runnable{
   ...
   private double[] force(Body pi, Body pj){
      double g = 667 * Math.pow(10,-11);
      double[] rij = u.sub(pi.getr(), pj.getr());
```

```
      double rm = u.length(rij);
      double fm = (g * pi.getm() * pj.getm())/Math.sqrt(rm);
      double[] eij = u.mult(rij,1/rm);
      double[] force = u.mult(eij, fm);
      return force;
   }
   ...
   public Body forceT(Body[] sist, int index){
     int tcps = sist.length;
     Body it = new Body();
     Body it2 = sist[index];
     double[] fij = new double[3];
     for(int i=0; i<tcps; i++){
         if(i != index){
             Body sec = sist[i];
             double[] tmpF = force(it2,sec);
             fij = u.add(fij, tmpF);
         }
     }
     double[] rs = u.add(it2.getf(),fij);
     it.setf(rs);
     it.setm(it2.getm());
     it.setr(it2.getr());
     it.setv(it2.getv());
     return it;
   }
   ...
   public Body move(Body pi, double increment){
     double[] ai = u.mult(pi.getf(), 1/(pi.getm()));
     double[] dvi = u.mult(ai, increment);
     double[] dri = u.mult(u.add(pi.getv(),u.mult(dvi, .5)), increment);
     pi.setv(u.add(pi.getv(), dvi));
     pi.setr(u.add(pi.getr(),dri));
     return pi;
   }
   ...
}
```

Once it has the array of bodies, each **Body** object is able to compute a local force summation and time integration as a single thread.

- *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].

- *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 [10, 16, 17] 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 *N-nody* simulation.

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 *N-body* simulation is presented as follows [17]:

1. *Analyze the design problem and obtain its specification.* Analyzing the problem description and the algorithmic solution provided, it is noticeable that an *N-body* simulation yields a group of operations that may be performed simultaneously on different data. Such data, in the form of the bodies array, can be distributed and operated at the same time.

2. *Select the category of parallelism.* Observing the form in which operations can be performed by each body at a time step, it is clear that the parallel soluton should operate on different data, while distributing the data. Hence, the solution description implies the category of **Activity Parallelism** [12, 13, 16, 17].

3. *Select the category of the nature of the processing components.* Also, from the description of the solution, it is clear that operations on each body use exactly the same algorithm. Thus, the nature of the processing components of a probable solution for the *N-body* simulation, using the algorithm proposed, is certainly a **Homogeneous** one [12, 13, 16, 17].

4. *Compare the problem specification with the architectural pattern's Problem section.* An Architectural Pattern that directly copes with the categories of activity parallelism and the homogeneous nature of processing components is the **Manager-Workers (MW) pattern** [15, 16, 17]. In order to verify that this architectural pattern actually copes with the *N-body* simulation, let us compare the problem description with the Problem section

of the MW pattern. From the MW pattern description, the problem is defined as [13, 16, 17]:

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

Observing the algorithmic solution for the *N-body* simulation, it can be defined in terms of a single set of operations performed over the data of a single body. Each body, at a single time step, can be operated completely and autonomously. The data or communication should be between an organizing component and several processing components, distributing the information of all the bodies so one body can be operated by each processing component. So, the MW is chosen as an adequate solution for the *N-body* simulation, and the architectural pattern selection is completed. The design of the parallel software system should continue, based on the Solution section of the MW pattern.

3.2 Functional description of components

This section describes each processing and communicating software components as participants of the Manager-Workers architectural pattern, establishing its responsibilities, input and output for solving the *N-body* simulation [13, 16, 17].

- **Manager**. The responsibilities of a manager are to create a number of workers, to distribute work among them, to start up their execution, and to assemble the overall simulation result from the sub-results from the workers.
- **Worker**. The responsibility of a worker is to seek for the array of bodies, to implement the operations of force summation and time integration on a single body, and to perform such operations.

3.3 Structure and dynamics

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

1. *Structure.* Using the Manager-Worker architectural pattern for an *N-body* simulation, all bodies information is distributed by a manager component,

and operated by workers as conceptually-independent components. Each worker performs the same operations to obtain an update of the state of a body, at a time step. So, the operations defined for a body are simultaneously performed.

An object diagram, representing the manager and worker components on which the bodies information is distribuited is shown in Figure 3.

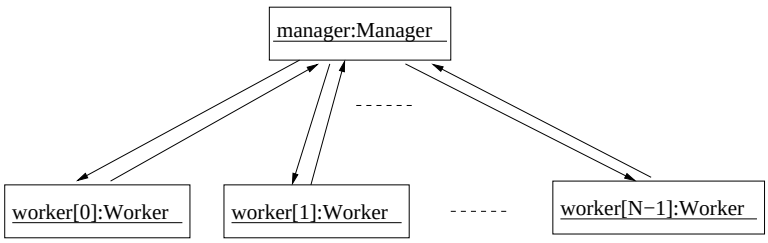

Figure 3: Object diagram of the Manager-Workers pattern applied for solving the *N-body* simulation.

Notice that this organization effectively allows to distribute the data of all bodies among worker components, as previously described in the problem analysis, so each body's new state can be computed independently from the others.

2. *Dynamics.* A typical scenario is used here to describe the basic runtime behavior of this pattern when applied to the *N-body* simulation. All components, whether manager or workers, are active at the same time, distributing and processing the information of different bodies, and assemlbling an overall state of the system at each time step. A single time step is described in Figure 4:

   - All participants are created, and wait until a data array of all the bodies `sist[]` is provided to the manager. When such data is available to the manager, this distributes it, sending all the array by request to each waiting worker.
   - Each worker receives a copy of the array. Notice that the i-th worker is associated with obtaining the force summation and the time integration for the i-th body. So, every worker starts processing an operation `forceT()` and `move()`. These operations are independent of the operations on other workers. When the i-th worker finishes processing, it returns a result of only the state of the i-th body to the manager. Once all results are received by the manager, and if the simulation is to be continued for further time steps, each worker requests again for the array representing all the state of all the bodies, and the process repeats.

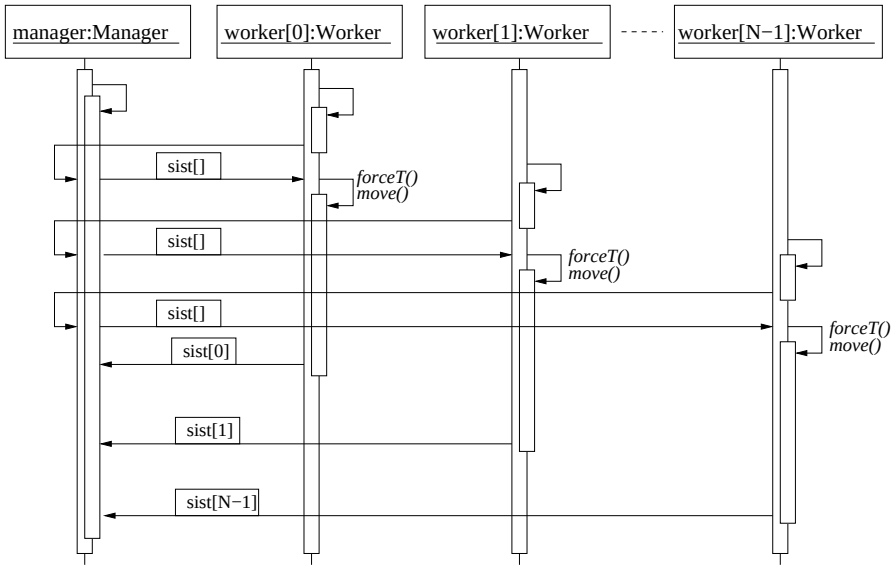

Figure 4: Sequence diagram of a single time step of the Manager-Workers pattern for solving the *N-body* simulation.

- For each time step, the manager is usually replying to requests of the array from the workers or receiving their partial results. Once all time steps have been processed, the manager assembles a total result from the partial results and the program finishes. Any non-serviced requests of data from the workers are ignored.

3.4 Description of the coordination

The Manager-Workers architectural pattern uses activity parallelism to execute the *N-body* simulation, allowing the simultaneous existence and execution of more than one worker components through time. Each one of these instances at the same time obtain the force summation and time integration for a single body, and during a time step. In a parallel system like this, the *N-body* simulation involves the distribution and execution of data for several time steps. Each time step starts by distributing the data among all workers, and finish only when all workers provide the manager with the new state of its correspondent body.

3.5 Coordination analysis

The use of the Manager-Workers pattern as a base for organizing the coordination of a parallel software system for solving the *N-body* simulation

has the following advantages and disadvantages:

- **Advantages**
  (a) The order and integrity of the whole array is preserved and granted due to the defined behaviour of the manager component. The manager takes care of what part of the array has been operated on by which worker, and what remains to be obtained by the rest of the workers.
  (b) An important characteristic of the Manager-Workers pattern is due to the independent nature of operations that each worker performs. Each worker requests for the bodies array during execution and operates considering a different body. Such an independence makes that the structure to present a natural load balance, and easily scale for larger sets of bodies.
  (c) Synchronisation is simply achieved because communications are restricted to only between manager and each worker. The manager is the component in which the synchronisation is stated.
  (d) Using the Manager-Worker pattern, the parallelising task is relatively straightforward, and it is possible to achieve a respectable performance. If designed carefully, the Manager-Worker pattern enables the performance to be increased without significant changes.
- **Liabilities**
  (a) The Manager-Workers systems may present poor performance if the number of bodies (and thus workers) is excesively large (as pointed out before, if $n > 10,000$). In such a case, workers may remain idle for periods of time while the manager is busy trying to serve all their requests. Granularity should be modified in order to balance the amount of work, by allowing that more than one body is operated by each worker.
  (b) Manager-Worker architectures may also have poor performance if the manager activities – data distribution, receive worker requests, send data, receive partial results, and assembling the final result – take a longer time compared with the processing time of the workers. The overall performance depends mostly on the manager, so programming the manager should be done taking special consideration to the time it takes to perform its activities. A poor performance of the manager impacts heavily on the performance of the whole system.
  (c) Many different considerations must be carefully considered, such as strategies for work distribution, manager and worker collaboration, and assemble of final result. In general, the issue is to find the right combination of worker number, active or passive manager, and data size in order to get the optimal performance,

but experience shows that this still remains a research issue. Moreover, it is necessary to provide error handling strategies for failure of worker execution, failure of communication between the manager and workers, or failure to start-up parallel workers.

4 Implementation

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 *N-body* simulation, such as the Matrix Multiplication [6].

4.1 Coordination

The Manager-Workers architectural pattern is used here to implement the main Java class of the parallel software system that solves the *N-body* simulation. The class `NbodyManager` is presented as follows. This class represents the Manager-Workers coordination for the *N-body* simulation.

```
class NbodyManager extends MyObject {
   ...
   private  static int N = 10000;
   private  static int numSolutions = 0;
   ...
   public static void main(String[] args) {
     ...
     Vector result = new Vector(N); // Overall result
     Body[] hist = new Body [numWorkers]; // Array for previous information
     ...
     // Initial bodies configuration is read from a file
     Body[] rs = g.parseXML("nbody.xml");
     ...
     // Start iterations for (N-1) time steps
```

```
    for(int it = 0; it < (N-1); it++){
        rs = new Body[numWorkers];
        ...
        for (int i = 0; i < numWorkers; i++) new NbodyWorker(i, er);
        // send out all the "work" (initial configurations)
        ...
        rs = new Body[numWorkers]; // Generate an N-body system
        ...
        // Clean previous information
        hist = new Body[numWorkers];
        ...
        // Store the result of this iteration
        result.addElement(hist);
        iterador++;
        ...
    }
    // Save information into a file
    file.genXML(result);
    System.exit(0);
  }
}
```

This class makes use of an array of bodies as the basic data structure that represents the three-dimensional space and the bodies within it. Thus, this class creates a `rs` data structure of `Body` components, which represents the coordination of the whole parallel software system, developed for executing on the available parallel hardware platform. Each `Body` operates on vectors in Java, instead of `double` arrays, to take advantage of the many possible operations that the Java programming language has available. So, the force summation and time integration are applied to `Body` in Java, as it is shown as follows.

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

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 `Body` components. It is now the moment to insert the sequential processing which corresponds to the operations of the *N-body* simulation and data description found in the problem analysis, This is done in the class `NBodyWorker`, which considers the particular declarations for the *N-body* computation:

```
class NbodyWorker extends MyObject implements Runnable {
   private int N = -1;
   private int id = -1;
   private Body cp;
```

```
   ...
   public NbodyWorker(int id, Body cp) {
      ...
      this.id = id;
      this.cp = new Body();
      new Thread(this).start();
   }
   ...
   public void run() {
      ...
      Body[] system = m.system;
      double imto = m.increment;
      int ps = m.pos;
      Body res = forceT(system,ps);
      res = move(res,imto);
      Body tmp = new Body();
      tmp.setm(res.getm());
      tmp.setr(res.getr());
      tmp.setf(res.getf());
      tmp.setv(res.getv());
      m = new Message(id, true, null, false, tmp, ps, imto);
      ...
   }
}
```

This simple, sequential Java code allows that each `NBodyWorker` component to obtain a local force summation and time inbtegration over a single `Body` is provided. Modifying this code implies modifying the processing behavior of the whole parallel software system, so the class `NbodyManager` can be modified and used for other parallel applications, as long as they are independent computations, and execute on a cluster or a distributed memory parallel computer.

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 *N-body* simulation 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 a distribution of work, executing on a distributed 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] 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] R. Feynman, R.B. Leighton, and M.L. Sands *The Feynman Lectures on Physics*, Vol. 1. Addison-Wesley, 1989.

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

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

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

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

[10] J.L. Ortega-Arjona and G.R. Roberts *Architectural Patterns for Parallel Programming*, Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP98), Kloster Irsee, Germany, 1998.

[11] J.L. Ortega-Arjona *The Communicating Sequential Elements Pattern. An Architectural Pattern for Domain Parallelism*, Proceedings of the 7th Conference on Pattern Languages of Programming (PLoP2000), Allerton Park, Illinois, USA, 2000.

[12] J.L. Ortega-Arjona *The Shared Resource Pattern. An Activity Parallelism Architectural Pattern for Parallel Programming*, Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLoP98), Kloster Irsee, Germany, 1998.

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

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

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

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

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