Architectural Patterns for Parallel Programming

Jorge L. Ortega Arjona and Graham Roberts
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, U.K.
September, 1998

Abstract

This paper introduces an approach to describing and selecting architectural patterns for parallel programming. The approach uses the requirements of the order of data and the computations of the problem, along with the nature of their processing components, to make selections between different architectural patterns.

1. Introduction

Parallel processing is the division of a problem, presented as a data structure or a set of actions, among multiple processing components that operate simultaneously. The expected result is a more efficient completion of the solution to the problem. Its main advantage is the ability to handle tasks of a scale that would be unrealistic or not cost-effective for other systems [CG88, Fos94, ST96, Pan96].

The power of parallelism relies on issues concerned with partitioning a big problem in order to deal with complexity. The partitioning is necessary to divide such big problem into smaller sub-problems that are more easily understood, and may be worked on separately, on a more comfortable scale. This is especially important in the case of parallel systems, where partitioning enables software components to be not only created separately but also executed simultaneously in parallel.

However, in practice parallel programming is hard, involving a steep learning curve. Due to the characteristics of parallel systems, finding a starting point to initiate the design can sometimes be more difficult than compared with sequential programming. Results may also be more discouraging: the programmer's efforts must focus on the design and implementation in new parallel ways, which may end up with a parallel program version that executes slower than a simpler previous sequential one. Furthermore, whether "parallel computer" refers to a high performance platform or a cluster of workstations, its runtime environment is inherently unstable and unpredictable, making it more difficult to decide which organisational structure can be used with less effort and risk of failure. The Architectural Patterns for Parallel Programming presented here represent an attempt to help with the initial selection of the overall structure of a parallel software program.

2. Architectural patterns

Architectural patterns are fundamental organisational descriptions of the common top-level structure observed in a group of software systems. They can be viewed as templates, expressing and specifying structural properties of their subsystems, and the responsibilities and relationships between them. The selection of an architectural pattern is considered to be a fundamental decision during the design of the overall structure of a software system [POSA96, Shaw95].

The initial attraction to architectural patterns is the promise of minimising the effects of imminent changes of requirements on the overall system structure. As they represent a means to capture and express experience in the design and development process of software, their use is expected to be beneficial during early stages of the software life cycle [Shaw95].

The patterns here share a formal structure, containing a name, a summary, examples, a problem statement (including a description of its forces), a solution statement (covering descriptions of its structure, participants, basic dynamics and implementation steps), consequences (describing benefits and liabilities), known uses and related patterns. These elements provide a uniform template for browsing pattern descriptions contained in several pattern systems [PLoP94, PLoP95, GHJV95, POSA96], making it easy to look for and find information about when and how to use each pattern.

3. Classification of Architectural Patterns for Parallel Programming

Architectural patterns for parallel programming can be classified following the characteristics of parallel systems as the classification criteria. Pancake [Pan96], Foster [Fos94], and Carriero and Gelernter [CG88] have studied and proposed classifications according to the characteristics of parallel applications, and their relation with performance. All of them agree that in parallel programming, the nature of the problem to be solved is tightly related to the structure and behaviour of the program that solves it.

Architectural patterns for parallel programming are defined and classified according to the requirements of order of data and operations, and the nature of their processing components.

Requirements of order dictate the way in which parallel computation has to be performed, and therefore, impact on the software design. Following this, it is possible to consider that most parallel applications fall into one of three forms of parallelism: *functional parallelism* [Fos94], *domain parallelism* [Fos94], and *activity parallelism* [CG88], which depend on the requirements of order of operations and data in the problem.

- *Functional parallelism* can be found in problems where a computation can be described in terms of a series of time-step ordered operations, on a series of ordered values or data with predictable organization and interdependencies. As each step represents a change of the input for value or effect over time, a high amount of communication between components in the solution, in the form of a flow of data or operations, should be considered. Conceptually, a single data value or transformation is performed repeatedly [CG88, Fos94, Pan96].

- *Domain parallelism* involves problems in which a set of almost independent operations is to be performed on ordered local data. Because each component in the solution is expected to execute a relatively autonomous computation, the amount of communication between them can be variable, following fixed and predictable paths that can be represented as a network. It is difficult to conceive the computation as a flow of data among processing stages or sequential steps in an algorithm [CG88, Fos94, Pan96].

- *Activity parallelism* involves problems that apply independent computations as sets of non-deterministic transformations, perhaps repeatedly, on values of an ordered or unordered data structure. Activity parallelism can be considered between the extremes of allowing all data to be absorbed by the components or all processes to be divided into components. Many components share access to pieces of a data structure. As each component performs independent computations, communication between processing components is not required. However, the amount of communication is not zero. Communication is required between a component that controls the access of components to the data structure and the processing components [CG88, Pan96].

The nature of processing components is another classification criteria that can be used for parallel systems. Generally, components of parallel systems perform coordination and processing activities. Considering only the processing characteristic of the components, parallel systems are classified as *homogenous systems* and *heterogeneous systems*, according to the same or different processing nature of their components. This nature exposes properties that have tangible effects on their number in the system and the kind of communications among them.

- *Homogeneous systems* are based on identical components interacting in accordance with simple sets of behavioural rules. They represent instances with the same behaviour. Individually, any component can be swapped with another without noticeable change in the operation of the system. Usually, homogeneous systems have a large number of components, which communicate using data exchange operations.

- *Heterogeneous systems* are based on different components with specialised behavioural rules and relations. Basically, the operation of the system relies on the differences between components, and therefore, no component can be swapped with another. In general, heterogeneous systems are composed of fewer components than homogeneous systems, and communicate using function calls.

Based on these classification criteria, this paper presents five architectural patterns commonly used in parallel systems programming:

- The *Pipes and Filters* pattern is an extension of the original *Pipes and Filters* pattern [POSA96, Shaw95, SG96] for parallel systems, using a functional parallelism approach where computations follow a precise order of operations on ordered data. Commonly, each component represents a different step of the computation and data is passed from one computation stage to another along the whole structure.

- The *Parallel Hierarchies* pattern extends the original approach of the *Layers* Pattern [POSA96, Shaw95, SG96] for parallel systems, considering a functional parallelism in which the order of operations on data is the important feature. Parallelism is introduced when two or more hierarchies of layers are able to run simultaneously, performing the same computation.

- The *Communicating Sequential Elements* pattern is used when the design problem at hand can be understood in terms of domain parallelism. The same operations are performed simultaneously on different pieces of ordered data [Fos94, CT92]. Operations in each component depend on partial results in neighbour components. Usually, this pattern is presented as a network or logical structure, conceived from the ordered data.

- The *Manager-Workers* pattern can be considered as a variant of the *Master-Slave* pattern [POSA96] for parallel systems, introducing an activity parallelism where the same operations are performed on ordered data. Each component performs the same operations, independent of the processing activity of other components. Different pieces of data are processed simultaneously. Preserving the order of data is the important feature.

- The *Shared Resource* pattern is a specialisation variant of the *Blackboard* pattern [POSA96] introducing activity parallelism characteristics, where different computations are performed simultaneously, without a prescribed order, on ordered data.

	Functional Parallelism	Domain Parallelism	Activity Parallelism
Heterogeneous processing	*Pipes and Filters*		*Shared Resource*
Homogeneous processing	*Parallel Hierarchies*	*Communicating Sequential Elements*	*Manager-Workers*

Table 1: Architectural patterns classification**.**

4. Selection of Architectural Patterns for Parallel Programming

The selection of an architectural pattern for parallel programming is guided mainly by the properties used for classifying them. However, it is important to notice that a particular architectural pattern, or its combination with others, is not a complete parallel software architecture. Its objective is just to provide a stable structure for a software system, as a first step on the design and implementation of parallel software.

Based on the classification schema and the pattern descriptions [POSA96, GHJV95], a procedure for selecting architectural pattern for parallel programming can be specified as follows:

1. *Analyse the design problem and obtain its specification.* Analyse and specify, as precisely as possible, the problem in terms of its characteristics of order of data and computations, the nature of its processing components, and performance requirements. It is important to also consider the context conditions about the chosen parallel platform and language (see section 5) that may influence the design. This stage is crucial to set up most of the basic forces to deal with during the design.

2. *Select the category of parallelism.* In accordance with the problem specification, select the category of parallelism - functional, domain or activity parallelism - that better describes it.

3. *Select the category of nature of processing components*. Select the nature of the process distribution - homogeneous or heterogeneous - among components that better describes the problem specification. The nature of process distribution indirectly reflects characteristics about the number of processing components and the amount and kind of communications between them in the solution.

4. *Compare the problem specification with the Problem section*. The categories of parallelism and nature of processing components can be simply used to guide the selection of an architectural pattern. In order to verify that the selected pattern copes with the problem at hand, compare the problem specification with the *Problem* section of the selected pattern. More specific information and knowledge about the problem to be solved is required. Unless troubles were encountered up to here, the architectural pattern selection can be considered as completed. The design of the parallel software system continues using the selected architectural pattern *Solution* section as a starting point. On the contrary, if the architectural pattern selected does not satisfactorily match aspects of the problem specification, it is possible to try to select an alternative pattern, as follows.

5. *Select an alternative architectural pattern*. If the selected pattern does not match the problem specification at hand, try to select another pattern that alternatively may provide a better approach when it is modified, specialised or combined with others. Checking the *Examples, Known Uses* and *Related Patterns* sections of the other pattern description may be helpful for this. If an alternative pattern is selected, return to the previous step to verify it copes with the problem specification.

If the previous steps do not provide a result, even after trying some alternative patterns, stop searching. The architectural patterns here do not provide a pattern that can help to solve this particular problem. It is possible to look at other more general pattern languages or systems [GHJV95, PLoP94, PLoP95, POSA96] to see if they contain a pattern that can be used. Or the alternative is trying to solve the design problem without using patterns.

5. Architectural Patterns for Parallel Programming

Parallel programming is characterised by a growing set of parallel architectures, paradigms and programming languages. This situation makes difficult to propose just one approach containing all the details to design and implement parallel software systems. The architectural patterns proposed here are proposed as an effort to help a programmer to decide a starting point when designing a parallel program. Thus, all the patterns contained here share similarities in their context and implementation.

Context and Implementation in general

Context in general

Start the design a software program for a parallel system, using a certain programming language for certain parallel hardware. Consider the following context assumptions:

- The problem involves tasks of a scale that would be unrealistic or not cost-effective for other systems to handle and lends itself to be solved using parallelism.
- The hardware platform or machine to be used is fixed, offering a reasonably good fit to the parallelism found in the problem.
- The language that will be used, based on a certain programming paradigm, is already determined, and a compiler version is available for the parallel platform.

In general, the existence of an available parallel platform and a programming language are considered as a basic context condition when starting the design and implementation of a parallel program. They are determinant of the performance that can be achieved, and also influence the design of software. Once fixed, the decision to use one architectural organisation or another relays mainly on the characteristics of the problem. This work focuses more precisely on these characteristics. Each pattern represents a form to identify how operations can be performed in parallel and/or how data can be operated simultaneously.

Implementation in general

Also, as an effect of the context, the implementation of all these patterns share the same steps, intended to describe an architectural exploratory approach to design, in which hardware-independent features are early considered, and hardware-specific issues are delayed in the implementation process. This method structures the implementation process of parallel software based on four stages [Fos94, CSG97]:

- *Partitioning.* The computation to be performed and/or the data operated are decomposed into operations and/or data pieces, defining possible components for the parallel program. During this stage, practical hardware-dependent issues are ignored. The attention is focussed on recognising the opportunities for parallel execution. In general, architectural pattern components can be implemented using design patterns.
- *Communication.* The communication to coordinate process execution is determined, defining appropriate communication structures between processing components. In general, architectural communication structures can be also based on design patterns.
- *Agglomeration.* The components and communication structures recognised in the previous steps are evaluated in accordance with performance requirements. In the case of parallel systems, usually a conjecture-test approach is used, in which components are recombined several times into larger components, aiming to maximise processor utilisation and reduce communication costs.
- *Mapping*. Components are assigned to real processors, trying to satisfy the attempts of the agglomeration stage. Mapping can be defined static or dynamic, depending directly on hardware issues.

The approach presented here is intended to deal with the implementation issues from an architectural point of view. During the first two stages, attention is focused on concurrency and scalability characteristics. In the last two stages, attention is aimed to shift locality and other performance-related issues. Nevertheless, it is preferred to present each stage in the form of general design considerations instead of provide details about precise implementation of participants. These implementation details are pointed more precisely in the form of references to design patterns for concurrent, parallel and distributed systems of several other authors [Sch95, Sch98a, Sch98b].

Further reference about features of parallel hardware platforms and parallel languages can be found in [CSG97, Fos94, Para98, Perr92, Pfis95, Phil95, ST96]. Also, good advice and guidelines about platform and language selection for performance, related with speed-up and scalability, can be found in [Pan96, PB90].

Pipes and Filters

The *Pipes and Filters* pattern extends the *Pipes and Filters* pattern [POSA96, Shaw95, SG96] with aspects of functional parallelism. Each parallel component performs a different step of the computation, following a precise order of operations on ordered data that is passed from one computation stage to another as a flow through the structure.

Examples

A Non-programming Example

Imagine the assembly line of an automobile factory. There are lots of tasks to manufacture a car, but consider only some very general tasks: 1) frame and power chain installation, 2) bolting the body on the frame, 3) mount the engine, and 4) put on the seats and the wheels. Based on this description of tasks, an assembly line is simply a pipeline of tasks, in which the procedure to manufacture a car requires four tasks that can be overlapped in time: while seats and wheels are put on for a car, the engine is being mounted on another, etc. Ideally, consider that each task is designed so that each completes in a cycle of time. After four cycles, seats and wheels are installed for the first car, the engine is mounted on the second car, the body of the third car is bolted, and the chassis of the fourth car is built. However, from the fifth cycle onwards, the manufacture of a new car requires only a cycle of time to be completed, exploiting the parallelism inherent in the ordered tasks.

A Programming Example

In image processing, rendering is a jargon word that has come to mean "the collection of operations necessary to project a view of an object or a scene onto a view surface" [Watt93]. The best way to break down the overall rendering process is to consider each object as a different coordinate space, in which independent operations can be carried out simultaneously. The input to a polygonal renderer is a list of polygons and the output is a colour for each pixel on the screen. To build up, for instance, a 3D scene, four general tasks are to be performed per object (Figure 1).

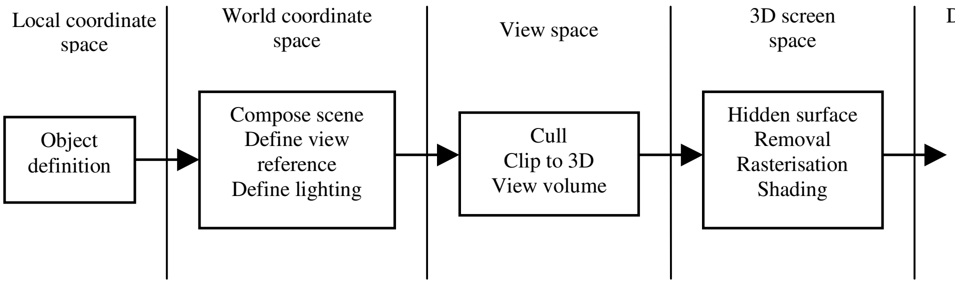

Figure 1. A 3D rendering pipeline.

As in the non-programming example, each one of the tasks can be overlapped in time. In common applications for the film and video industry, the time required to render scene usually can be decreased. For example, the rendering of a special effect scene of 10 seconds using a standard resolution of 2048×1536 pixels takes up to 130 hours of processing time on a single high-end Macintosh or PC platform. Using a parallel approach, with a 16 node CYCORE (a Parsytec parallel machine) this process can be reduced to 10.5 hours [HPCN98].

Problem

The application of a series of ordered but independent computations is required, perhaps as a series of time-step operations, on ordered data. Conceptually, a single data object is transformed. If the computations were carried out serially, the output data set of the first operation would serve as input to the operations during the next step, whose output would in turn serve as input to the subsequent step-operations. Generally, performance as execution time is the feature of interest.

Forces

According to the problem description and considering granularity and load balance as other important forces for parallel software design [Fos94, CT92] the following forces should be considered for a parallel version of the *Pipes and Filters* pattern:

- Maintain the precise order of operations.
- Preserve the order of shared data among all operations.
- Consider the introduction of parallelism, in which different step-operations can process different pieces of data at the same time.
- Distribute process into similar amounts among all step-operation.
- Improvement in performance is achieved when execution time decreases.

Solution

Parallelism is represented by overlapping operations through time. The operations produce data output that depends on preceding operations on its data input, as incrementally ordered steps. Data from different steps are used to generate change of the input over time. The first set of components begins to compute as soon as the first data are available, during the first time-step. When its computation is finished, the result data is passed to another set of components in the second time-step, following the order of the algorithm. Then, while this computation takes place on the data, the first set of components is free to accept more new data. The results from the second time-step components can also be passed forward to be operated on by a set of components in a third-step, while now the first time-step can accept more new data, and the second time-step operates on the second group of data, and so forth [POSA96, CG88, Shaw95, Pan96].

Structure

This pattern is called Pipes and Filters since data is passed as a flow from one computation stage to another along a pipeline of different processing elements. The key feature is that data results are passed just one way through the structure. The complete parallel execution incrementally builds up, when data becomes available at each stage. Different components simultaneously exist and process during the execution time (Figure 2).

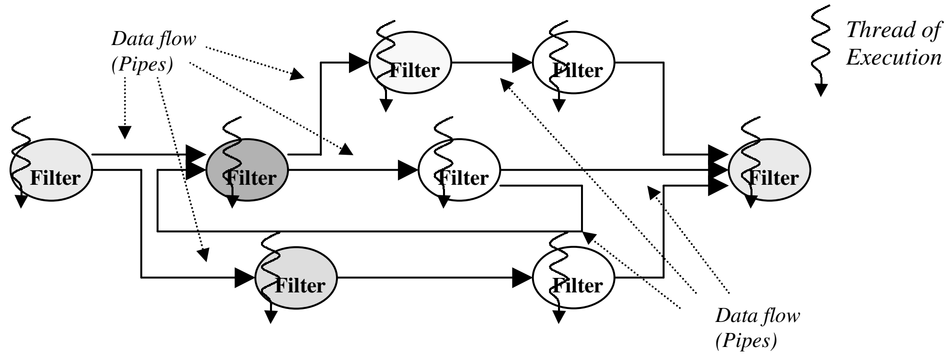

Figure 2. Pipes and Filters pattern.

Participants

- **Filter**. The responsibilities of a filter component are to generate data or get input data from a pipe, to perform an operation on its local data, and to send output result data to one or several pipes.
- **Pipe**. The responsibilities of a pipe component are to transfer data between filters, sometimes to buffer data or to synchronise activity between neighbouring filters.

Dynamics

Due to the parallel execution of the components of the pattern, the following typical scenario is proposed to describe its basic run-time behaviour. As all filters and pipes are active simultaneously, they accept data, operate on it in the case of filters, and send it to the next step. Pipes synchronise the activity between filters. This approach is based on the dynamic behaviour exposed by the *Pipes and Filters* pattern in [POSA96].

In this scenario (figure 3), the following steps are followed:

- **Pipe A** receives data from a Data Source or another previous filter, synchronising and transferring it to the **Filter N**.
- **Filter N** receives the package of data, performs operation *Op.n* on it, and delivers the results to **Pipe B**. At the same time, new data arrives to the **Pipe A**, which delivers it as soon as it can synchronise with **Filter N**. **Pipe B** synchronises and transfers the data to **Filter M**.
- **Filter M** receives the data, performs *Op.m* on it, and delivers it to **Pipe C**, which sends it to the next filter or Data Sink. Simultaneously, **Filter N** has received the new data, performed *Op.n* on it, and synchronising with **Pipe B** to deliver it.
- The previous steps are repeated over and over until no further data is received from the previous Data Source or filter.

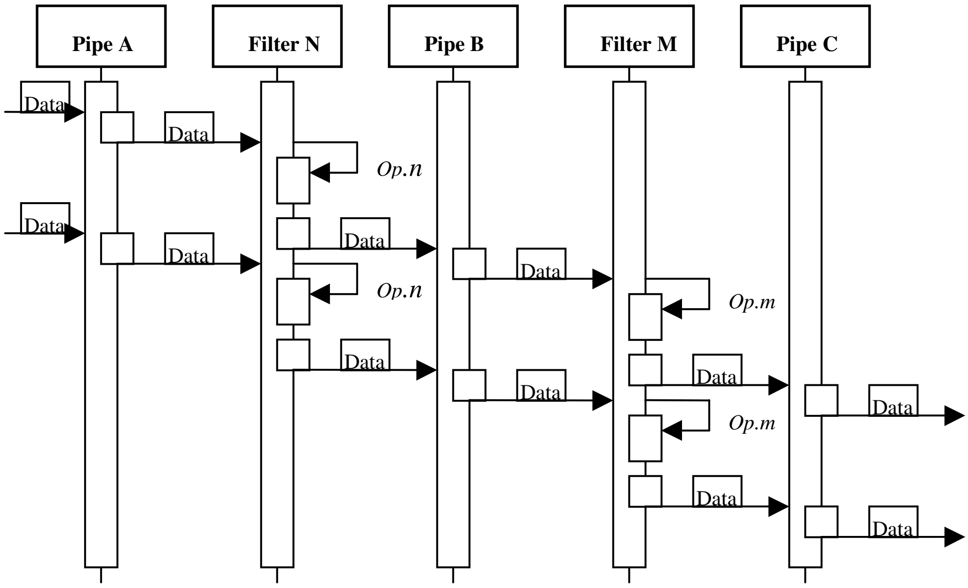

Figure 3. Scenario of Pipes and filters pattern.

Implementation

The implementation process is based on the four stages mentioned above in the *Context and Implementation in general* sections.

- *Partitioning*. The computation that is to be performed is decomposed, attending the ordered operations to be performed into a sequence of different operation stages, in which orderly data is received, operated on and passed to the next stage. Attention focuses on recognising opportunities for simultaneous execution between subsequent operations, to assign and define potential filter components. Initially, filter components are defined by gathering operation stages, considering characteristics of granularity and load-balance. As each stage represents a transformational relation between input/output data, filters can be composed of a single processing element (for instance, a process, task, function, object, etc.) or a subsystem of processing elements. Design patterns [GHJV95, POSA96, PLoP94, PLoP95] can be useful to implement the latter ones; particularly, consider the *Active Object* pattern [LS95] and the *"Ubiquitous Agent"* pattern [JP96].
- *Communication*. The communication required to coordinate the simultaneous execution of stages is determined, considering communication structures and procedures to define the pipe components. Common characteristics that should be carefully considered are the type and size of the data to be passed, and the synchronous or asynchronous coordination schema, trying to reduce the costs of communication and synchronisation. Usually, a synchronous coordination is commonly used in *Pipes and Filters* pattern systems. The implementation of pipe components obeys to features of the programming language used. If the programming language has defined the necessary communication structures for the size and type of the data, a pipe in general can be usually defined in terms of a single communicating element (for instance, a process, a stream, a channel, etc.). However, in case that more complexity in data size and type is required, a pipe component can be implemented as a subsystem of elements, using design patterns. Especially, patterns like the *Broker* pattern [POSA96], the *Composite Messages* pattern [SC95], and those defined in [CMP95] can help to define and implement pipe components.
- *Agglomeration*. The filter and pipe structures defined in the previous stages should be evaluated with respect to the performance requirements and implementation costs. Once initial filters are defined, pipes are considered simply to allow data flow between filters. If an initial proposed agglomeration does not accomplish the expected performance, the conjecture-test approach can be used to propose another

agglomeration schema. Recombining the operations by replacing pipes between them modifies the granularity and load balance, aiming to balance the workload and to reduce communication costs.

- *Mapping*. Each component is assigned to a processor, attempting to maximise processor utilisation and minimise communication costs. Usually, mapping is specified as static for *Pipes and Filters* pattern systems. As a "rule of thumb", these systems may have a good performance when implemented using shared-memory machines, or can be adapted to distributed-memory systems, if the communication network is fast enough to pipe data sets from one filter to the next [Pan96, Pfis95].

Consequences

Benefits

- The use of *Pipes and Filters* pattern allows the description of a computation in terms of the composition of ordered operations of its component filters. Every operation can be understood then in terms of input/output relations of ordered data [SG96].
- Systems based on the *Pipes and Filters* pattern support in a natural form parallel execution. Each filter is considered a separate operation that potentially can be executed simultaneously with other filters [SG96].
- If process is distributed into similar amounts, Pipes and Filters systems are relatively easy to enhance and maintain by filter exchange and recombination. For parallel systems, reuse is enhanced as filters and pipes are composed as active components. Flexibility is introduced by the addition of new filters, and replacement of old filters by improved ones. As filters and pipes present a simple interface, it is relatively easy to exchange and recombine them within the same architecture [POSA96, SG96].
- The performance of Pipes and Filters architectures depends mostly on the number of steps to be computed. Once all components are active, the processing efficiency is constant [POSA96, NHST94].
- Pipes and Filters structures permit certain specialised analysis methods relevant to parallel systems, such as throughput and deadlock analysis [SG96].

Liabilities

- The use of Pipes and filters introduces potential execution problems if they are not properly load-balanced; this is, if the stages do not all present a similar execution speed. As faster stages will finish processing before slower ones, the parallel system will be as fast as its slowest stage. A common solution to this problem is to execute slow stages on faster processors, but load balancing can still be quite difficult. Another solution is to modify the mapping of software components to hardware processors, and test each stage to get a similar speed. If it is not possible to load-balance the work, performance that could potentially be obtained from a Pipes and Filters system may not be worth the programming effort [Pan96, NHST94].
- Synchronisation is another potential problem of Pipes and Filters systems related with load balance. If each stage causes delay during execution, this delay is spread through all the following filters. Furthermore, if feedback to previous stages is used, the whole system tends to slow down after each operation.
- Granularity of Pipes and Filters systems is usually set medium or coarse. This is due to the efficiency of these systems is based on the supposition that pipe communication is a simple action compared to the filters operation. If the time spent communicating tends to be larger than the time required to operate on the flow of data, the performance of the system decreases.
- Pipes and Filters systems can degenerate to the point where they become a batch sequential system, this is each step processes all data as a single entity. In this case, each stage does not incrementally process a stream of data. To avoid this situation each filter must be designed to provide a complete incremental parallel transformation of input data to output data [SG96].
- The most difficult aspect of Pipes and Filters systems is error handling. An error reporting strategy should at least be defined throughout the system. However, concrete strategies for error recovery or handling depend directly on the problem to solve. Most applications consider that if an error occurs, the system either restarts the pipe, or ignores it. If none of these are possible the use of alternative patterns, such as the *Layers* pattern [POSA96] is advised.

Known uses

- The butterfly communication structure, used in many parallel systems to obtain the Fast Fourier Transform (FFT), presents a basic *Pipes and Filters* pattern. Input values are propagated through intermediate stages, where filters perform calculations on data when it is available. The whole computation can be viewed as a flow through crossing pipes that connect filters [Fos94].
- Parallel search algorithms mainly present a Pipes and Filters structure. An example is the parallel implementation of the CYK Algorithm (Cocke, Younger and Kasami), used to answer the membership question: "Given a string and a grammar, is the string member of the language generated by the grammar?" [CG88, NHST94].
- Operations for image processing, like convolution, where two images are passed as streams of data through several filters (FFT, multiplication and inverse FFT) in order to calculate their convolution [Fos94].

Related patterns

The *Pipes and Filters* pattern for parallel programming is presented as an extension of the original *Pipes and Filters* pattern [POSA96] and *Pipes and Filters* architectural style [Shaw95, SG96]. Other patterns that share the similar ordered transformation approach can be found in [PLoP94]; especially consider the *Pipe and Filters* pattern and the *Streams* pattern. Another pattern that can be consulted for implementation issues using C++ is the *Pipeline Design Pattern* [VBT95].

Parallel Hierarchies

The *Parallel Hierarchies* pattern is a parallel extension of the *Layers* pattern approach [POSA96, Shaw95, SG96] with elements of functional parallelism. The order of operations on data is the most important feature. Parallelism is introduced when two or more components of a layer are able to exist simultaneously, performing the same operation. Components can be created statically, waiting for calls from higher layers, or dynamically, when a call triggers their creation.

Examples

A Non-programming Example

Consider the control of hand motion by the brain. Nerves conduct impulses from the brain to muscles, controlling their contraction, and sensing their position. A muscle contracts or relaxes, performing forces on points of the bone structure, modifying its position. However, a coordinated movement is only the result of the simultaneous and ordered contraction or relaxation of several muscles, each one returning information of its position through nerves to the brain. A simple or complex movement is achieved as a result of a request from the brain, triggering several nerves to contract or relax muscles, which at the same time return feedback signals until the bone has a precise position. The hierarchy of nervous, muscular and bone tissues accomplishes the motion of any part of the body.

A Programming Example

Suppose a computer-controlled industrial robot system, which consists of several arms with hands [Fro96, PLoP94]. Each hand grabs or releases objects, and each arm moves its hand around. The structure of the robot program is presented as a hierarchy (Figure 4). In order to manufacture an article, the robot communicates with its hands and arms, activating them and causing them to carry out a set of physical tasks and movements. When a physical hand completes its grab or release operation, the hand sends a message back to its associated arm. Now, the arm knows that it is safe to start moving the physical arm. Similarly, an arm sends back messages to the robot when it has completed its move operation. Arms act simultaneously, until they have manufactured an article.

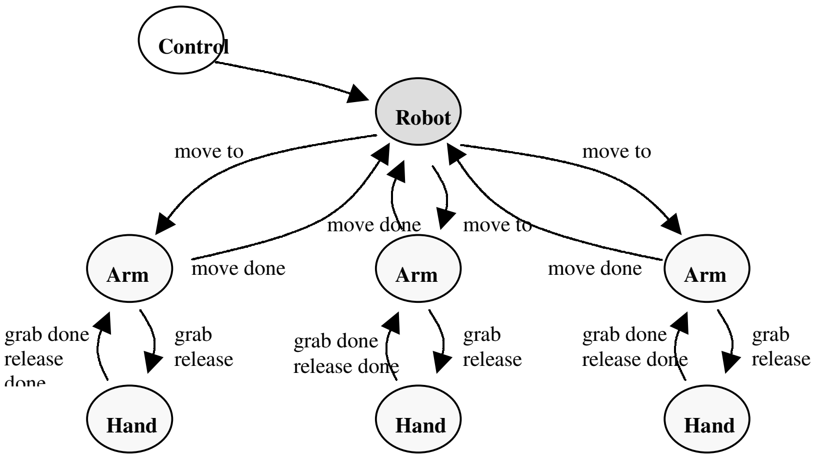

Figure 4. A robot system program hierarchy.

Problem

It is necessary to perform a computation repeatedly, composed of a series of ordered operations on a set of ordered data. Not every problem meets this criterion. Consider a program whose output may be the result of just a single complex computation as a series of conceptually ordered simple operations, executed not for value but for effect, at different levels. An operation at a high level requires the execution of one or more operations at lower levels. If this program is carried out serially, it could be viewed as a chain of subroutine calls, evaluated one after another. Generally, performance as execution time is the feature of interest.

Forces

From the problem description and other additional considerations of granularity and load balance in parallel design [Fos94, CT92], the following forces should be considered for the *Parallel Hierarchies* pattern:

- Perform computation as a hierarchy of ordered operations several times, in a single execution.
- Data is shared among adjacent layers.
- The same group of operations can be simultaneously performed several times on different pieces of data.
- Operations may be different in size and level of complexity.
- Dynamic creation and destruction of components is preferred over static, to achieve load balance.
- Improvement in performance is achieved when execution time decreases.

Solution

Parallelism is introduced by allowing the simultaneous execution of more than one instance per layer through time. In a *Layer* pattern system, when an operation triggers an operation, this may involve the execution of operations in several layers. These operations are usually triggered by a function call, and data is shared in the form of arguments for these function calls. During the execution of operations in each layer, usually the higher layers have to wait for a result from lower layers. However, if each layer is represented by more than one component, they can be executed in parallel and service new requests. Therefore, at the same time, several ordered sets of operations can be carried out by the same system. Several computations can be overlapped in time [POSA96, Shaw95].

Structure

This pattern is composed of conceptually-independent entities, ordered in the shape of hierarchies of layers. Each layer, as an implicit different level of abstraction, is composed of several components that perform the same operation. To communicate, layers use function calls, referring to each other as elements of some composed structure. The same computation is performed by different groups of functionally related components. Components simultaneously exist and process during the execution time (Figure 5).

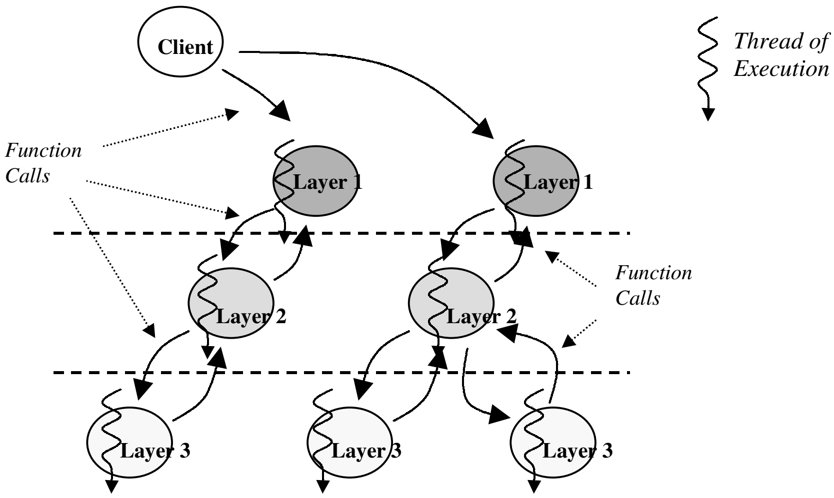

Figure 5. Parallel Hierarchies pattern.

Participants

- **Layer**. The responsibilities of a layer component are to provide operations or functions to more complex level layers, and to delegate more simple subtasks to layers in less complex levels. During run-time, more than one component per layer is allowed to execute concurrently with others.

Dynamics

As the parallel execution of layer elements is allowed, a typical scenario is proposed to describe its basic run-time behaviour. All layer elements are active at the same time, accepting function calls, operating, and returning or sending another function call to elements in lower level layers. If a new function call arrives from the client, a free element of the first layer takes it and starts a new computation.

As stated in the problem description, this pattern is used when it is necessary to perform repeatedly a computation, as series of ordered operations. The scenario presented here takes the simple case when two computations, namely **Computation 1** and **Computation 2**, have to be performed. **Computation 1** requires the operations *Op.A*, which requires the evaluation of *Op.B*, which needs the evaluation of *Op.C*. **Computation 2** is less complex than **Computation 1**, but requires to perform the same operations *Op.A* and *Op.B*. The parallel execution is as follows (figure 6):

- The **Client** calls a component **Layer A1** to perform **Computation 1**. This component calls to a component **Layer B1**, which similarly calls a component **Layer C1**. Both components **Layer A1** and **Layer B1** remain blocked waiting to receive a return message from their respective sub-layers. This is the same behaviour than the sequential version of the *Layers* pattern [POSA96].
- Parallelism is introduced when the **Client** issues another call for **Computation 2**. This cannot be serviced **by Layer A1**, **Layer B1** and **Layer C1**. Another instance of the component in Layer A, called **Layer A2** - that either can be created dynamically or be waiting for requests statically - receives it and calls another instance of Layer B, **Layer B2**, to service this call. Due to the homogeneous nature of the components of each layer, every component in a layer can perform exactly the same operation. That is precisely the advantage of allowing them to operate in parallel. Therefore, any component in Layer B is capable to serve calls from components in Layer A. As the components of a layer are not exclusive resources, it is in general possible to have more than one instance to serve calls. Coordination between components of different layers is based on a kind of client/server schema. Finally, each component operates with the result of the return message. The main idea is that all computations are performed in a shorter time.

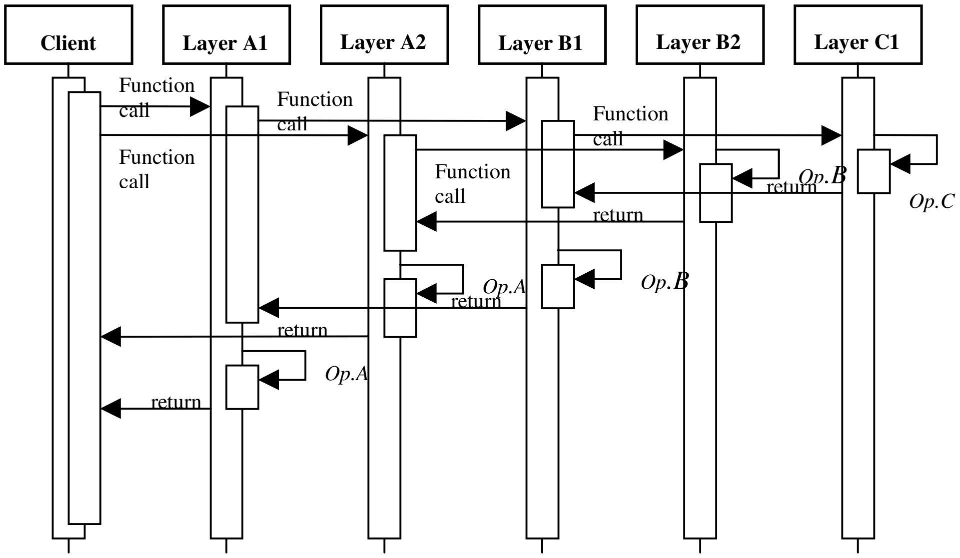

Figure 6. Scenario for Parallel Hierarchies pattern.

Implementation

The implementation process is based on the four stages mentioned above in the *Context and Implementation in general* sections.

- *Partitioning*. Initially, it is necessary to define the basic *Layer* pattern system which will be used with parallel instances: the computation to be performed is decomposed into a set of ordered operations, hierarchically defined and related, determining the number of layers. Following this decomposition, the component representative of each layer can be defined. For a concurrent execution, the number of components per-layer depends on the number of requests to be serviced. Several design patterns have been proposed to deal with layered systems. Advice and guidelines to recognise and implement these systems can be found in [POSA96, PLoP94]. Also, consider the patterns used to generate hierarchies, like *A Hierarchy of Control Layers* [AEM95] and the *Layered Agent Pattern* [KMJ96].
- *Communication*. The communication, required to coordinate the parallel execution of layer components, is determined by the services that each layer provides. Characteristics that should be carefully considered are the type and size of the shared data to be passed as arguments and return values, the interface for layer components, and the synchronous or asynchronous coordination schema. In general, an asynchronous coordination is preferred over a synchronous one. The implementation of communication structures between components depends on the features of the programming language used. Usually, if the programming language has defined the communication structures (for instance, function calls or remote procedure calls), the implementation is very simple. However, if the language does not support communication between remote components, it is proposed the construction of an extension in the form of a communication subsystem. Design patterns can be used for this. Particularly, patterns like the *Broker* pattern [POSA96], the *Composite Messages* pattern [SC95], the *Service Configurator* pattern [JS96] and the *Visibility and Communication between Control Modules* and *Actions Triggered by Events* [AEM95] can help to define and implement the required communication structures.
- *Agglomeration*. The hierarchical structure is evaluated with respect to the expected performance. Usually, systems based on identical hierarchies present a good load-balance. However, if necessary, using the conjecture-test approach, layer components can be refined by combination or decomposition of operations, modifying their granularity to improve performance or to reduce development costs.
- *Mapping*. In the best case, each layer component executes simultaneously on a different processor, if enough processors are available. Usually this is not the case. An approach proposes to execute each hierarchy on a processor, but if the number of requests is large, some hierarchies would have to block, keeping the client(s) waiting. Another mapping proposal attempts to place every layer on a processor.

This simplifies the restriction about the number of requests, but if not all operations require all layers, this may overcharge the some processors, introducing load-balance problems. The most realistic approach seems to be a combination of the both previous ones, trying to maximise processor utilisation and minimise communication costs. In general, mapping of layers to processors is specified static, allowing an internal dynamic creation of new components to serve new requests. As a “rule of thumb”, a *Parallel Hierarchies* pattern system will perform best on a shared-memory machine, but a good performance can be achieved if it can be adapted to a distributed-memory system with a fast communication network [Pan96, Pfis95].

Consequences

Benefits

- The *Parallel Hierarchies* 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 [SG96]. Allowing each layer to be presented as multiple components executing in parallel allows to perform the computation several times, enhancing performance.
- 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, standardising the interfaces between layers usually confines the effect of changes exclusively to the layer that is changed. [POSA96, SG96].
- Layers support reuse. If a layer represents a well-defined operation, and communicates via a standardised interface, it can be used interchangeably in multiple contexts. A layer can be replaced by a semantically equivalent layer without great programming effort [POSA96, SG96].
- 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.
- Due to several instances of the same computation are executed independently on different data, synchronisation 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 [Pan96].

Liabilities

- Not every system computation can be 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 [SG96, Pan96].
- Many times, a layered system is not as efficient as a structure of communicating elements. 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 [POSA96].
- 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 [POSA96, SG96, NHST94]
- If the level of complexity of the layers is not correct, problems can arise when the behaviour 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 [POSA96].

Known uses

- The homomorphic skeletons approach, developed from the Bird-Meertens formalism and based on data types, can be considered as an example of the *Parallel Hierarchies* pattern: individual computations and communications are executed by replacing functions at different levels of abstraction [ST96].
- Tree structure operations like search trees, where a search process is created for each node. Starting from the root node of the tree, each process evaluates its associated node, and if it does not represent a solution,

recursively creates a new search layer, composed of processes that evaluate each node of the tree. Processes are active simultaneously, expanding the search until they find a solution in a node, report it and terminate [Fos94, NHST94].

- The Gaussian elimination method, used to solve systems of linear equations, is a numerical problem that can be solved using a Parallel Hierarchies structure. The original system of equations, expressed as a matrix, is reduced to a triangular form by performing linear operations on the elements of each row as a layer. Once the triangular equivalent of the matrix is available, other arithmetic operations must be performed by each layer to obtain the solution of each linear equation [Fos94].

Related patterns

The *Parallel Hierarchies* pattern extends the *Layers* pattern [POSA96] and the Layers style [Shaw95, SG96] for parallel systems. Several other related patterns are found in [PLoP94]; more precisely, *A Hierarchy of Control Layers* pattern, *Actions Triggered by Events* pattern, and those under the generic name of *Layered Service Composition pattern*.

Communicating Sequential Elements

The *Communicating Sequential Elements* pattern is a domain parallelism pattern where each component performs the same operations on different pieces of ordered data [Fos94, CT92]. Operations in each component depend on partial results in neighbour components. Usually, this pattern is presented as a logical structure, conceived from the ordered data.

Examples

A Non-programming Example

Consider the case of a hive of bees. The workers act as constructors, harvesters, soldiers, and nursemaids. Each individual is capable to eventually perform any of these activities in cooperation with the others during its short lifetime. Minute-to-minute control of the hive's behaviour is dispersed. Changes, like many shifts in foraging patterns of honey bees come from signals among the workers, whose success or failure at various tasks can rise hive-permeating hormone levels that then brings about changes in the whole hive.

A Programming Example

Consider the case of data mining problems, in which government and businesses organisations require information processing to acquire and analyse data about customers information, products, taxes, etc., devoting a lot of computational effort to automatically extract useful information or "knowledge" from these data. A particular type of data mining is mining for associations [CSG97, AS96], in which it is important to discover relations or associations among the information to generate rules for the inference of behaviour. For example, given a database in which records correspond to customer purchase transactions, the goal in mining for associations is to determine which sets of a number of items occur together in more than a given threshold fraction of these transactions.

A proposed solution uses an algorithm of several passes over the database. The first pass simply counts item occurrences to determine sets of items with frequency 1. A subsequent pass, say k, consists of two parts or phases: first the sets of items with frequency k-1 are used to generate a set of candidates, say C_k, using a certain generation procedure. Next, the database is scanned and the support of candidates in C_k is counted. Fast counting is required to efficiently determine the candidates in C_k contained in a given transaction. Data distribution is used for this. Each processing element counts mutually exclusive local candidates. With a large number of elements, a large number of candidates can be counted in a pass. However, as part of the procedure, every element must broadcast its local data to all other elements at the end of every pass. Finally, after k passes, the results are available for analysis [CSG97, AS96].

Problem

A computation is required that can be performed as a set of quasi-independent operations on ordered data. Results cannot be constrained to a one-way flow among processing stages, but each component executes a relatively autonomous computation. Communications between components follow fixed and predictable paths. Consider, for example, a dynamics problem simulation: the data represents a model of a real system, where any change or modification in one region influences areas above and below it, and perhaps to a different extent, those on either side. Over time, the effects propagate to other areas, extending in all directions; even the source area may experience reverberations or other changes from neighbouring regions. If this simulation would be executed serially, it would require that computations be performed across all the data to obtain some intermediate state, and then, a new iteration should begin. Generally, performance as execution time is the feature of interest.

Forces

Considering the problem description and granularity and load balance, as other elements of parallel design [Fos94, CT92], the following forces should be considered:

- Preserve the precise order of data distributed among processing elements.
- Computations are performed autonomously, on local pieces of data.
- Every element performs the same operations, in number and complexity.
- Partial results are usually communicated among neighbour processing elements.
- Improvement in performance is achieved when execution time decreases.

Solution

Parallelism is introduced as multiple participating concurrent components, each one applying the same set of operations on a data subset. Components communicate partial results by exchanging data, usually through communication channels. No data objects are directly shared among components; each one may access its own private data subset only. A component communicates by sending data objects from its local space to another. This communication may have different variants: synchronous or asynchronous, exchange of a single data object or a stream of data objects, and one to one, one to many, many to one or many or many communications. Often the data of the problem can be conceived in terms of an ordered logical structure. The solution is presented as a network that may reflect this logical structure in a transparent and natural form [CG88, Shaw95, Pan96].

Structure

In this pattern, the same operation is almost simultaneously applied to different pieces of data. However, operations in each element depend on the partial results of operations in other components. The structure of the solution involves an ordered logical structure, conceived from the data structure of the problem. Therefore, the solution is presented as a network of elements that in general follows the shape imposed by this structure. Identical components simultaneously exist and process during the execution time (Figure 7).

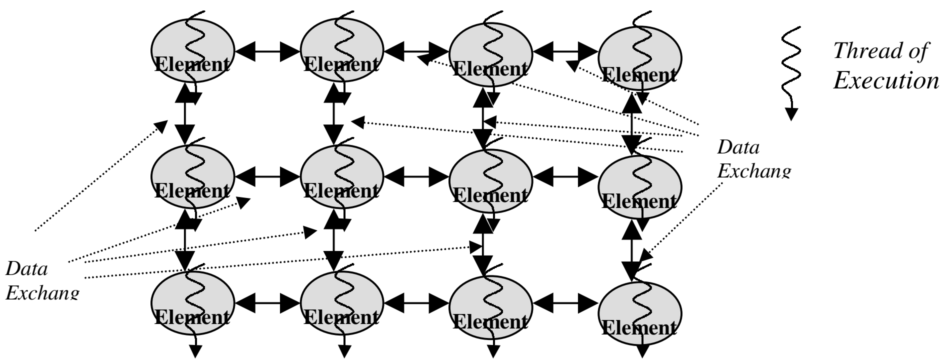

Figure 7. Communicating Sequential Elements.

Participants

- **Sequential element**. The responsibilities of a processing element are to perform a set of operations on its local data, and to provide a general interface for sending and receiving messages.
- **Communication channels**. The responsibilities of a communication channel are to represent a medium to send and receive data between concurrent elements, and to synchronise communication activity between them.

Dynamics

A typical scenario to describe the basic run-time behaviour of this pattern is presented, where all the Sequential Elements are active at the same time. Every Sequential Element performs the same operations, as a piece of a processing network. In the most simple case, each one communicates only with a previous and next others (figure 8). The processing and communicating scenario is as follows:

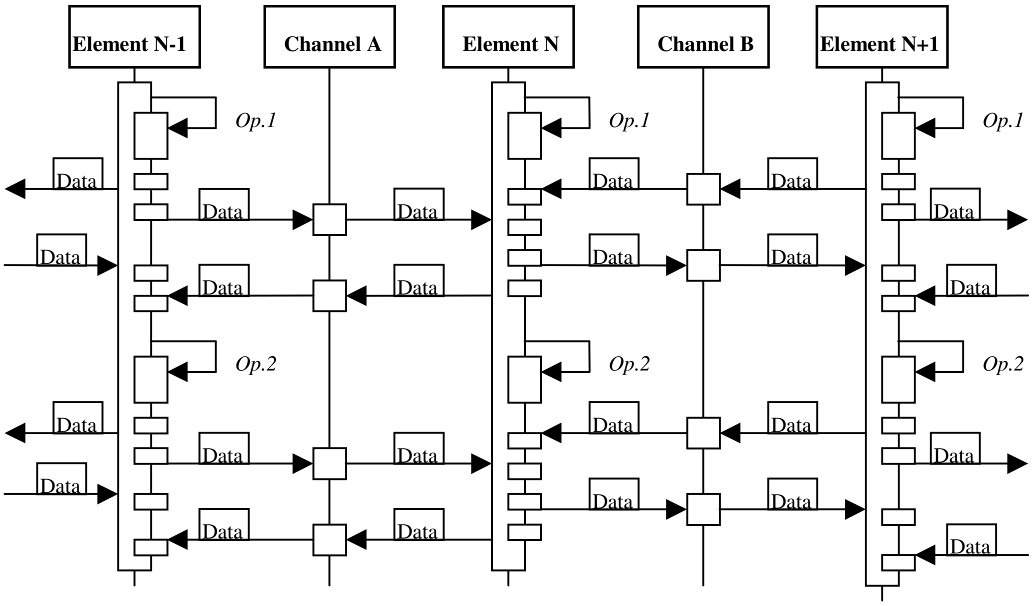

Figure 8. Scenario of Communicating Sequential Elements.

- A computation is started when all components **Element N-1**, **Element N**, **Element N+1**, etc. perform at the same time operation *Op.1*.
- To continue the computation, all component communicate their partial results through the communication channels available (Here, **Channel A** and **Channel B**). Then all components synchronise and receive the partial results from their previous and next neighbours.
- Once synchronisation and communications are finished, each component continues computing the next operation (in this case *Op.2*). The process repeats until each component has finished its computations.

Implementation

The implementation process is based on the four stages mentioned above in the *Context and Implementation in general* sections.

- *Partitioning*. The ordered logical structure of data is a natural candidate to be initially decomposed into a network of data sub-structures or pieces. Depending on the precision required, data pieces have different size and shape. However, in order to maintain the process load-balance, they normally present the same size and shape. Trying to expose the maximum concurrency, the basic sequential element is defined to process a unique sequence of operations on only one data piece. Hence, the total number of sequential elements is equal to the number of data pieces, and each computation presents the same complexity per time step. While each sequential element performs the same operations on different data pieces, they share the same processing nature and structure. However, they can be represented by a single processing element (for instance, a process, task, function, object, etc.) or a subsystem of processing elements, which may be designed using design patterns [GHJV95, POSA96, PLoP94, PLoP95]. Some design patterns that particularly can be considered for implementing concurrent components are the *Active Object* pattern [LS95] and the *"Ubiquitous Agent"* pattern [JP96].

- *Communication*. The communication is also associated with the network of data pieces. Each sequential element is expected to exchange partial results with its neighbours from time to time through channels. Channels should perform data exchange and coordinate the operation execution appropriately. An efficient communication depends on the amount and format of the data to be exchanged, and the synchronisation schema used. Both synchronous and asynchronous schemes can be found in several parallel systems, but a synchronous schema is commonly preferred. An important issue to consider here is how communication channels are defined. In general, this decision is linked with the programming language used. Some languages define precisely a type channel where it is possible to send and to receive values. Any sequential element is defined to write on the channel, and to read from it. No further implementation is necessary. On the other hand, other languages do not define the channel type. Thus, it should be designed and implemented in such a way that allows data exchange between elements. As the use of channels depends on the language, decisions about their implementation are delayed to other refining design stages. From an architectural point of view, channels are defined, whether they are implicit in the language, or they need to be explicitly created. Design patterns that can help with the implementation of channel structures are the *Composite Messages* pattern [SC95] and the *Service Configurator* pattern [JS96].
- *Agglomeration.* The sequential elements and channels defined are evaluated with respect to an expected performance. Often, the number of processing and communicating elements is bigger than what is required, and some degree of agglomeration can be considered. Causes of agglomeration can be redundant communications, and the amount of communications in a dimension or direction. As each sequential element performs the same operations, changes in the granularity involve only the size of the amount of data pieces in the network to be processed per component. If they maintain the same granularity, the structure is normally load balanced. The conjecture-test approach is only used then to modify the granularity to achieve a better performance.
- *Mapping*. In the most optimistic case, each sequential element is assigned to a processor. However, usually the number of processors is a number of times less than the number of processing elements. Taking this number as the number of elements per processor, it is possible to assign them in a more realistic form. The important feature to maximise processor utilisation and minimise communication costs, is balance. In these structures, computational efficiency is decreased due to load imbalances. If the design is to be used extensively, it is worthwhile to improve its load balance. Approaches to do this can use cyclic mapping or dynamic mapping. As a "rule of thumb", systems based on the *Communicating Sequential Elements* pattern will perform best on a SIMD (single-instruction, multiple-data) computer, if array operations are available. However, if the computations are relatively independent, a respectable performance can be achieved using a shared-memory system [Pan96].

Consequences

Benefits

- Data order and integrity is granted, due to each sequential element accesses only its own local data subset, and there is no data directly shared among components [SG96, ST96].
- As all sequential elements share the same functional structure, their behaviour can be modified or changed without great effort [SG96, ST96].
- It is relatively easy to structure the solution in a transparent and natural form as a network of elements, reflecting the logical structure of data in the problem [CG88, Shaw95, Pan96].
- As all components perform the same computation, granularity is independent of functionality, depending only on the size and number of the elements in which data is divided. It is easily to change in case a better resolution or precision is required.

- This pattern can be used on most hardware systems, considering the synchronisation characteristic between elements as the only restriction [Pan96].

Liabilities

- The performance of systems based on communicating elements is heavily impacted by the communication strategy (global or local) used. Usually, the processors available are not enough to support all elements. In order to apply a computation, each processor operates on a subset of the data. Due to this, dependencies between data, expressed as communications, can slow down the program execution [Fos94, Pan96].
- In the use of this pattern load balance is still a difficult problem. Often, data is not easily divided into same size subsets of data and the computational intensity varies on different processors. To maintain synchronisation means that fast processors must wait until the slow ones can catch up, before the computation can proceed to the next set of operations. An inadequate load balance impacts strongly on performance. The decision to use this pattern should be based on how uniform in almost every aspect can the system be [Pan96].
- The synchronous characteristic of the application determines efficiency. If the application is synchronous, a significant amount of effort is required to get a minimal increment in performance. If the application is asynchronous, it is more difficult to parallelise, and probably the effort will not be worthwhile, unless communications between processors are very infrequent [Pan96].

Known uses

- The one-dimensional wave equation, used to numerical model the motion of vibrating systems, is an example of the *Communicating Sequential Elements* pattern. The vibrating system is divided in sections, and each processing element is responsible for the computation of the position at any moment of a section. Each computation depends only on partial results of the computation at neighbouring sections. Thus, each computation can be done independently, except when data is required from the previous or next sections [NHST94].
- Simulation of dynamic systems, such as an atmosphere model, is another use of Communicating Sequential Elements. The model usually is divided as a rectangular grid of blocks. The simulation proceeds in a series of time steps, where each processing element computes and updates the temporal state in a block with data of the previous state and updates of the state of neighbouring blocks. Integrating the time steps and the blocks makes possible to determine the state of the dynamic system at some future time, based on an initial state [Fos94].
- Image processing problems, such as the component labelling problem. An image is given as a matrix of pixels, and each pixel must be labelled according to certain property - for instance, connection. The image is divided in sub-images, and mapped to a network of processing elements. Each processing element tests for connection, and labels all the non-edge pixel of its sub-image. Edge pixels between sub-images are labelled in cooperation by the two respective processing elements [Fos94].

Related patterns

The *Communication Sequential Elements* pattern is based on the original concept of Communicating Sequential Processes (CSP) [Hoare84]. Patterns that can be considered related to this processing approach are the *Ubiquitous Agent* Design Pattern [JP96] and the *Visibility and Communication between Agents* pattern [ABM96].

Manager-Workers

The *Manager-Workers* pattern is a variant of the *Master-Slave* pattern [POSA96] for parallel systems, considering an activity parallelism approach where the same operations are performed on ordered data. The variation is based on the fact that components of this pattern are proactive rather than reactive [CT92]. Each processing component simultaneously performs the same operations, independent of the processing activity of other components. An important feature is to preserve the order of data.

Examples

A Non-programming Example

Suppose a simple scheme for a repair centre, involving a manager and a group of technicians. The manager is responsible for receiving articles, and assigning an article to be repaired to a technician. All technicians have similar skills for repairing articles, and each one is responsible to repair one article at a time, independent of the other technicians. When a technician finishes repairing his assignment, he notifies the manager; the manager then assigns him a new article to be repaired, and so on. In general, repairing articles represents an irregular problem: some articles may present a simple fix and take a little amount of time, while others may require a more complex repair. Also, the effectiveness of this scheme relies on the fact that the number of articles that arrive to the centre can be substantially larger than the number of technicians available.

A Programming Example

Consider a real-time ultrasonic imaging system [GSVOM97] designed to acquire, process and display a tomographic image. Data is acquired based on the reflection of an ultrasonic signal that excites an array of 56 ceramic sensors. Data is amplified and digitalised to form a black and white image of 56×256 pixels, each one represented by a byte. An interpolation program is required to process the image, enlarging it to make it clearer to the observer. The image is displayed on a standard resolution monitor (640×480 pixels) in real-time, this is, at least 25 frames per second. In accordance with these requirements, an interpolation by a factor 3 between columns was chosen, enlarging the information of each image three times. A calculation shows the volume of data to be processes per second: each frame is represented as 168×256×1 bytes, and using 25 frames per second, makes a total of 1.075200 Mbytes per second. Using a manager-worker system for the cubic interpolation, the image is received a stream of pixels by the manager, which assigns to each worker a couple of pixels. Each worker uses each couple of pixels as input data, and calculates the cubic interpolation between them, producing other four interpolated pixels. As the number of workers is less than the total number of pixels, each worker requests for more work to the manager as soon as it finishes its process, and so on.

Problem

A computation is required where independent computations are performed, perhaps repeatedly, on all elements of some ordered data. Each computation can be performed completely, independent of the activity of other elements. Data is distributed among components without a specific order. However, an important feature is to preserve the order of data. Consider, for example, an imaging problem, where the image can be divided into smaller sub-images, and the computation on each sub-image does not require the exchange of messages between components. This can be carried out until completion, and then the partial results gathered. The overall affect is to apply the computation to the whole image. If this computation is carried out serially, it should be executed as a sequence of serial jobs, applying the same computation to each sub-image one after another. Generally, performance as execution time is the feature of interest.

Forces

Using the previous problem description and other elements of parallel design, such as granularity and load balance [Fos94, CT92], the following forces are found:

- Preserve the order of data. However, the specific order of data distribution and operation among processing elements is not important
- The same computation can be performed independently and simultaneously on different pieces of data.
- Data pieces may exhibit different sizes.
- Changes in the number of processing elements should be reflected by the execution time.
- Improvement in performance is achieved when execution time decreases.

Solution

Parallelism is presented by having multiple data sets processed at the same time. The most flexible representation of this is the *Manager-Workers* pattern approach. This structure is composed of a manager component and a group of identical worker components. Each worker is capable of performing the same computation. It repeatedly seeks a task to perform, performs it and repeats; when no tasks remain, the program is finished. The execution model is the same, independent of the number of workers (at least one). If tasks are distributed at run time, the structure is naturally load balanced: while a worker is busy with a heavy task, another may perform several shorter tasks. To preserve data integrity, the manager program takes care of what part of the data has been operated on, and what remains to be computed by the workers [POSA96, CG88, Shaw95, Pan96, CT92].

Structure

The *Manager-Workers* pattern is represented by a manager, preserving the order of data and controlling a group of processing elements or workers. Conceptually, workers have access to different pieces of data at the same time. Usually, only one manager and several identical worker components simultaneously exist and process during the execution time (Figure 9).

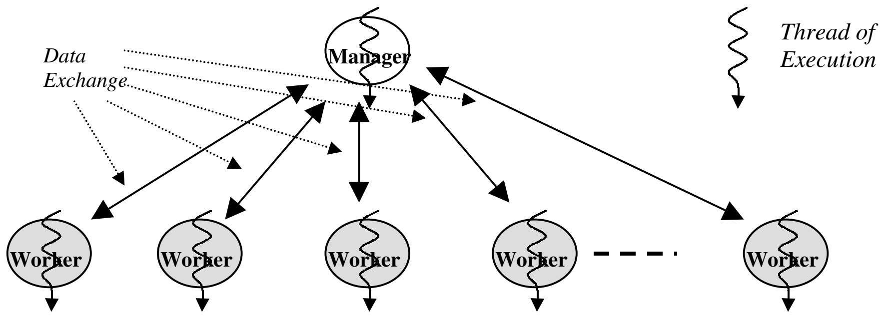

Figure 9. Manager-Workers pattern.

Participants

- **Manager**. The responsibilities of a manager are to create a number of workers, to partition work among them, to start up their execution, and to compute the overall result from the sub-results from the workers.
- **Worker**. The responsibility of a worker is to seek for a task, to implement the computation in the form of a set of operations required by the manager, and to perform the computation.

Dynamics

A typical scenario to describe the run-time behaviour of the *Manager-Worker* pattern is presented, where all participants are simultaneously active. Every worker performs the same operation on its available piece of data. As soon as it finishes processing, returns a result to the manager, requiring for more data. Communications are restricted between the manager and each worker. No communication between workers is allowed (figure 10).

In this scenario, the steps to perform a set of computations is as follows:

- All participants are created, and wait until a computation is required to the manager. When data is available to the manager, this divides it, sending data pieces to each waiting worker.
- Each worker receives the data and starts processing an operation *Op* on it. This operation is independent to the operations on other workers. When the worker finishes processing, it returns a result to the manager, requiring for more data. If there is still data to be operated, the process repeats.

- The manager is usually replying to requests of data from the workers or receiving their partial results. Once all data pieces have been processed, the manager assembles a total result with the partial results and the program finishes. The non-serviced requests of data from the workers are ignored.

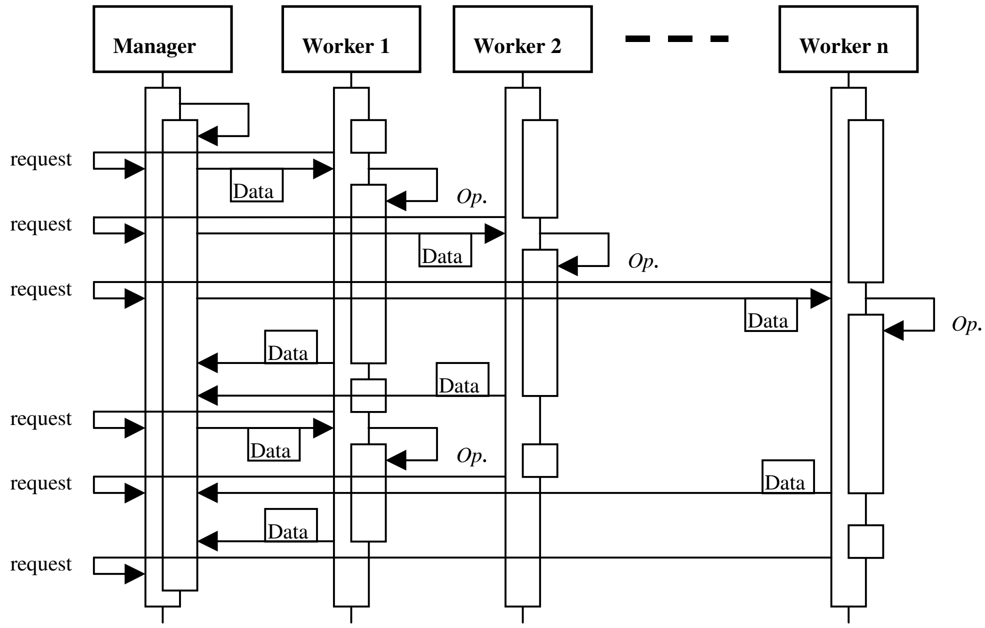

Figure 10. Scenario for the Manager-Workers pattern.

Implementation

The implementation process is based on the four stages mentioned above in the *Context and Implementation in general* sections.

- *Partitioning*. The ordered data to be operated on by the common computation is decomposed into a set of data pieces. This partitioning criteria of the ordered data is a clear opportunity for parallel execution, and it is used to define the partitioning and gathering activity of the manager component. On the other hand, the same computation to be performed on different data pieces is used to define the structure of each one of the worker components. Sometimes, the manager is also implemented to perform the computation on data pieces as well. Usually, the structure of the manager component can be reused if it is designed to deal with different data types and sizes, delimiting its behaviour to divide, deliver and gather data pieces to the worker components. It is possible to implement a manager component using a single element approach (for instance, a process, a task, a function, an object, etc.), or to define a sub-system of elements that perform manager activities. Usually, concurrency the sub-system elements can be used, defining different interfaces for different actions. Design patterns [GHJV95, POSA96, PLoP94, PLoP95] can help to define and implement such interfaces. On the other hand, worker components can be also defined either by single elements or by sub-systems of elements. Patterns that particularly can help with the design and implementation of the manager and worker components are the *Active Object* pattern [LS95] and the *Service Configurator* pattern [JS96]. In the case of the worker components, other design patterns that may provide information about their implementation are the *"Ubiquitous Agent"* pattern [JP96] and the *Object group* pattern [Maf96].
- *Communication*. The communication structure that coordinates the execution between the manager and worker should be defined. As workers are just allowed to communicate with the manager to get more work, defining an appropriate communication structure between manager and worker components is a key task. The communication structure should allow the interactions between the manager and each worker (request a data piece and, once processed, its delivery to the manager). Important parameters to consider are the size and format of data, the interface to service a request of data, and the synchronisation criteria. In general, a synchronous coordination is commonly used in *Manager-Worker* pattern systems.

The implementation of communication structures depends on the programming language used. In general, if the language contains basic communication and synchronisation instructions, communication structures can be implemented relatively easily, following the single element approach. However, if it is possible to reuse the design in more than one application, it would be convenient to consider a more flexible approach using configurable communication sub-systems for the exchange of different types and sizes of data. Design patterns can help to support to the implementation of these structures; especially, consider the *Composite Messages* pattern [SC95], the *Service Configurator* pattern [JS96], and the *Visibility and Communication between Control Modules* and *Client/Server/Service* patterns [AEM95, ABM96]

- *Agglomeration*. The data division and communication structure defined previously are evaluated with respect to the performance requirements. If necessary, the size of data pieces is changed, modifying the granularity of the system. Data pieces are combined or divided into larger or smaller pieces to improve performance or to reduce communication costs. Due to inherent characteristics of this pattern, the process is automatically balanced among the worker components, but granularity is modified in order to balance the process between the manager and the workers. If the operations performed by the workers are simple enough and workers receive relatively small amount of data, workers may remain idle while the manager is busy trying to serve their requests. On the contrary, if worker operations are too complex, the manager will have to keep a buffer of pending data to be processed. It is noticeable that load-balance between manager and workers can be achieved simply by modifying the granularity of data division.
- *Mapping*. In the best case, the hardware allows that each component is assigned to a processor with enough communication links to perform efficiently. However, generally the number of components is defined a lot bigger than the number of available processors. In this case, it is common to place a similar number of worker components on each processor. To keep the structure the most balanced possible, the manager component can be executed on a dedicated processor, or at least on a processor with a reduced number of working components. The competing forces of maximising processor utilisation and minimising communication costs are almost totally fulfilled by this pattern. Mapping can be specified statically or determined at run-time, allowing a better load-balance. As a "rule of thumb", parallel systems based on the *Manager-Workers* pattern will perform reasonably well on a MIMD (multiple-instruction, multiple-data) computer, and it may be difficult to adapt to a SIMD (single-instruction, multiple-data) computer [Pan96].

Consequences

Benefits

- The order and integrity of data is preserved and granted due to the defined behaviour of the manager component. The manager takes care of what part of the data has been operated on, and what remains to be computed by the workers
- An important characteristic of the *Manager-Workers* pattern is due to the independent nature of operations that each worker performs. Each worker requests for a different piece of data during execution, which makes the structure to present a natural load balance [POSA96, CT92].
- As every worker component performs the same computation, granularity can be modified easily, due to it depends only on the size of the pieces in which the manager divides the data. Furthermore, it is possible to exchange worker components or add new ones without significant changes to the manager, if an abstract description of the worker is provided [POSA96].
- 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.
- Using the *Manager-Worker* pattern, the parallelising task is relatively straightforward, and it is possible to achieve a respectable performance if the application fits this pattern. If designed carefully, the *Manager-Worker* pattern enables the performance to be increased without significant changes [POSA96, Pan96].

Liabilities

- The *Manager-Workers* pattern presents execution problems if the number workers is large, the operations performed by the workers are too simple, or workers receive small amounts of data. In all cases, workers may remain idle while the manager is busy trying to serve all their requests. Granularity should be modified in order to balance the amount of data among the workers.

- Manager-Worker architectures are not always feasible when performance is a critical issue. If the manager activities - data partition, receive worker requests, send data, receive partial results, and computing the final result - may take a longer time compared with the processing time of the workers, it can be considered that the overall performance depends mostly on the manager performance. A poor performance of the manager impacts heavily on the performance of the whole system [POSA96, CT92].
- Many different parameters must be carefully considered, such as strategies for work subdivision, manager and worker collaboration, and computation of final result. Also, 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 [POSA96].

Known uses

- Connectivity and Bridge Algorithms are an application of the *Manager-Workers* pattern. The problem is to determine if a connected graph has a bridge. A bridge is an edge whose removal disconnects the graph. A simple algorithm attempts to verify if an edge is a bridge by removing it and testing the connectivity of the graph. However, the required computation is very dense if the number of edges in the graph is large. A parallel version using a *Manager-Worker* pattern approach is described as follows: each worker, using the algorithm proposed, is responsible for verifying if an edge is a bridge. Different workers check for different edges. The manager distributes the graph information to the workers, builds the final solution, and produces results [NHST94].
- In matrix multiplication, the matrixes are distributed among the workers by the manager. Each worker calculates products and returns the result to the manager. Finally, with all the results available, the manager can build the final result matrix [POSA96, Fos94].
- In image processing, the *Manager-Worker* pattern is used for transformations on an image that involve an operation on each piece of the image independently. For example, in computing discrete cosine transform (DCT), the manager divides the image in sub-images, and distributes them among the workers. Each separate worker obtains the DCT of its sub-images or pixel block and returns the result to the manager. The final image is then composed by the manager, using all the partial results provided by the workers [POSA96, Fos94].

Related patterns

The *Manager-Workers* pattern can be considered as a variant of the *Master-Slave* pattern [POSA96, PLoP94] for parallel systems. Other related patterns with similar approaches are the *Object Group* pattern [Maf96], and the *Client/Server/Service* pattern [PLoP94].

Shared Resource

The *Shared Resource* pattern is a specialisation of the *Blackboard* pattern [POSA96], lacking a control component and introducing aspects of activity parallelism, in which computations are performed, without a prescribed order, on ordered data. Commonly, components perform different computations on different data pieces simultaneously.

Examples

A Non-programming Example

Imagine a project of construction in general. An artifact (a building, a software program, a business program, etc.) is the result of the cooperation among several persons or teams, with different skills. Each one has different interests, responsibilities and perceptions of the project. All people or teams participate independently and simultaneously with different actions to achieve the result. A common objective and unity among them will produce a satisfactory result in time and consistent form.

A Programming Example

Consider a printed circuit board router program [Chien93]. A route is a series of wires or nets from one pin to another on a circuit board. Complex circuit boards might have a large number of routes. Routes must avoid obstacles on the board and other routes. A parallel implementation proposes to process each route by independent components, searching simultaneously on a data structure (representing the grid of coordinates on the board) for the shortest route between pins. Each component accesses the data structure, proposing a new probable free coordinate to add to its route from a given start point on the board. If this is occupied by an obstacle or another route, the component makes another proposal. Once a coordinate is added, each component calculates the lowest total distance estimate for its route.

Problem

It is necessary to apply completely independent computations as sets of non-deterministic interactions on elements of some centralised, perhaps not even ordered, data structure. Consider for example, a knowledge exchange problem. Components communicate via a shared resource, each one indicating its interest in a certain data object. The shared resource provides such data immediately if no other component is accessing it. Data consistency and preservation are tasks of the shared resource. The internal representation or order of data is important, but the order of operations on it is not a central issue. Generally, performance as execution time is the feature of interest.

Forces

Taking the previous problem description and other elements of parallel design such as granularity and load balance [Fos94, CT92], the following forces can be found:

- Preserve the order or integrity of data structure.
- Each processing element performs simultaneously a different and independent computation on different pieces of data.
- No specific order of data access by processing elements is defined.
- Improvement in performance is achieved when execution time decreases.

Solution

Each component executes simultaneously, capable of performing different and independent operations, accessing the data as shared resource when needed. Parallelism is absolute among components: any component can be performing different operations on a different piece data at the same time, without any prescribed order. The restriction is that no piece of information is accessed at the same time by different components. Communication can be achieved as function calls to require a service from the shared resource. The *Shared Resource* pattern can be considered as an activity parallel variation of the *Blackboard* pattern [POSA96] without a control instance that triggers the execution of sources. An important feature is that the execution does not follow a precise order of computations [Shaw95, Pan96].

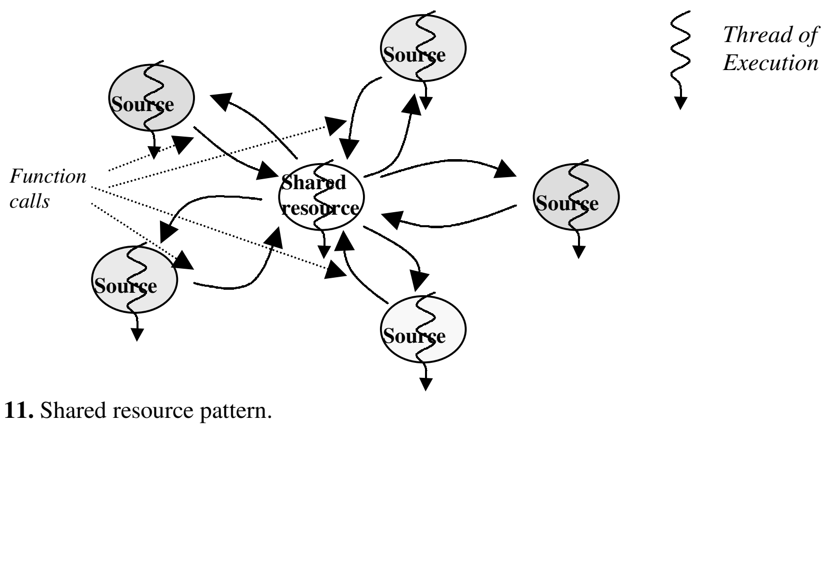

Figure 11. Shared resource pattern.

Structure

This pattern is based only on a shared resource as a central structure that controls the access of different sources. Usually, a shared resource component and several different source components simultaneously exist and process during the execution time (Figure 11).

Participants

- **Shared resource**. The responsibility of a shared resource is to coordinate access of sources, preserving the integrity of data.
- **Source**. The responsibilities of a source are to perform its independent computation, until requiring a service from the shared resource. Then, the source has to cope with any access restriction imposed by the shared resource.

Dynamics

A typical scenario to describe the basic run-time behaviour of the *Shared Resource* pattern is presented. All participants are simultaneously active. Every source performs a different operation, requiring the shared resource for operations. As soon as it finishes processing, returns to the calling source to continue its computations. Communications between sources are not allowed. The shared resource is the only common component among the sources (figure 12).

The functionality of this general scenario is explained as follows:

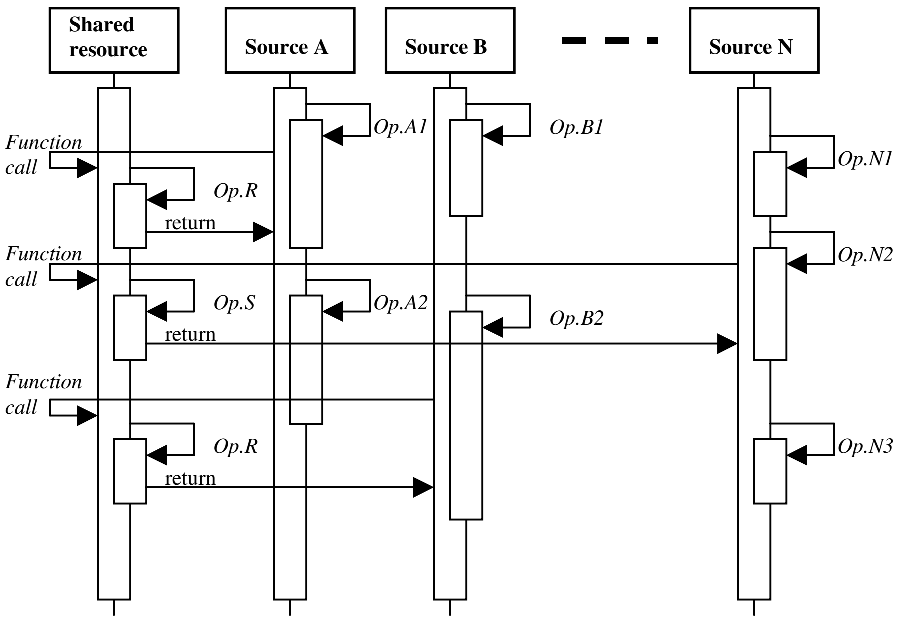

Figure 12. Scenario of Shared resource pattern.

- The **Shared resource** component is able to perform a couple of action, *Op.R* and *Op.S*. Each source starts processing, and calls the **Shared resource** to perform one of its operations. Sources perform different, independent operations.
- **Source A** performs *Op.A1*, requiring the **Shared resource** to perform operation *Op.R*. At the same time, **Source B** and **Source N** start performing *Op.B1* and *Op.N1*. The **Shared resource** returns a value to **Source A**.
- **Source N** starts performing *Op.N2*, requiring now the **Shared resource** to perform *Op.S*. Meanwhile, **Source A** processes *Op.A2*, and **Source B** processes *Op.B2*. After some time, operation *Op.B2* requires

the **Shared resource** to perform again *Op.R*. This returns a value to **Source N** and continues with the call from **Source B**.
- All operations continue in time, until all independent computations in the sources finish.

Implementation

The implementation process is based on the four stages mentioned above in the *Context and Implementation in general* section.

- *Partitioning*. The computation to be performed can be viewed as the effect of different independent computations on the data structure. Each source component is defined to perform a complete computation on the shared resource. Sources can be executed simultaneously due to their independent processing nature. However, the shared resource implementation should reflect a division and integrity criteria of the data structure, following the basic assumption that no piece of data is operated at the same time by two different sources. Therefore, sources may be implemented by a single entity (for instance, a process, a task, an object, etc.) that performs a defined computation, or a sub-system of entities. Design patterns in general [GHJV95, POSA96, PLoP94, PLoP95] may help with the implementation of the sources components as sub-systems of entities. Also, patterns used in concurrent programming like the *Object group* pattern [Maf96], the *Active Object* pattern [LS95], and *Categorize Objects for Concurrency* pattern [AEM95] can help to define and implement sources. Other design patterns like the *Double-Checked Locking* pattern [SH96], the *Thread-Specific Storage* pattern [HS97] and those presented in [McKe95] deal with issues about the safe use of threads and locks, and may provide help to implement the expected behaviour of the shared resource component.
- *Communication*. The communication to coordinate interaction of sources and shared resource is represented by an appropriate communication interface that allows the access to the shared resource. This interface should consider the form in which requests are issued to the shared resource, and the format and size of the data as argument or return value. In general, an asynchronous coordination schema is used, due to the heterogeneous behaviour of the sources. The implementation of a flexible interface between sources and shared resource can be done using design patterns for communication, like the *Service Configurator* pattern [JS96], the *Composite Messages* pattern [SC95], and the *Compatible Heterogeneous Agents* and *Visibility and Communication between Agents* patterns [ABM96].
- *Agglomeration*. The components and communication structures defined in the first two stages of a design are evaluated, comparing with the performance requirements. If necessary, operations can be recombined and reassigned to create different sets of source components with different granularity and load-balance. Usually, due to the independent nature of the sources, it is difficult to achieve a good performance initially, but at the same time, it is easy to perform changes on the sources without affecting the whole structure. The conjecture-test approach is used intensively, modifying both granularity and load-balance of source components to observe which combination can be used to improve performance. However, especial care should be taken with the load-balance between sources and shared resource. The operations of the shared resource should be lighter than any source computation, to allow a fast response of the shared resource to requests. Most of the computation activity is suppose to be performed by the sources.
- *Mapping*. In the best case, trying to maximise processor utilisation and minimise communication costs, each component should be assigned to a different processor. Due to the number of components is usually expected to be not too large, enough parallel processor can be commonly available. Also, the independent nature of sources allows that each source component can be executed on a different processor. The shared resource also is expected to be executed on a single processor, and all sources should have communication access to it. However, if the number of processors is limited and less than the number of components, it is difficult and complex to load-balance of the whole structure. To solve this, mapping can be determined at run-time by load-balancing algorithms. As a “rule of thumb”, systems based on the *Shared resource* patterns present a good performance when implemented on a MIMD (multiple-instruction, multiple-data) computer. Also, it would be very difficult to implement them for a SIMD (single-instruction, multiple-data) computer [Pan96, Pfis95].

Consequences

Benefits

- Integrity of the Shared resource data structure is preserved by the restriction that no data piece is accessed at the same time by different components.
- From the perspective of a parallel designer, this pattern is the simplest to develop due to the minimal dependence between components. Fundamentally, the operations on each data element are completely independent. That is, each piece of data can be computed on different machines, running independently as long as the appropriate input data are available to each one. It is relatively easy to achieve a significant performance in an application that fits this pattern [Pan96].
- As its components (the shared resource and the component sources) are strictly separated, *the Shared resource* pattern supports changeability and maintainability [POSA96, Pan96].
- The *Shared resource* pattern supports several levels of granularity. If required, the shared resource component can provide operations for different data sizes.
- Due to source components perform different and independent operations, they can be reused in different structures. The only requirement for reuse is that the source to be reused is able to perform certain operations on the data type in the new shared resource [POSA96, Pan96].
- A shared resource can provide fault tolerance for noise in data [POSA96, SG96].
- The activities of synchronisation are defined and localised generally in the shared resource components, and restricted between the shared resource and each one of the source components.

Liabilities

- Due to the different nature of each component, load balance is difficult to achieve, even when executing each component on a different processor. The difficulty increases if several components run together in a processor [Pan96].
- Results in a shared resource application are difficult to reproduce. Inherently, computations are not ordered following a deterministic algorithm and its results are not reproducible [POSA96]. Furthermore, the parallelism of its components introduces a non-deterministic feature to the execution [Pan96].
- Most shared resource structures require a great development effort, due to the variety of requirements in different problem domains [POSA96].
- Even when parallelism is straightforward, often the shared resource does not consider the use of control strategies to exploit the parallelism of sources and to synchronise their actions. Due to this, in order to preserve its integrity, the design of the shared resource component must consider extra mechanisms or synchronisation constraints to access its data. An alternative is perhaps to use the *Blackboard* pattern [POSA96].

Known uses

- Mobile robotics control is an application example of the *Shared resource* pattern. The software functions for a mobile robotics system has to deal with external sensors for acquiring input and actuators for controlling its motion and planning its future path in real-time. Unpredictable events may demand a rapid response: imperfect sensor input, power failures, mechanical limitations in the motion, etc. A solution example, the CODGER system, uses the *Shared resource* pattern to model the cooperation of tasks for coordination and resolution of uncertain situations in a flexible form. CODGER is composed of a "captain", a "map navigator", a "lookout", a "pilot" and a perception system, each one sharing information by a common shared resource [SG96].
- A real-time scheduler is another application of the *Shared resource* pattern. The application is a process control system, in which a number of independent processes are executed, each having its own real time requirements, and therefore, no process can make assumptions about the relative speed of other processes. Conceptually, they are regarded as different concurrent processes coordinated by a real-time scheduler, accessing, for instance, computer resources (Consoles, printers, I/O devices, etc.) which are shared among them. The real-time scheduler is implemented as a shared resource component to give processes exclusive access to a computer resource, but does not perform any operation on the resource itself. Each different

process performs its operations, requiring from time to time the use of the computer resources. The shared resource component grants the use of the resources, maintaining the integrity of the data read from or written to a resource by each different process [Han77].

- A Tuple space, used to contain data, presents a *Shared resource* pattern structure. Sources can generate asynchronous requests to read, remove and add tuples. The tuple space is encapsulated in a single shared resource component that maintains the set of tuples, preventing two parallel sources from act simultaneously on the same tuple [Fos94].

Related patterns

The *Shared Resource* pattern is considered a specialisation of the *Blackboard* pattern [POSA96] without control component, and introducing aspects of activity parallelism Also, it is related to the *Repository* architectural style [Shaw95, SG96]. Other patterns that can be considered related to this pattern are *the Compatible Heterogeneous Agents* pattern [ABM96] and the *Object Group* pattern [Maf96].

6. Summary

The goal of the present work is to provide software designers and engineers with an overview of the common structures used for parallel software systems, and provide a guidelines on the selection of architectural patterns during the initial design stages of parallel software applications. However, as a first attempt at the creation of a more organised pattern system for parallel programming it is not complete or detailed enough to consider every issue of parallel programming. The patterns described here can be linked with other current pattern developments for concurrent, parallel and distributed systems. Work on patterns that support the design and implementation of such systems has been addressed previously by several authors [Sch95, Sch98a, Sch98b].

7. Acknowledgements

We are especially grateful to our shepherd for EuroPLoP'98, Frank Buschmann, for his comments, suggestions and criticisms, providing invaluable insights, promoting discussion and profound reflections about the contents of this work and its improvement. We are also grateful to Dirk Riehle for his timely intervention and precise observations. Thanks also to all those who attended the *Patterns for Technical Domains* Workshop, at EuroPLoP’98, for their suggestions of improvements of this work. The work presented here is part of an ongoing PhD research by Jorge L. Ortega Arjona, supervised by Graham Roberts, in the Department of Computer Science, University College London.

8.References

[ABM96] Amund Aarsten, David Brugali and Giuseppe Menga. *Patterns for Cooperation*. Pattern Languages of Programming Conference (PLoP’96). Allerton Park, Illinois, USA. September 1996.

[AEM95] Amund Aarsten, Gabriele Elia and Guiseppe Menga. *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing.* In *Patterns Languages of Programming* (PLOP’94). Addison-Wesley, 1995.

[AS96] Rakesh Agrawal and John C. Shafer. *Parallel Mining of Association Rules: Design, Implementation and Experience.* IBM Research Report RJ 10004. IBM Research Division, Almaden Research Center. San Jose, CA, 1996.

[CG88] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs. A Guide to the Perplexed.* Yale University, Department of Computer Science, New Heaven, Connecticut. May 1988.

[Chien93] Andrew A. Chien. *Supporting Modularity in Highly-Parallel Programs.* In Research Directions in Concurrent Object-Oriented Programming, Gul Agha, Peter Wegner and Akinori Yonezawa (eds.). The MIT press, 1993.

[CMP95] Stephen Crane, Jeff Magee and Nat Pryce. *Design Patterns for Binding in Distributed Systems.* OOPSLA’95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. October 1995.

[CSG97] David Culler, Jaswinder Pal Singh and Anoop Gupta. *Parallel Computer Architecture. A Hardware/Software Approach (Preliminary draft).* Morgan Kaufmann Publishers, 1997

[CT92] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett Publishers, Inc., Boston, 1992.

[Fos94] Ian Foster. *Designing and Building Parallel Programs, Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Publishing Company, 1994.

[Fro96] Svend Frolund. *Coordinating Distributed Objects. An Actor-based Approach to Synchronization.* The MIT Press, Cambridge, Massachusetts 1996.

[Gab96] Richard Gabriel. *Patterns of Software: Tales from the Software Community.* Oxford University Press, 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Systems.* Addison-Wesley, Reading, MA, 1994.

[GSVOM97] D.F. Garcia Nocetti, J. Solano Gonzalez, M.F. Valdivieso Casique, R. Ortiz Ramirez and E. Moreno Hernandez. *Parallel Processing in Real-Time Ultrasonic Imaging.* 4th IFAC Workshop on Algorithms and Architectures for Real-Time Control, AARTC'97. Vilmoura, Portugal. April 1997.

[Han77] Per Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice-Hall Series in Automatic Computation, Englewood Cliffs, New Jersey, 1977.

[Hoare84] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[HPCN98] High Performance Computing and Networking Technology Transfer Nodes. *Film, entertainment and video page.* ESPRIT project. http://www.hpcn-ttn.org/themegroupsswitch.html. 1998.

[HTD93] Waldemar Horwat, Brian Totty and William J. Dally. *COSMOS: An Operating System for a Fine-Grain Concurrent Computer.* In Research Directions in Concurrent Object-Oriented Programming, Gul Agha, Peter Wegner and Akinori Yonezawa (eds.). The MIT press, 1993.

[HS97] Tim Harrison and Douglas C. Schmidt. *Thread-Specific Storage. A Behavioral Pattern for Efficiently Accessing per-Thread State.* Second annual European Pattern Languages of Programming Conference (EuroPLoP'97). Kloster Irsee, Germany. July 1997.

[JS96] Prashant Jain and Douglas C. Schmidt. *Service Configurator. A Pattern for DynamicConfiguration and Reconfiguration of Communication Services.* Third Annual Pattern Languages of Programming Conference, Allerton Park, Illinois. September 1996.

[JP96] Jean-Marc Jezequel and Jean-Lin Pacherie. *The "Ubiquitous Agent" Design Patterns.* Pattern Languages of Programming Conference (PLoP'96). Allerton Park, Illinois, USA, 1996.

[KMJ96] Elizabeth A. Kendall, Margaret T. Malkoun and C. Harvey Jiang. *The Layered Agent Pattern Language.* . Third Annual Pattern Languages of Programming Conference, Allerton Park, Illinois. September 1996.

[LS95] R. Greg Lavender and Douglas C. Schmidt. *Active Object. An Object Behavioral Pattern for Concurrent Programming* .In *Patterns Languages of Programming 2* (PLOP'95). Addison-Wesley, 1996.

[Maf96] Silvano Maffeis. *Object Group Design Pattern.* Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS). Toronto, Ontario, Canada, 1996.

[McKe95] Paul E. McKenney. *Selecting Locking Primitives for Parallel Programs.* In *Patterns Languages of Programming 2* (PLoP'95). Addison-Wesley, 1996.

[NHST94] Christopher H. Nevison, Daniel C. Hyde, G. Michael Schneider, Paul T. Tymann. *Laboratories for Parallel Computing.* Jones and Bartlett Publishers, 1994.

[Pan96] Cherri M. Pancake. *Is Parallelism for You?* Oregon State University. Originally published in Computational Science and Engineering, Vol. 3, No. 2. Summer, 1996.

[Para98] David A. Bader. *Parascope. A listing of Parallel Computing Sites.* http://computer.org/parascope/index.html . August 1998.

[PB90] Cherri M. Pancake and Donna Bergmark. *Do Parallel Languages Respond to the Needs of Scientific Programmers?* Computer magazine, IEEE Computer Society. December 1990.

[Perr92] R.H. Perrot. *Parallel language developments in Europe: an overview.* In Concurrency: Practice and Experience, Vol. 4(8). John Wiley & Sons, Ltd. December 1992.

[Pfis95] Gregory F. Pfister. *In Search of Clusters. The Coming Battle in Lowly Parallel Computing.* Prentice Hall Inc. 1995.

[Phil95] Michael Philippsen. *Imperative Concurrent Object Oriented Languages.* Technical report TR-95-050. International Computer Science Institute. Berkeley, California. August 1995.

[PLoP94] James O. Coplien and Douglas C. Schmidt (editors). *Patterns Languages of Programming.* Addison-Wesley, 1995.

[PLoP95] James O. Coplien, Norman L. Kerth and John M. Vlissides (editors). *Patterns Languages of Programming 2.* Addison-Wesley, 1996.

[POSA96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal. *Pattern-Oriented Software Architecture.* John Wiley & Sons, Ltd., 1996.

[SC95] Aamond Sane and Roy Campbell. *Composite Messages: A Structural Pattern for Communication Between Components.* OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. October 1995.

[Sch95] Douglas Schmidt. *Accepted Patterns Papers.* OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. http://www.cs.wustl.edu/~schmidt/OOPSLA-95/html/papers.html. October, 1995.

[Sch98a] Douglas Schmidt. *Design Patterns for Concurrent, Parallel and Distributed Systems.* http://www.cs.wustl.edu/~schmidt/patterns-ace.html. January, 1998.

[Sch98b] Douglas Schmidt. *Other Pattern URL's. Information on Concurrent, Parallel and Distributed Patterns.* http://www.cs.wustl.edu/~schmidt/patterns-info.html. January, 1998.

[SH96] Douglas Schmidt and Tim Harrison. *Double-Checked Locking. An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently.* Third annual Pattern Languages of Programming Conference. Allerton Park, Illinois. September 1996.

[Shaw95] Mary Shaw. *Patterns for Software Architectures.* Carnegie Mellon University. In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall Publishing, 1996.

[ST96] David B. Skillicorn and Domenico Talia. *Models and Languages for Parallel Computation.* Computing and Information Science, Queen's University and Universita della Calabria. October 1996.

[VBT95] Allan Vermeulen, Gabe Beged-Dov and Patrick Thompson. *The Pipeline Design Pattern.* OOPSLA'95, Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems. October 1995.

[Watt93] Alan Watt. *3D Computer Graphics.* Second Edition, Addison-Wesley, 1993.