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.
April, 1998.

Abstract

This paper introduces an approach for describing and selecting architectural patterns for parallel programming, according to the requirements of order of data and computations of the problem and the nature of their processing components, using pattern descriptions.

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 [2,3,8,9].

The power of parallelism relies on issues concerning with partitioning a big problem in order to deal with complexity. The partitioning is necessary to divide a big problem into smaller sub-problems that are possible to understand, and perhaps work on them separately, at a more comfortable level. 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.

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. It is considered that the selection of an architectural pattern is a fundamental decision during the design of the overall structure of a software system [1,6].

The initial attraction to architectural patterns is the promise of minimising the effects of imminent requirements changes 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 [6].

Patterns share a formal structure. At every level of scale (architectural patterns, design patterns and idioms) they regularly contain the same elements: a name, a short summary, a context, a problem statement (that includes a description of its forces), a solution statement (which covers descriptions of its structure, participants, basic dynamics and implementation steps), known uses and consequences (describing benefits and liabilities of the pattern) [1,4,5]. These elements provide a uniform template for writing and browsing pattern descriptions contained in a pattern system, 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 [9], Foster [3], and Carriero and Gelernter [2] 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 with 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* [3], *domain parallelism* [3], and *activity parallelism* [2], 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 [2,3,9].

- 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 [2,3,9].

- Activity parallelism is related with 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 null. Communication is required between a component that controls the access of components to the data structure and the processing components [2,9].

The nature of processing components is another classification element that can be used for parallel systems. Generally, components of a 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 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 switched with other without noticeable change in the operation of the system. Usually, homogeneous systems have a large number of components, which communicate using operations of data exchange.

- 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 switched with another. In general, heterogeneous systems are composed by fewer components than homogeneous systems, communicating with function calls.

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

- The *Pipes and Filters* pattern is an extension of the original Pipes and Filters pattern [1,6,7] 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 *Layers* pattern extends the original approach of the Layers Pattern [1,6,7] 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 components of a layer 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 a domain parallelism. The same operations are performed simultaneously on different pieces of ordered data [3,11]. 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 [1] for parallel systems, introducing an activity parallelism where the same operations are performed on ordered data. Each components 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 *Repository* pattern is a specialisation variant of the Blackboard pattern [1] introducing activity parallelism characteristics, where different computations are performed, without a prescribed order, on ordered data. Components perform different computations simultaneously.

	Functional Parallelism	Domain Parallelism	Activity Parallelism
Heterogeneous processing	*Pipes and Filters*		*Repository*
Homogeneous processing	*Layers*	*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 expected to be guided mainly by the properties of order of data and computations and the nature of its processing components of the problem. However, it is important to state that a particular architectural pattern, or its combination with others, is not still a complete parallel software application. Its objective is just to provide a stable structure for a software system, as a first step on the design and implementation of a parallel software architecture.

Based on the classification schema and the pattern descriptions [1,5], 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 precisely the problem in terms of its forces - order of data and computations, the nature of its processing components, requirements of performance, etc. - is the first step to identify an architectural pattern that helps to solve a concrete design problem.

2. *Select the category of parallelism*. In accordance with the requirements of order of data and operations of the design problem, select the category of parallelism - functional, domain or activity parallelism - that better describes and matches the design problem specification.

3. *Select the category of nature of processing components*. Select the homogeneous or heterogeneous nature of process distribution among components that matches the design problem specification. The nature of process distribution reflects indirectly characteristics about the number of processing components and the amount and kind of communications between them that can be used later in the solution.

4. *Compare the specification of the design problem with the Problem description*. The categories of parallelism and nature of processing components can be simply used to guide the selection of an architectural pattern. However, they only address certain aspects of the Problem description, and in order to verify that the

Problem description and forces of the selected pattern cope with the design problem at hand, compare the specification of the design problem with the Problem section of the selected pattern. A more specific knowledge about the design problem to be solved is required. Unless some problems were encountered before, the architectural pattern selection is completed. The design of the parallel system is continued using the architectural pattern Solution description in conjunction with other patterns that help to go on with the design and implementation of a complete solution. On the contrary, if the architectural pattern selected does not satisfactorily address aspects of the design problem at hand, it is possible to try to select an alternative pattern of other categories.

5. *Select an alternative architectural pattern*. If the selected pattern is not appropriate for the design problem at hand, try to select another pattern that alternatively can be used. Other categories of parallelism and nature of processing components that introduce another pattern may provide a better approach when modified, specialised or combined with others, to address the design problem specification. Checking the Known Uses section of the pattern description may be helpful to look for an alternative pattern too. Once an alternative pattern is selected, return to the previous step to verify it.

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 design problem. It is possible to look at other more general pattern languages or systems to see if they contain a pattern that can be used. Or the alternative is trying to solve the design problem without applying patterns.

5. Architectural Patterns for Parallel Programming

5.1. Pipes and Filters

The *Pipes and Filters* pattern extends the Pipes and Filters pattern [1,6,7] 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.

Context

Design a software program for a parallel system, using a certain programming language for certain parallel hardware.

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 are 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 some other forces for parallel software design [3,11] 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.
- Different step-operations can process different pieces of data at the same time.
- Distribute process into similar amounts among all step-operation.
- Performance is considered usually as maximizing processor utilization and minimizing communication costs in execution time.

Solution

Parallelism is represented by the overlapping of operations through time. The operations produce data output that depend on preceding operations on its data input, as incrementally ordered steps over time. 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 [1,2,6,9].

Structure

This pattern is called Pipes and Filters since data is passed 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 1).

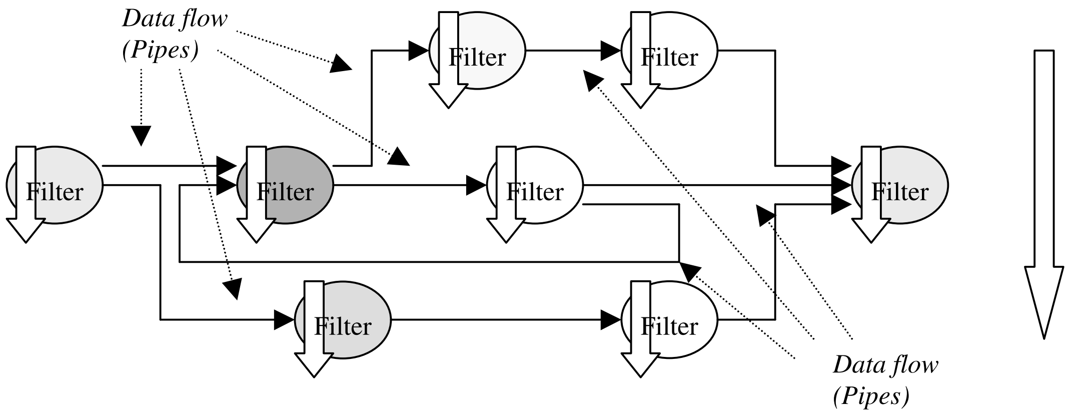

Figure 1. Pipes and filters.

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 synchronize 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 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. A pipes synchronise the activity between filters. This approach is based on the dynamic behaviour exposed by the Pipes and Filters pattern in [1].

In this scenario (figure2), 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**.

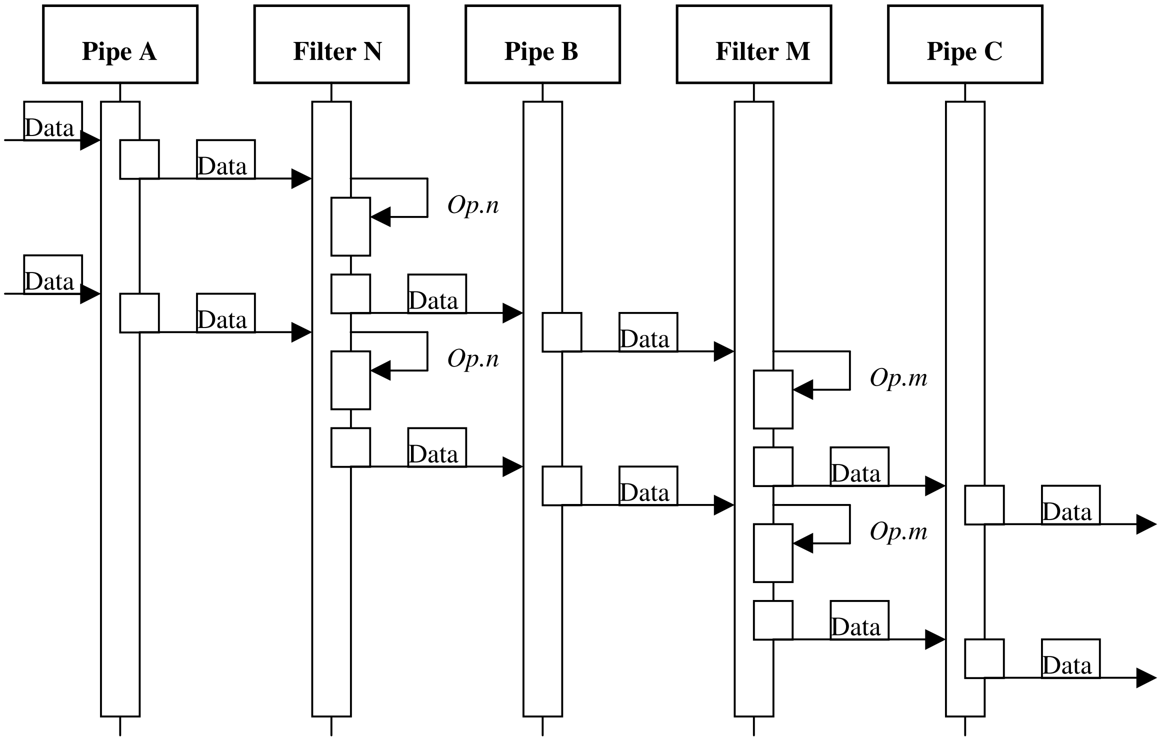

Figure 2. Scenario of Pipes and filters.

- **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.

Implementation

The implementation is intended to describe an exploratory approach to design, in which hardware-independent features are considered early, and hardware-specific issues are delayed in the implementation process. This method structures the implementation process based on four stages: *partitioning, communication, agglomeration* and *mapping*. 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 [1,3,13]:

1. *Partitioning*. The computation that is to be performed is decomposed, in accordance with the characteristic of order of operations, into a sequence of different operation stages, in which orderly data is received, operated on and passed to the next stage. Attention should be focused on recognizing opportunities for parallel execution among ordered operations, to be assigned and define filter components.

2. *Communication*. The communication required to coordinate the parallel execution 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 used in common Pipe and Filter systems.

3. *Agglomeration*. The filter and pipe structures defined in the previous stages are evaluated with respect to the performance requirements and implementation costs. Operations may be recombined, and filter components redefined to modify the characteristics of granularity and load-balance, aiming to balance the workload and to reduce communication costs.

4. *Mapping*. Each component is assigned to a processor, attempting to satisfy the competing forces of maximizing processor utilization and minimizing communication costs. Usually, mapping is specified as static for systems designed with this pattern.

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 [3].

- 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?" [2,10].

- 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 [3].

Consequences

Benefits

- The use of Pipes and Filters pattern allows for the description of the computation of a parallel system in terms of the composition of operations of its component filters. Every operation can be understood then in terms of input/output relations [7].

- 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 [7].

- Normally, 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 [1,7].

- Pipes and Filters structures permit certain specialised analysis methods relevant to parallel systems, such as throughput and deadlock analysis [7].

- The performance of Pipes and Filters architectures depends on the number of steps to be computed. Once all components are active, the processing efficiency is constant [1,10].

Liabilities

- The use of Pipes and filters introduces potential execution problems if they are not load balanced properly; this is 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 balance the work load, performance that could potentially be obtained from a Pipes and Filters system may not be worth the programming effort [9,10].

- 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 from medium to coarse. This is due to he 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 [7].

- 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 Layers, is advised [1].

5.2. Layers

The *Layers* pattern for parallel systems is an extension of the original Layers Pattern approach [1,6,7] 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. A components can be created statically, waiting for calls from higher layers, or dynamically, when a call triggers its creation.

Context

Design a software program for a parallel system, using a certain programming language for certain parallel hardware.

Problem

It is necessary to perform a computation repeatedly, that is 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 of abstraction. An operation at a high level of abstraction requires the evaluation of one or more operations at lower abstraction 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 [3,11], the following forces should be considered for a parallel version of the Layers pattern:

- Perform a complex computation - as a hierarchy of ordered operations - several times, in a single execution.
- Data is shared among layers.
- In each layer, an operation can be performed several times, simultaneously, on different pieces of data.
- Operations of different layers may be different in size and level of complexity.

- Dynamic creation and destruction of components is preferred over static, to achieve load balance.
- Performance is considered usually as maximizing processor utilization and minimizing communication costs in execution time.

Solution

Parallelism is introduced by overlapping different level operations through time in layers. When an operation triggers an operation at a higher level, this may also require the evaluation of other operations in lower levels. These operations are triggered by a function call, and may also need the evaluation of operations, as function calls, in even lower levels. Data can be shared, in the form of arguments for the function calls. During the evaluation of operations in each level, usually the higher levels need to wait for a result from lower levels. 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. More complex computations may require a larger number of abstraction levels, while simple computations may require a reduced number of levels. The time required to evaluate a computation depends on its complexity or level of abstraction [1,6].

Structure

This pattern is composed of conceptually-independent entities, ordered in the shape 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 3).

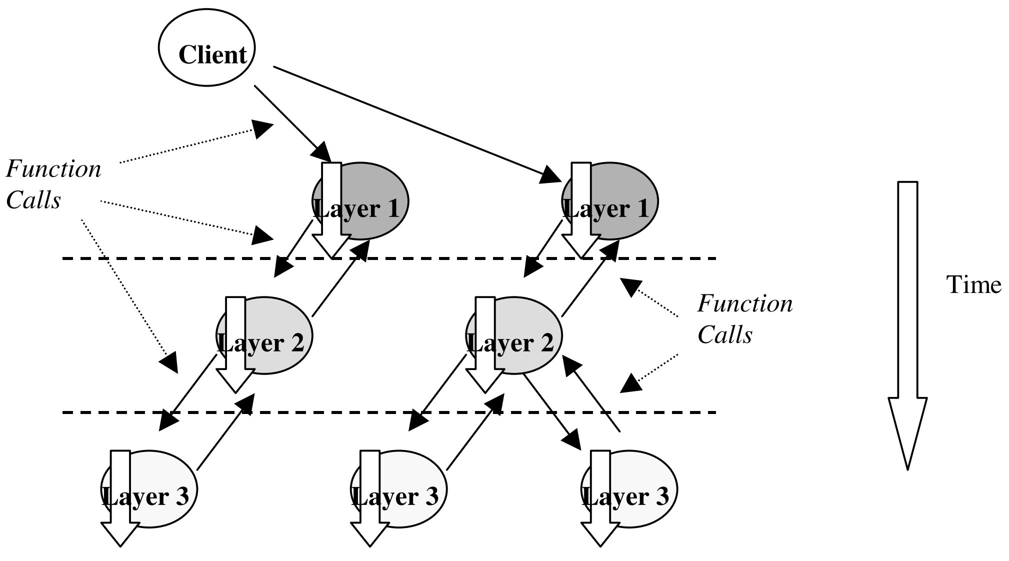

Figure 3. Layers.

Participants

- **Layer element**. The responsibilities of a layer component is to provide operations or functions to more abstract level layers, and to delegate more simple subtasks to layers in less abstract levels.

Dynamics

As the parallel execution of layer elements is allowed, a typical scenario is proposed to describe its 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 4):

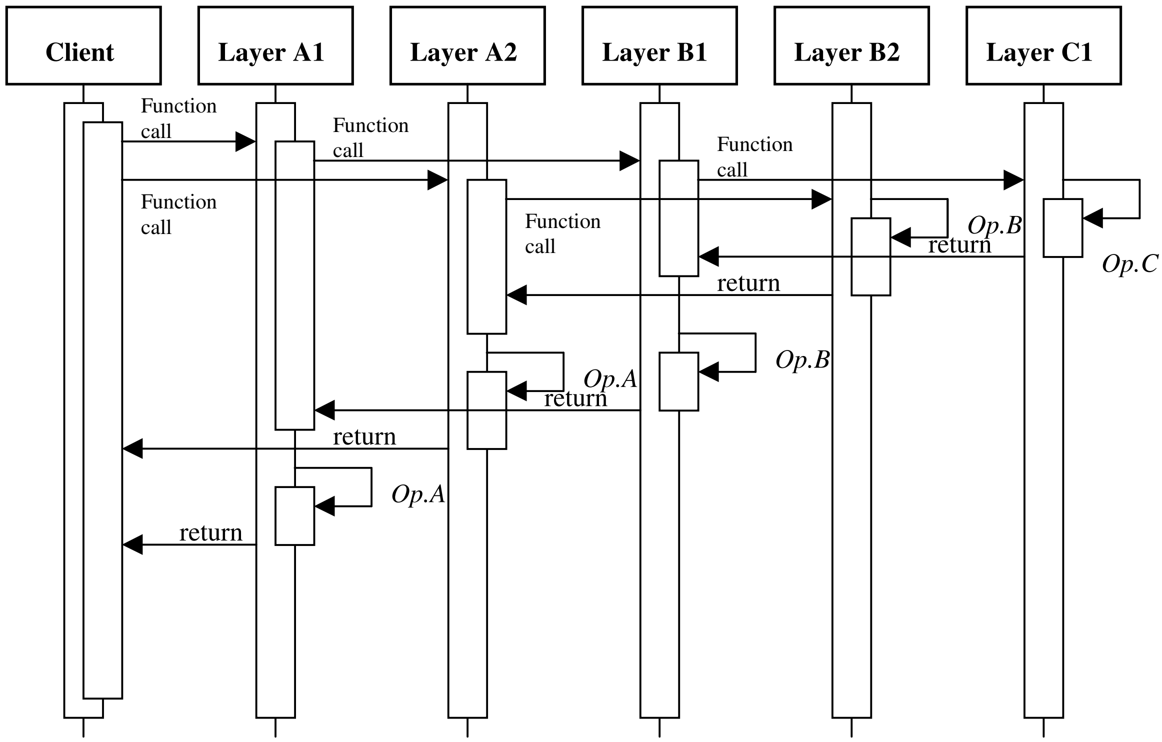

Figure 4. Scenario for Layers patterns.

- 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 sublayers. This is the same behaviour than the sequential version of the Layers pattern [1].

- 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 good enough to serve calls from components in Layer A. As 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.

Implementation

The implementation method is described as an exploratory approach to design, where hardware-independent features are considered early, and hardware-specific issues are delayed in the implementation process. Four stages - *partitioning, communication, agglomeration* and *mapping*- are considered. In the first two stages, attention is focused on concurrency and scalability characteristics. The third and fourth stages, attention is aimed to locality and other performance-related issues [1,3,13]:

1. *Partitioning*. The computation to be performed is decomposed into a set of operations, hierarchically defined and related in accordance with complexity of operations as abstraction criteria: Complex operations at higher levels are recursively decomposed to more simple operations at lower levels, determining the number of abstraction levels. Following this decomposition, the component representative of each layer can be defined. The number of instances of this component representative is driven by its operational complexity. Due to the hierarchy of operations is ordered by a complexity criteria, the cost to create a simple component at lower layers is commonly less than the cost of creating a complex component in higher layers. Therefore, the number of complex components in a higher layer is usually less than the number of more simple components in a lower layer. Following this general rule in hierarchies, the number of components per layer depends inversely on its complexity: more complex components should be less in number, and more simple components should be more in number.

2. *Communication*. The communication required to coordinate the parallel execution of layer components is determined. Communication structures and procedures should be considered to specify 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.

3. *Agglomeration*. The layer structure and components defined in the previous stages are evaluated with respect to the performance requirements and implementation costs. If necessary, layer components are refined by combination or decoupling of operations, modifying the granularity and load-balance of components to improve performance or to reduce development costs.

4. *Mapping*. Usually, all equal components of a layer can be executed concurrently on a processor, trying to satisfy the competing forces of maximizing processor utilization and minimizing communication costs. In general, mapping of layers to processors is specified static, allowing an internal dynamic creation of new components to serve new requests.

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 Layers pattern: individual computations and communications are executed by replacing functions at different levels of abstraction [8].

- 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 [3,10].

- The Gaussian elimination method, used to solve systems of linear equations, is a numerical problem that can be solved using a Layer structure. The original system of equations, expressed as a matrix, is reduced to a triangular form by performing a set of linear operations on the elements of each one of the rows as a layer. Once the triangular equivalent of the matrix is available, arithmetic operations must be performed by each layer to obtain the solution of each linear equation [3].

Consequences

Benefits

- The 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, more simple operations [7]. Allowing each layer to be presented as multiple components executing in parallel allows to perform the computation several times, enhacing 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. [1,7].

- 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 effort [1,7].

- 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 [9].

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 [7,9].

- 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 [1].

- 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 [1,7,10]

- If the levels of complexity of the layers is not correct, a problem 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 [1].

5.3. Communicating Sequential Elements

The *Communicating Sequential Elements* pattern is a domain parallelism pattern where each components performs the same operations on different pieces of ordered data [3,11]. 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.

Context

Design a software program for a parallel system, using a certain programming language for certain parallel hardware.

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 was 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 [3,11], the following forces should be considered:

- Preserve the precise order of data objects.
- Computations are performed on local pieces of data.
- Every element performs exactly the same operations in number and complexity.
- Partial results are usually shared among neighbour computations.
- Data is distributed into similar amounts among computations.
- Performance is considered usually as maximizing processor utilization and minimizing communication costs in execution time.

Solution

Parallelism is introduced as multiple participating concurrent components, each one applying a set of operations to 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. [2,6,9].

Structure

In this pattern, conceptually, the same operation is applied almost simultaneously 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 order imposed by this structure. Identical components simultaneously exist and process during the execution time (Figure 5).

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 synchronize communication activity between them.

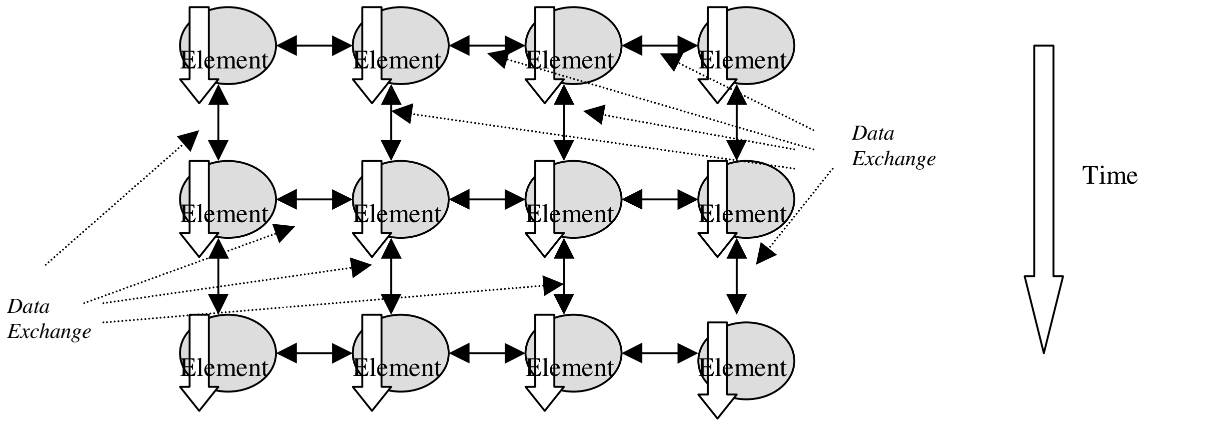

Figure 5. Communicating Sequential Elements.

Dynamics

A typical scenario to describe its run-time behaviour of this pattern, where all the Sequential Elements are active at the same time is presented.

Every Sequential Element performs the same operations, as a piece of a processing network. In the most simple case, each Sequential Element communicates only with a previous and next others (figure 6). The processing and communicating scenario is as follows:

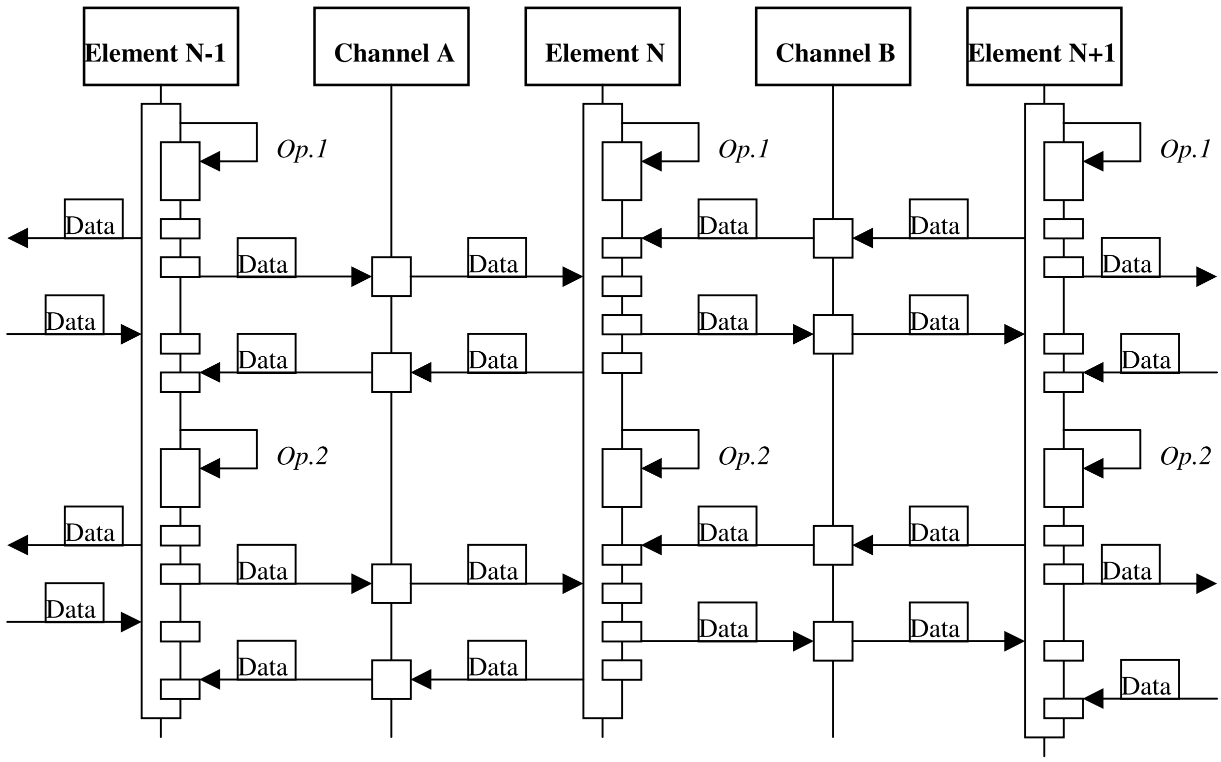

Figure 6. Scenario of Communicating Sequential Elements.

- 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 description is presented as an exploratory approach to design, considering hardware-independent features early and delaying hardware-specific issues in the implementation process. This implementation process is based on *partitioning, communication, agglomeration* and *mapping* stages. Attention is focused on concurrency and scalability characteristics during the first two stages, and locality and other performance-related issues in the last t two stages [3,13].

1. *Partitioning*. The ordered logical structure of data is a natural candidate to be decomposed into a network of data sub-structures or pieces. Depending on the precision required, data pieces can expose different size and shape. However, to maintain an equilibrium in the process, 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.

2. *Communication*. The communication is also associated with the network of data pieces. Each sequential element exchanges partial results with its neighbours through a channel. 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.

3. *Agglomeration.* The sequential elements and channels defined task are evaluated with respect to the performance requirements and implementation costs. 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.

4. *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 maximize processor utilization and minimize communication costs, is balance. In these kind of 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.

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 [10].

- 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 [3].

- 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 subimage. Edge pixels between sub-images are labelled in cooperation by the two respective processing elements [3].

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 [7,8].

- As all sequential elements share the same functional structure, their behaviour can be modified or changed without great effort [7,8].

- 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 [2,6,9].

- 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 [9].

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 [3,9].

- 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 [9].

- The synchronous characteristics of the application determines the 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 [9].

5.4. Manager-Workers

The *Manager-Workers* pattern can be considered as a variant of the Master-Slave pattern [1] 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 [11]. Each processing components performs the same operations, independent of the processing activity of other components. Different pieces of data are processed simultaneously. Preserve the order of data is the important feature.

Context

Design a software program for a parallel system, using a certain programming language for certain parallel hardware.

Problem

A computation is required where the application performs, perhaps repeatedly, independent computations as sets of transformations 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 an 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 [3,11], the following forces are found:

- Preserve the order of data.
- Each processing element performs the same computation on a different piece of data.
- The computation in each processing element is independent of the computations in other processing elements.
- No specific order of data distribution among processing elements is defined.
- Data pieces are distributed equally among processing elements.
- Each computation is considered to take the same amount of time.
- Performance is considered usually as maximizing processor utilization and minimizing communication costs in execution time.

Solution

Parallelism can be presented by having multiple data sets processed at the same time. The most flexible representation of this is the Manager-Workers approach. In this structure, a manager component initializes the computation, creating some number 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 [1,2,6,9,11].

Structure

This pattern is represented by a manager that preserves the order of data, controlling a group of workers as processing elements. 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 7).

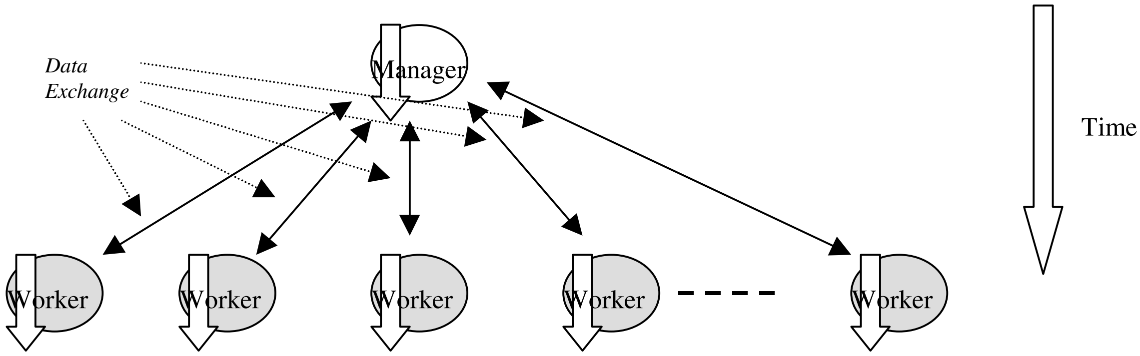

Figure 7. Manager-Workers.

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 its run-time behaviour of the Manager-Worker pattern, where all participants are simultaneously active, is presented. 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 8).

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

- All participants are created, and wait until a computation is required to the manager. Data is available to the manager, which divides it and sends equal 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, and requiring more data. If there is still data to be operated, the process repeats.

- The manager receives the partial results, and replies to requests of data from the workers. Once the data has been completely 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

Implementation

An exploratory approach to design, in which hardware-independent features are considered early and hardware-specific issues are delayed, is proposed for the implementation process. The implementation process is based on four stages. The first two stages, *partitioning* and *communication*, are focused on concurrency and scalability characteristics. The last two stages, *agglomeration* and *mapping*, are aimed to shift locality and other performance-related issues [1,3,13].

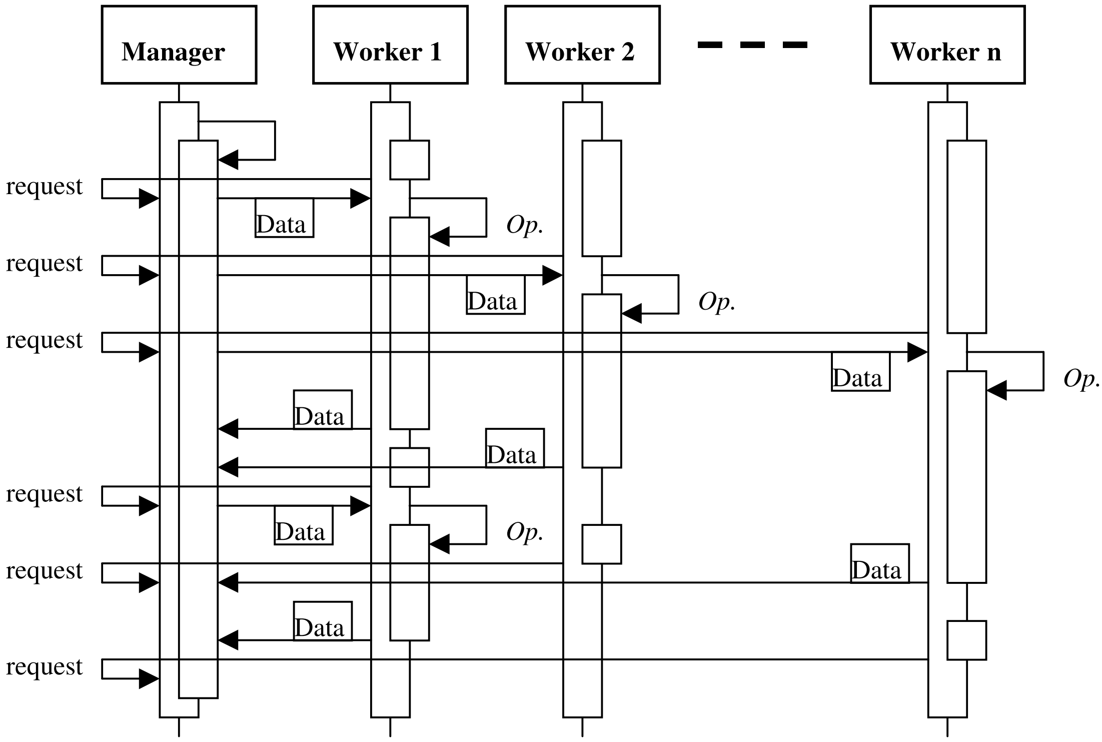

Figure 8. Scenario of Manager-Workers.

1. *Partitioning*. The data that is to be operated on by the computation is decomposed into a set of ordered pieces of similar size. This division criteria and the order of data are clear opportunities for parallel execution. They are used to define the divide and combination activity of the manager component. The independent operations to be performed as part of the computation are the same in each one of the worker components. Sometimes, the manager is implemented to perform some operation on the data as well. However, this is not advisable, to maintain the load balance of operations.

2. *Communication*. The communication required to coordinate the execution between the manager and worker. As workers are just allowed to communicate with the manager to get more work, defining an appropriate communication between manager and worker components is an important task. The communication structure should allow a worker to request a piece of data, combined with it reception and, once processed, its delivery to the manager. The manager should perform equivalent communication operations. Important parameter 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 more commonly used in Manager-Worker systems.

3. *Agglomeration*. The data division and communication structure defined are evaluated with respect to the performance requirements and implementation costs. If necessary, the size of data pieces is changed. Data pieces are combined or divided into larger or smaller pieces to improve performance or to reduce communication costs. In this pattern, granularity is modified in order to balance the amount of data among 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.

4. *Mapping*. In the best case, if the hardware allows it, each component is assigned to a processor that counts with the communication links to perform efficiently. However, often, the number of components is bigger than the number of processors. In this case, it is common to place the same number of worker components on each processor. Also, to keep the structure the most balanced possible, the manager component can be executed on a dedicated processor, or at least with a little number of working components. The competing

forces of maximizing processor utilization and minimizing communication costs are almost totally fulfilled by this pattern,. Mapping can be specified statically or determined at run-time, allowing a better load-balance.

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 in a connected graph 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 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 [10].

- In matrix multiplication, the component values of two matrixes are distributed by the manager among the workers. 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 [1,3].

- 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 from the workers [1,3].

Consequences

Benefits

- An important characteristic of the Manager-Workers 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 [1,11].

- 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.

- 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 parallelisation 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 [1,9].

- 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 [1].

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.

- A Manager-Worker architecture is 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 [1,11].

- 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 [1].

5.5. Repository

The *Repository* pattern is a specialization of the Blackboard pattern [1] 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.

Context

Design a software program for a parallel system, using a certain programming language for certain parallel hardware.

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 repository, each one indicating its interest in a certain data object. The repository provides such data immediately, or when some other component inserts it into the repository. Data consistency and preservation are tasks of the repository. The internal representation or order of data 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 [3,11], the following forces can be found:

- Preserve the order or integrity of data.
- Each processing element performs simultaneously different computation on a different pieces of data.
- The operations in each processing element is independent of the computations in the other processing elements.
- No specific order of data distribution among processing elements is defined.
- Data pieces can be distributed in different sizes among processing elements.
- Each computation can take different amount of time.
- Performance is considered usually as maximizing processor utilization and minimizing communication costs in execution time.

Solution

Parallelism is absolute between 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 repository. The Repository pattern can be considered as an activity parallel variation of the Blackboard pattern [1] without a control instance that triggers the execution of sources. Each source executes simultaneously, capable of performing different and independent operations, accessing the repository when needed. An important feature is that the execution does not follow a precise order of computations [6,9].

Structure

This pattern is based only on a repository as a central structure that controls the access of different sources. Usually, a repository component and several different source components simultaneously exist and process during the execution time (Figure 9).

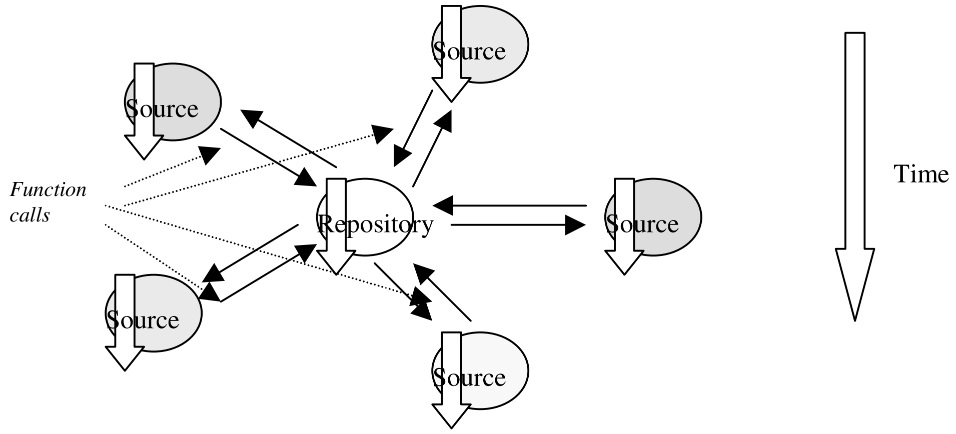

Figure 9. Repository.

Participants

- **Repository**. The responsibility of a repository 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 repository. Then, the source has to cope with any access restriction imposed by the repository.

Dynamics

A typical scenario to describe its run-time behaviour of the Repository pattern is presented. All participants are simultaneously active. Every source performs a different operation, requiring the Repository for operations. As soon as it finishes processing, returns to the calling source to continue its computations. Communications between sources are not allowed. The repository is the only common component among the sources (figure10).

The functionality of this general scenario is explained as follows:

- The **Repository** component is able to perform a couple of action, *Op.R* and *Op.S*. Each source starts processing, and calls the **Repository** to perform one of its operations. Sources perform different, independent operations.

- **Source A** performs *Op.A1*, requiring the **Repository** to perform operation *Op.R*. At the same time, **Source B** and **Source N** start performing *Op.B1* and *Op.N1*. The **Repository** returns a value to **Source A**.

- **Source N** starts performing *Op.N2*, requiring now the **Repository** to perform *Op.S*. Meanwhile, **Source A** processes *Op.A2*, and **Source B** processes *Op.B2*. After some time, operation *Op.B2* requires the **Repository** 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.

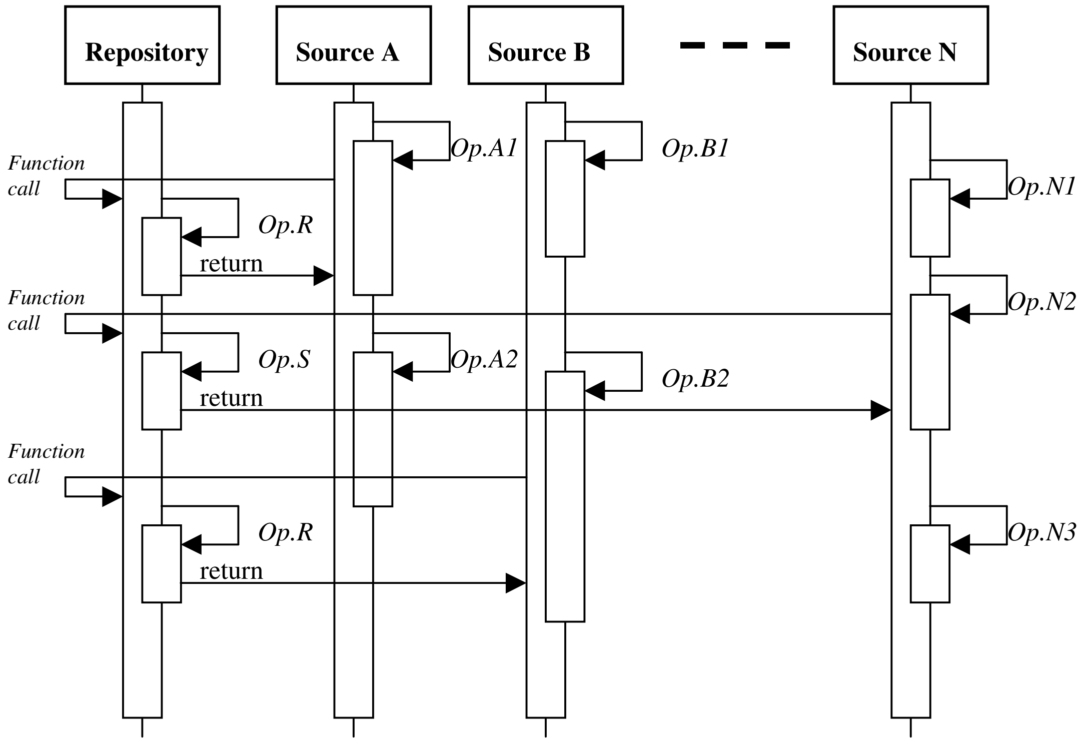

Figure 10. Scenario of Repository.

Implementation

The implementation process is intended to consider early hardware-independent features and delay hardware-specific issues, describing an exploratory approach to design. The implementation process is based on the *partitioning, communication, agglomeration* and *mapping* stages. The first two stages are developed, focusing on concurrency and scalability characteristics, and the last two stages, on locality and other performance-related issues [3,13].

1. *Partitioning*. The computation to be performed is decomposed into different simple operations. Each operation, or a group of them, are candidates to define the activity of the source components. Attention is focused on recognizing opportunities for the parallel execution of operations. On the other hand, the division and integrity criteria of the data are used to define the behaviour of the repository.

2. *Communication*. The communication required to coordinate the sources execution is determined, by an appropriate communication interface that controls the access to the repository. This should considerate the form in which request for data are passed to the repository, 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.

3. *Agglomeration*. The component and communication structures defined in the first two stages of a design are evaluated with respect to the performance requirements and implementation costs. If necessary, operations are combined and assigned to a single source component, modifying the granularity and load balance to improve performance or to reduce communication costs.

4. *Mapping*. Each component is assigned to a processor, if possible, trying to balance the competing forces of maximizing processor utilization and minimizing communication costs. As the number of components is usually not large, the number of parallel processor is commonly available. Due to their independent behaviour, each source component are executed on a different processor.The repository is also executed on a processor, that all sources have communication access to.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 structure. To solve this, mapping can be determined at run-time by load-balancing algorithms.

Known uses

- Mobile robotics control is an application example of the Repository 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 problems and failures, mechanical limitations in the motion, and so on. A solution example, the CODGER system, uses the Repository pattern to model the cooperation of tasks for coordination and resolution of uncertain situations in a flexible form. CODGER is composed by a "captain", a "map navigator", a "lookout", a "pilot" and a perception system. Each one shares information with the others using a common repository [7].

- A real-time scheduler is another application of the Repository 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 repository 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 resource. The repository component grants the use of the resource, maintaining the integrity of the data read from or written to the resource by each different process [12].

- A Tuple space, used to contain data, presents a Repository structure. Sources can generate asynchronous requests to read, remove and add tuples. The tuple space is encapsulated in a single repository component that maintains the set of tuples, preventing two parallel sources from act simultaneously on the same tuple [3].

Consequences

Benefits

- 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 [9].

- As its components (the repository and the component sources) are strictly separated, the Repository pattern supports changeability and maintainability [1,9].

- Repository supports several levels of granularity. If required, the repository component can provide operations for different data sizes.

- Due to source components perform different and independent operations, they can be reused in different repositories. 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 repository [1,9].

- A repository can provide fault tolerance for noise in data [1,7].

- The activities of synchronisation are defined and localised generally in the repository components, and restricted between the repository 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 [9].

- Results in a repository application are difficult to reproduce. Inherently, computations are not ordered following a deterministic algorithm and its results are not reproducible [1]. Furthermore, the parallelism among its components introduce a non-deterministic feature to the execution [9].

- Most repositories require a great development effort, due to the variety of requirements in different problem domains [1].

- Even when parallelism is straightforward, often the repository 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 repository component must consider extra mechanisms or synchronisation constraints to access its data [1].

6. Summary.

The goal of the present work is to help software engineers and designers to get an overview of the structure of parallel software systems. An architectural approach is presented as a guideline to the selection of architectural patterns during initial stages of the development of parallel applications. It can also be used during the analysis and design stages, taking information related to the problem from the analysis and providing a model to start the design of a software system. However, as a first stage in the creation of a pattern system for parallel programming, it cannot be considered as complete or detailed enough to consider every issue in parallel programming. We are trying to describe our patterns in a manner that they can be linked to current pattern developments for concurrent, parallel and distributed systems. Previous work about patterns that support the design and implementation of concurrent, parallel and distributed systems have been addressed by Douglas Schmidt, Douglas Lea, and others [14].

A problem has arisen when trying to fill the gap of a counterpart for the Communicating Sequential Elements pattern, that matches with the categories of domain parallelism and heterogeneous processing. This pattern is difficult to state because it should provide support for computation and communication issues for an overall organization structure that does not have a regular or precise shape. Examples of parallel applications that exhibit such kind of structure can be found in the area of simulation, in which the logical structure depends on the problem domain description. The structure of this pattern is considered to be irregular because the components are related to form a network without a shape. However, the creation and coordination of those components is still necessary, and if a precise and regular shape structure cannot be stated, there should be at least a pattern that covers how to construct an irregular structure.

Such a pattern is hard to find, and at this point we only have a proposal, namely the *Communicating Parallel Elements* pattern. This pattern is intended to address the situation when the design problem is expressed as a domain parallelism and different operations are executed simultaneously on ordered pieces of data [3,11]. Each component is expected to perform different operations, requiring and providing services intermittently. Commonly, the structure of this pattern should be presented as a network of relations between components, defined from the communication relations among components. Due to this, its shape may vary considerably from one application to another. Up to now, this pattern is still under discussion.

7. Acknowledgements

We are specially grateful to our shepherd for EuroPLoP'98, Frank Buschmann, for his comments, suggestions and critics, providing invaluable insights and promoting discussion and profound reflection about the contents to the improvement of this work. We are also grateful to Dirk Riehle for his in time intervention and precise

observations. 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

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

[2] 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.

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

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

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

[6] 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.

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

[8] 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.

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

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

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

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

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

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