The Parallel Layers Pattern

A Functional Parallelism Architectural Pattern for Parallel Programming

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

Abstract. *The Parallel Layers pattern is an architectural pattern for parallel programming used when the problem is understood in terms of functional parallelism. This pattern describes a solution in a layered form, in which each layer is composed of two or more components that are able to simultaneously exist and perform the same operation.*

1. Introduction

Parallel processing is the division of a problem, presented as a data structure and/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. The main advantage of parallel processing is its 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 centres on partitioning a big problem in order to deal with complexity. Partitioning is necessary to divide such a big problem into smaller sub-problems that are more easily understood, and may be worked on separately, on a more "comfortable" level. Partitioning is especially important for parallel processing, because it enables software components to be not only created separately but also executed simultaneously.

Requirements of order of data and operations dictate the way in which a parallel computation has to be performed, and therefore, impact on the software design [OR98]. Depending on how the order of data and operations are present in the problem description, it is possible to consider that most parallel applications fall into one of three forms of parallelism: *functional parallelism*, *domain parallelism*, and *activity parallelism* [OR98]. Examples of each form of parallelism are the Pipes and Filters pattern [OR05], representing functional parallelism; the Communicating Sequential Elements pattern [OR00], as an example of domain parallelism; and Shared Resource [OR03], as an instance of activity parallelism.

2. The Parallel Layers Pattern

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

Functional parallelism is the form of parallelism described in terms of a series of simultaneous step ordered operations, applied on ordered data with predictable organization and interdependencies. As each step represents a change of the input for value or effect over time, an amount of communication between components in the solution should be considered. Conceptually, data is repeatedly divided and transformed [CG88, Fos94, Pan96].

Example: Single-Source Shortest Path Algorithm

Search is defined as a systematic examination of a problem space, starting from an initial state and terminating at some final state or states. Each of the intermediate states, between the initial and the final states, can be reached by applying an operation on a given state. This operation is determined by an objective function that assures heading to the final state.

Any search problem can be conveniently represented using a graph. Given a graph is a set of vertices and edges. Each edge has a positive integer weight representing the distance between the vertices it connects (Figure 1). The objective, hence, is to search for the shortest path between the source vertex and the rest of the vertices.

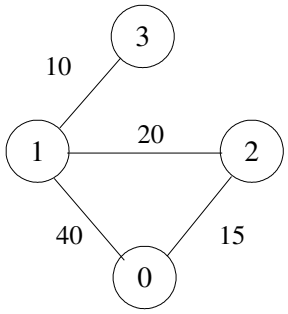

Figure 1. A typical graph

The Single-Source Shortest Path (SSSP) algorithm was originally proposed by Dijkstra, and described later by Chandy and Misra [CM88]. It is an efficient algorithm for exhaustively searching into this kind of graph representation. The SSSP algorithm is applied in cycles. In a cycle, the algorithm selects the vertex with the minimum distance, marking it as having its minimum distance determined. On the next cycle, all unknown vertices (those vertices whose minimum distance to the others has not been determined) are examined to see if there is a shorter path to them via the most recently marked vertex. Algorithmically, the SSSP algorithm reduces the search time to $O(N^2)$ because *N-1* vertices are examined on each cycle. Hence, *N1* cycles are still required to determine the minimum distances.

A sequential approach considers that the graph can be represented by an adjacency matrix G, whose elements represent the weight of the edges between vertices. In this approach two additional data structures are used: a boolean array `Known`, to determine which vertices have had their distance established, and an array `D` to record the most recently established distance between the source and vertices. A function `MinV` returns the vertex with the shortest

unknown distance of the two vertices passed as its arguments. If one vertex is known, the other vertex is returned. It is assumed that `MinV` is not called with two known vertices. The sequential pseudocode is shown in Figure 2.

```
Begin
   For i:=0 to N-1
     Known[i]:=(i=0) // only source vertex is known
   For i:=0 to N-1
     D[i]:= G[0,i] // initial distance of source to vertex
   LastKnown := 0 // only source is known
   KnownCount := 1

   While KnownCount < N
     MinVertex : = 0
     For i:= 1 to N-1 // check the shorter route via last marked vertex
        if Not Known[i]
           D[i] := Min(D[i], D[LastKnown] + G[LastKnown, i])
        MinVertex := MinV(MinVertex, i)
     End For
     // select next vertex to mark known
     LastKnown := MinVertex
     Known [LastKnown] := TRUE
     KnownCount ++
   End While
End
```

Figure 2. Pseudocode for the sequential SSSP algorithm.

However, this algorithm can potentially be carried out more efficiently by:

1. Using a group of parallel components that exploit the tree structure representing the search, and
2. Simultaneously calculating the value minimum distance for each vertex, and only then, computing and marking the overall minimum distance vertex.

Context

Starting the design of a software program for a parallel system, using a particular programming language for certain parallel hardware. Consider the following contextual assumptions:

- The problem to solve, expressed as an algorithm and data, is found to be an open ended one, that is, involving tasks of a scale that would be unrealistic or not cost-effective for other systems to handle. Consider the SSSP algorithm example: since its execution time is $O(N^2)$, if the number of vertices is large enough, the whole computation grows up to an enormous extent.
- The parallel platform and programming environment to be used are known, offering a reasonably level of parallelism in terms of number of processors or parallel cycles available.
- The programming language to be used, based on a certain paradigm, is determined, and a compiler is commonly available for the parallel platform. Many programming languages

have parallel extensions for many parallel platforms [Pan96], as it is the case of C, which can be extended for a particular parallel computer or use libraries to achieve process communication [ST96].

- The main objective is to execute the tasks in the most time-efficient way.

Problem

An algorithm is composed of two or more simpler sub-algorithms, which can be divided into further sub-algorithms, and so on, recursively growing as an ordered tree-like structure until a level in which the sub-parts of the algorithm are the simplest possible. The order of the tree structure (algorithm, sub-algorithms, sub-sub-algorithms, etc.) is a strict one. Nevertheless, data can be divided into data pieces which are not strictly dependent, and thus, can be operated on the same level in a more relaxed order. If the whole algorithm is performed serially, it could be viewed as a chain of calls to the sub-algorithms, evaluated one level after another. Generally, performance as execution time is the feature of interest. Thus, how do we solve the problem (expressed as algorithm and data) in a cost-effective and realistic manner?

Forces

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

- Perform a computation as a tree structure of ordered sub-computations. For example, in the SSSP, each minimum distance for each vertex is calculated using the same operation several times, but using different information per layer.
- Data can be only vertically shared among layers. In the SSSP example, data is distributed through the tree structure, where autonomous operations are carried out.
- The same group of operations can be independently performed on different pieces of data. In the SSSP example, the same operation is performed on each subgroup of data to obtain its minimum distance from the lower layers. So, several distances can be obtained simultaneously.
- Operations may be different in size and level of complexity. In the SSSP example, operations are similar from one layer to the next, but the amount of data processed tends to diminish.
- Dynamic creation and destruction of components is preferred over static, to achieve load balance. For example, in the SSSP example, the creation of new components in lower layers can be used to extend the solution to larger problems.
- Improvement in performance is achieved when execution time decreases. Our main objective is to carry out the computation in the most time-efficient way. The question is: how can the problem be broken down to optimise performance?

Solution

Use functional parallelism to execute the sub-algorithms, allowing the simultaneous existence and execution of more than one instance of a layer component through time. Each one of these instances can be composed of the simplest sub-algorithms. In a layered system, an

operation involves the execution of operations in several layers. These operations are usually triggered by a call, and data is vertically shared among layers in the form of arguments for these function calls. During the execution of 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

In this architectural pattern, different operations are carried out by 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 calls, referring to each other as components 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. An Object Diagram, representing the network of components that follows the parallel layers structure is shown in Figure 3.

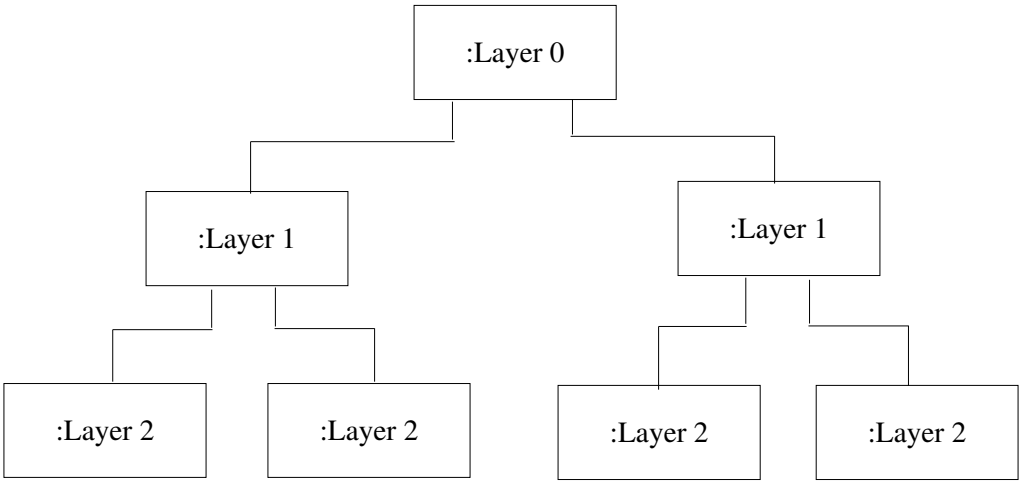

Figure 3. Object Diagram of the Parallel Layers pattern.

Participants

- **Layer component**. The responsibilities of a layer component are to allow the creation of an algorithmic tree structure. Hence, it has to provide a level of operation or functionality to the layer component above, while delegating operations or functionalities to the two or more layer components below. It also has to allow the flow of data and results, by receiving data from the layer component above, distributing it to the layers components below, receiving partial results from these components, and making a result available to the layer

component above. Each component is independent from the activity of other components. This makes it easy to execute them in parallel.

Dynamics

As the parallel execution of layer components is allowed, a typical scenario is proposed to describe its basic run-time behaviour. All layer components are active at the same time, accepting function calls, operating, and returning or sending another function call to other components 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):

- 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].

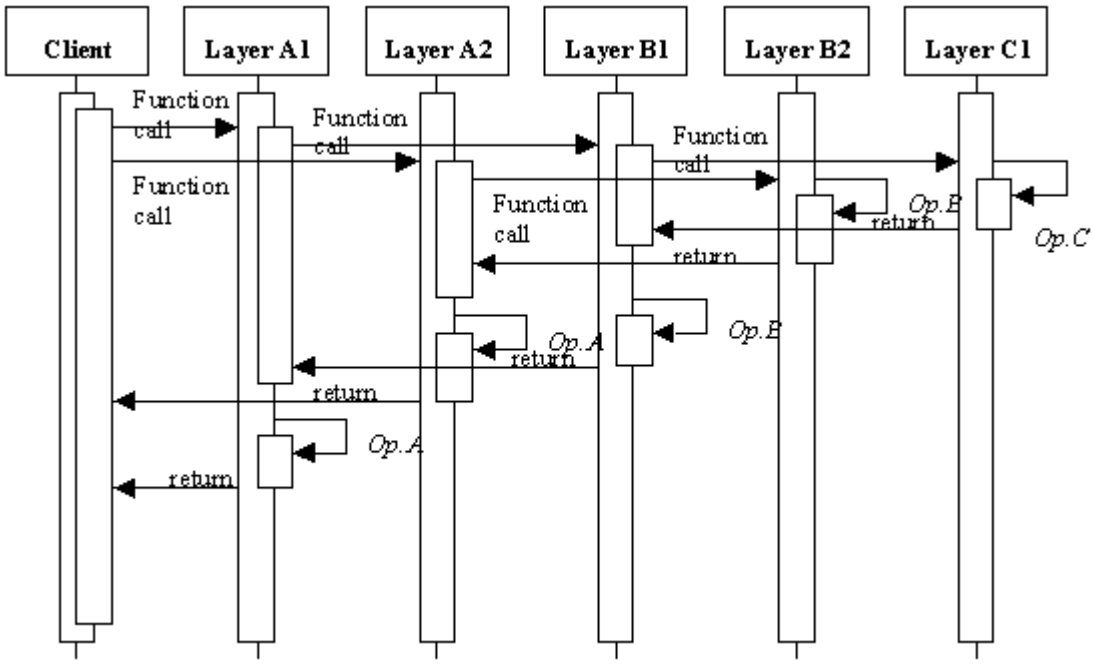

Figure 4. Interaction Diagram of the Parallel Layers pattern.

- 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.

Implementation

An architectural exploratory approach to design is described below, in which hardware-independent features are considered early, and hardware-specific issues are delayed in the implementation process. This method structures the implementation process of parallel software based on four stages [OR98]. 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 as general considerations for design instead of providing details about precise implementation. 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, POSA00].

1. *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. 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 layers, like *A Hierarchy of Control Layers* [AEM95] and the *Layered Agent Pattern* [KMJ96].

2. *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. 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, POSA00] and the *Visibility and Communication between Control Modules* and *Actions Triggered by Events* [AEM95] can help to define and implement the required communication structures.

3. *Agglomeration.* The hierarchical structure is evaluated with respect to the expected performance. Usually, systems based on identical layer components 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.

4. *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 of layers on a processor, but if the number of requests is large, some layers 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 some processors, introducing load-balance problems. The most realistic approach seems to be a combination of both, 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 Layers* 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].

Example Resolved

The potential parallelism for the SSSP is explained as follows. On each cycle, the current distance to a given vertex must be compared to the distance to the vertex via the last known vertex and the minimum recorded as the new distance. This calculation depends only on the graph array G. Thus, the minimum distance for each vertex can be computed and marked. If there are N processes, the algorithm would have a running time $O(N\log_2 N)$. N-1 cycles are still required to compute the minimum of all vertices. However, each cycle will require one time step to update the minimum for each vertex and $O(\log_2 N)$ time steps to compute the overall minimum vertex.

To move to a parallel solution, we must determine two things:
1. the communications network topology that will be used, and
2. what information will be stored on the processors and what will be passed as messages.

Partitioning

Both communication and computation of a minimum can be done in $O(\log_2 N)$ time by using a cubic array of processes. In such an arrangement, each process would compute its minimum

distance; then half of the processes would pick the minimum between its distance and that of a neighbour in one dimension (Figure 5). Half of these processes would in turn select a minimum, until the root process selects the global minimum distance vertex. Communication and selecting the minimum can be done in $O(log_2N)$ time, assuring an overall $O(Nlog_2N)$ performance.

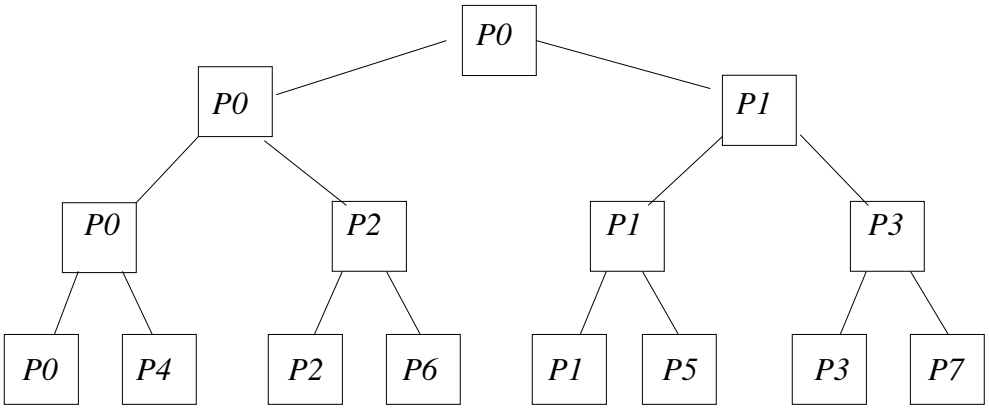

Figure 5. Tree representation for the SSSP algorithm.

Communication

The communication for *N* processes has to consider how to distribute data over the network of processes. This is done by reviewing the computations of a root and children processes, and determining what data must be available for the computations.

The root process *P0* calculates which of the two vertices has the shorter unknown distance. To do so, it must have available which vertices have already had their distances marked (the array `Known`), and the distance and id of the vertices being compared.

The children processes, on the other hand, must compare their current vertex distance to the distance between the last known vertex and themselves. Thus, they must have available the original graph G and the distance and id of the last known vertex. In addition, some children processes will be calculating the minimum between two vertices, so they will also need to know which of the vertices are known.

The basic data that needs to be communicated between processes is the id of the vertex and its most recent distance. This data will be used to calculate the minimum distance vertex and to announce which vertex has been marked as known. Thus, a message is a two-element array, one being a vertex id, the second a distance.

Since the message marking a vertex is distributed to all vertices, each process can keep track of which vertices are known. Thus each should locally store and update the array `Known`. Likewise, the graph G, which is not changed during the computation, must be distributed to all processes and stored locally before computation begins.

Finally, the function `MinV` would no longer have access to the array `D` to look up the distances of the vertices being compared. The parameters must be changed so that the distances of the vertices being compared are passed as well as the vertex identifiers.

Agglomeration and Mapping

If a 3D-cube is used for the computations (Figure 6), the code for synchronising and communicating between the root process and the remaining processes would be as the one shown in Figures 7 and 8, respectively.

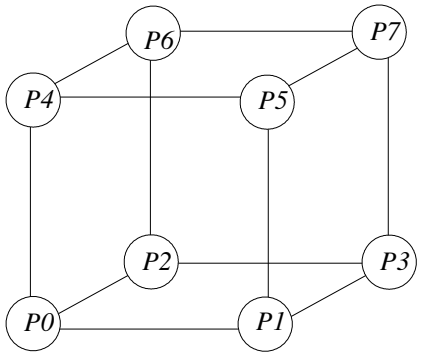

Figure 6. A 3D-cube.

```
Process 0 (the root process)
    i := 1
    While i < N
       // receive distances from 3 neighbours
       MinVertex := 0
       receive vertex id from z dimension
       MinVertex := MinV(MinVertex,Zvertex)
       receive vertex id from y dimension
       MinVertex := MinV(MinVertex,Yvertex)
       receive vertex id from x dimension
       MinVertex := MinV(MinVertex,Xvertex)
       Known[MinVertex] := TRUE // Update Known array
       LastKnown := MinVertex
       distribute LastKnown out x, y and z // Inform neighbours of the result
       i++
     End While
End Process 0
```

Figure 7. root process (Process 0).

```
Process k, 1<=k<N
// The remaining processes
     i := 1
     While i < N
        // find overall unknown minimum distance vertex
        LocalMinVertex := k
        if k < 4 then begin
           // processes 1, 2, and 3 receive and compute min
           receive Zvertex from z dimension
           LocalMinVertex := MinV(LocalMinVertex,Zvertex)
        else
           // processes 4, 5, 6, and 7 send out their vertices
           send LocalMinVertex out z dimension
        if k < 4 then
           // processes 4, 5, 6, and 7 do nothing
           if k = 1 then begin
               // process 1 receives and computes minimum
               receive Yvertex from y dimension
               LocalMinVertex := MinV(LocalMinVertex,Yvertex)
           else
               // processes 2 and 3 send out their vertices
               send LocalMinVertex out y
        if k = 1 then
           // process 1 sends its local min to process 0
           send LocalMinVertex out x
        // now receive overall minimum vertex LastKnown from Process 0
        if k = 1 then begin
           // process 1 receives from 0, distributes to 3
           receive LastKnown in x dimension
           send LastKnown in y dimension
        else
           if k < 4 then
               // processes 2 and 3 receive from 0 and 1, distribute to 4 and 5
               receive LastKnown in y dimension
               send LastKnown in z dimension
           else
                // processes 4, 5, 6, and 7 receive from 0, 1, 2, and 3
               receive LastKnown in z dimension
        D[k] := Min(D[k],D[LastKnown]+G[LastKnown,k]")"
        // now update Distances
        i++
     End While
End Process k
```

Figure 8. The children processes (Process *k*).

Synchronisation is achieved by the links between processes. Thus process 3 cannot compute the minimum distance vertex between itself and process 7 until process 7 sends its distance. Once computed, it sends the distance to process 1, which in turn waits until this message is received to compute the minimum between processes 3 and 1.

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 Layers* 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 is solved using a Parallel Layers 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].

Consequences

Benefits

- The *Parallel Layers* pattern, as the original *Layers* pattern, is based on increasing levels of complexity. This allows the partitioning of the computation of a complex problem into a sequence of incremental, simple operations [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 efficiently structured as layers. Considerations of performance may require a strong coupling between high-level functions and their lower-level implementations. Load balance among layers is also a difficult issue for performance [SG96, Pan96].
- Many times, a layered system is not as efficient as a structure of communicating components. If services in upper layers rely heavily on the lowest layers, all data must be transferred through the system. Also, if lower layers perform excessive or duplicate work,

there is a negative influence on the performance. In certain cases, it is possible to consider a Pipe and Filter architecture instead [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].

Related patterns

The *Parallel Layers* 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*. The *Divide and Conquer* pattern [MSM05] describes a very similar structural solution to the *Parallel Layers* pattern. However, its context and problem descriptions do not cope with the basic idea that, in order to guide the use of parallel programming, it is necessary to analyse how to divide the algorithm and/or the data to find a suitable partition, and hence, link it with a programming structure that allows for such a division.

3. Summary

The goal of the present work is to provide software designers and engineers with an overview of the Parallel Layers pattern as a description of a common structure used for parallel software systems. Its application depends on the feasibility of the algorithm to be expressed in the form of a tree, which maps into the layers structure. Also, such an application is based on allowing data to be divided into pieces which are operated without a dependence among themselves. The architectural pattern described here is directly related with several developments in the field of algorithmic analysis, where it is proven its efficiency when dealing with fixed size problems. This pattern can be also 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, POSA00].

4. Acknowledgements

The author wishes to thank Joseph W. Yoder, my shepherd, for his important suggestions and advises for the improvement of this paper. This paper has been developed as part of the Subproject EN101603 of the Support Program to Institutional Projects for Teaching Improvement (PAPIME), supported by DGAPA-UNAM.

5. References

[AEM95]Aarsten, A., Gabriele Elia, G., and Giuseppe Menga, G. *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing.* Department of Automatica e Informatica, Politecnico de Torino. In J. Coplien and D. Schmidt (eds.) Pattern Languages of Program Design. Reading, MA: Addison-Wesley, 1995.

[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.

[CM88] K. Mani Chandy and J. Misra. *Parallel Programming Design.* Addison-Wesley, New York, 1988.

[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.

[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.

[MSM05] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. A Pattern Language for Parallel Programming. Addison Wesley Software Patterns Series, 2005.

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

[OR98] Jorge L. Ortega-Arjona and Graham Roberts. *Architectural Patterns for Parallel Programming.* Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing, EuroPloP'98. Universitätsverlag Konstantz GmbH, 1999.

[OR00] Jorge L. Ortega-Arjona. *The Communicating Sequential Elements Pattern.* Proceedings of the 7th Annual Conference on Pattern Languages of Programming, PloP'98. Washigton University Technical Report wucs-00 29, 2000.

[OR03] Jorge L. Ortega-Arjona. *The Shared Resource Pattern.* Proceedings of the 10th Annual Conference on Pattern Languages of Programming, PloP 2003. Washigton University Technical Report wucs-00 29, 2000.

[OR05] Jorge L. Ortega-Arjona. *The Pipes and Filters Pattern.* Proceedings of the 10th European Conference on Pattern Languages of Programming, EuroPloP 2005. Universitätsverlag Konstantz GmbH, 2005.

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

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

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

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

[POSA00] Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2. Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Ltd., 2000.

[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.

[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.