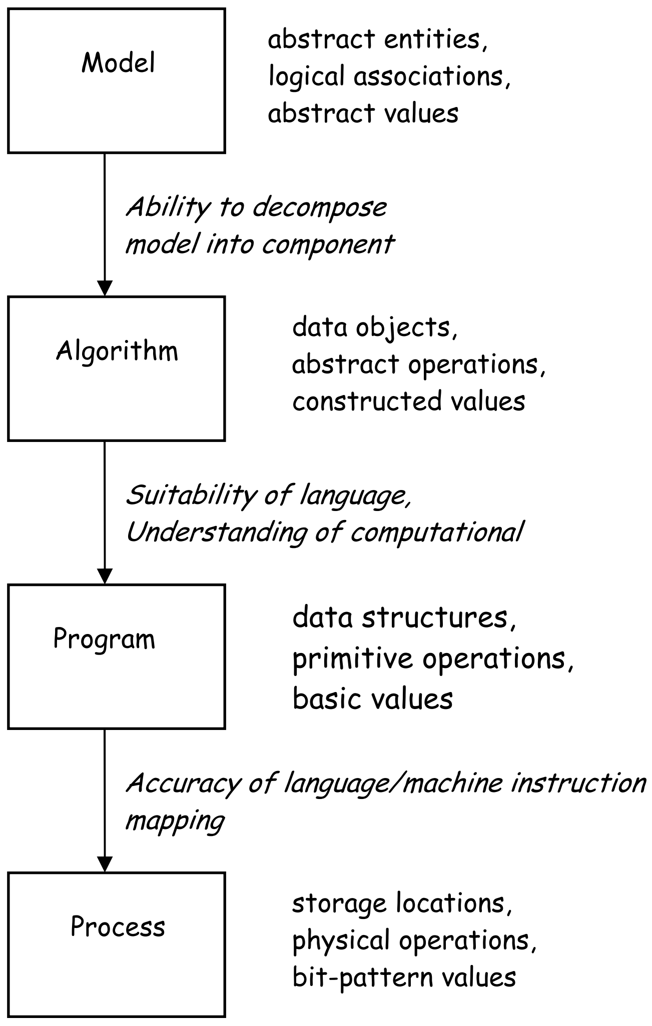
Parallel Software Design

A parallel program is *'the specification of a set of processes executing simultaneously, and communicating among themselves in order to achieve a common objective'*. This definition is obtained from the original research work in parallel programming provided by E.W. Dijkstra [Dij68], C.A.R. Hoare [Hoa78], P. Brinch-Hansen [Bri78], and many others, who have established the main basis for parallel programming today. Moreover, the current use of parallel computers implies that software plays an increasingly important role. From clusters to supercomputers, success heavily depends on the design skills of software developers. Specifically, obtaining a parallel program from an algorithmic description is the main objective of the area of Parallel Software Design. This area is proposed to study how and at what point the organisation of a parallel program affects the development and performance of a parallel system.

1. A General Parallel Software Design Process

The term 'Parallel Software Design' is considered here as a problem solving activity. While many software developers and programmers solve problems routinely, developing programs, the problems that parallel software designers face tend to be very complex. An important distinction between Parallel Software Design and other types of Software Design is that parallel problems are solved by applying specialised scientific and mathematical approaches, abstract knowledge and, largely, a lot of intuition.

Most Parallel Software Design problems are so complex that initially no software designer can foresee a solution. To produce a parallel program as a solution, the software designer should proceed methodically, solving the problem in a step by step process. A general design process that is used for Parallel Software Design is shown in Figure 1 [Pan90].



Model
abstract entities, logical associations, abstract values
Ability to decompose model into component
Algorithm
data objects, abstract operations, constructed values
Suitability of language, Understanding of computational
Program
data structures, primitive operations, basic values
Accuracy of language/machine instruction mapping
Process
storage locations, physical operations, bit-pattern values

Figure 1.

In the particular case of parallel systems, when dealing with scientific problems, the general design process usually can be characterized as four different levels of abstraction (Figure 1). Each level defines its own collection of objects, a set of operations or manipulations applicable to those objects, and domains representing the values each object may assume [Pan90]:

- Model level. At this level, the problem is expressed in abstract terms, and the solution is described in terms of abstract entities, logical associations, and abstract values. The solution is outlined in general terms, regardless the computer system on which it will be executed. Commonly, the descriptions are made in natural language or diagrams. At this level, the software designer may start to notice portions of the solution as candidates for parallelization. Normally, since scientific users look for performance improvement, attention has to be focused on activities that may be computationally intensive in the implementation [Pan90].

- Algorithmic level. This level defines a set of specific steps required to solve the problem. Even though the operations are still abstract, they are applied to data objects with specific range of values. Typically, algorithmic solutions are thought as a group of sequential steps. The descriptions are made in a notational form chosen on the basis of appropriateness to the model from the previous level, and not precisely related to the computing environment in which the solution will be executed. Nevertheless, the algorithmic specification reflects the fact that the solution will be carried out on a computer system. There is generally no explicit mention of parallelism in the algorithmic solution. At most, if parallelism occurs at this level, it is limited to the notion that two or more steps of the algorithm may be allowed to proceed concurrently [Pan90].
- Program level. This level describes the problem in terms supported by a programming language: data structures, primitive operations, and basic values. The selected programming language imposes a rigorous formalism, but at the same time, attempts to provide expressiveness, generality, and portability. This is the most challenging phase since the software designer must devise concrete representations of all data and operations, and describe them in the restrictive notation of a programming language. It is common that parallelism is incorporated at this level of description. Commonly, a sequential solution is developed first, adding parallel features once the designer is confident that the solution works [Pan90].
- Process level. This level involves a description of the solution based on computer terms: storage locations, physical operations, and bit-pattern values. This representation is commonly obtained from compiling the programming language description of the previous level. Parallelism at this level is reflected by the fact that software portions or components are allowed to be simultaneously executed, depending on the programming language description of the previous level [Pan90].

Notice from Figure 1 that the descriptions of a parallel system provided at each level require transformations from the previous level in order to obtain a new description to be passed to the next level. The initial abstract model has to be transformed into an

algorithmic solution, which has to be manually transformed into program code, and automatically transformed into executable code. Notice that each transformation is a source of potential error and distortion.
Nevertheless, the first and third transformations are normally considered to present no particular problems, since both them benefit from past research experience in modelling and compiling. The first, from model to algorithm, is bounded by the designer's ability to decompose an abstract model into a sequence of suitable, high-level representations and operations. The third, from program to process, is bounded by the accuracy of mapping language constructs to machine instructions, relying on compiler technology, and beyond the control of the designer.

In contrast, the second transformation, from algorithm to program, poses special difficulties for the parallel software designer. Since it involves translation from a logical to a quasi-physical form, its success relies on the designer's understanding of parallel computation methods. It is at this transformation that Software Patterns may provide aid for the parallel software designer. Even though the conceptual support provided by Software Patterns is useful at any level of abstraction, this book mainly focus on the Software Patterns for design a parallel program taking as input an algorithmic description.

2. A Pattern-based Parallel Software Design Method

During the development of a parallel application, experienced software designers may try different methods depending on their understanding of the different levels of abstraction within the general design process. The study of different methods is called 'methodology'. A software designer selects a method depending on a number of contextual design particularities: the complexity of the design, size of the design team, experience, personal style, preferences, and so on. No matter what method is selected, the goal is the same: obtain the parallel program with the best possible performance, with the least design effort and implementation cost, and in the shortest possible time. Selecting the adequate method has an enormous impact on how long it takes to solve a problem.

Conflicting requirements contribute to complicate the selection of a method. For example, producing the best performance solution may conflict with minimising costs. In fact, the 'best' solution is determined by finding a balance among performance, cost, reliability, maintainability, and so on. But it may only be possible to accomplish improving performance

or reducing costs by investing in a thorough analysis which, by the way, may at the same time increase costs and extend development time.
Performance and cost are the main features considered in the problem specification for most parallel programs. They are used as the ultimate criteria for reviewing design alternatives. Performance refers to how a parallel program carries out its function. Cost refers to the cost and effort to construct the parallel program.

Other features that are normally specified for parallel program alternatives are reliability and maintainability. Reliability is related with the frequency of failure of the parallel system once in use. Maintainability is related with the cost, expertise, and any other resources needed to keep the parallel system operational during its lifetime.
Evaluating different parallel design alternatives is complicated due to conflicting features. It may be possible, commonly, to improve performance by increasing costs. The requirements of the parallel design solution must be specified in advance for each of these two features, preferably with a range of permissible limits. Alternatives are revised against each criterion, being the 'best' design the one that fits to a performance/cost balance. Moreover, this implies that exceeding a specified need is not necessarily better. For example, it is almost possible to attempt to improve the performance of a parallel program, but this does not completely means that this is the best design alternative if this solution does not meet specifications of cost, reliability, or maintainability. Commonly, a parallel program with a lower performance happens to be cheaper to produce and easier to maintain. This, therefore, would be the best alternative at hand, as long as it solves the main aspects of the problem. In practice, however, a best solution is up to some extent subjective.

In Parallel Programming, the used design method tends to impact both efficiency and effectiveness of a software designer. Nevertheless, the ability to select and/or adapt a design method given a design context only comes with experience. Hence, in order to support the software designer, an initial, general, and practical design method is proposed as follows. This design method is advised in general when quality and limited resources (which seem to be the norm) are required.

A ‘good’ design software method determines early in the development process if a solution for a given problem can be found. Not all problems, and particularly, parallel programming problems, have a solution. Perhaps one among several parallel programming problems is taken to a successful end. Since the cost and effort wasted on the rest of the attempts should be recovered from the only successful one, it is very important that the cost of failure is made as small as possible.

Figure 2 shows a Pattern-based Parallel Software Design Method that is proposed here to produce solutions, attempting to minimise design effort, whereas early establishing a solution. This method is described as a step by step process, addressing a piece of the solution description at the end of each step. Thus, each step is considered as a sub-problem, in which activities are described in general terms, and whose documentation is described as well in general terms after each step. Thus, the solution of each step is expressed in terms of documentation and some developed code which incrementally describes part by part of the parallel program. At each stage, the documentation as a description (except from the first step) is obtained using a basic design process.

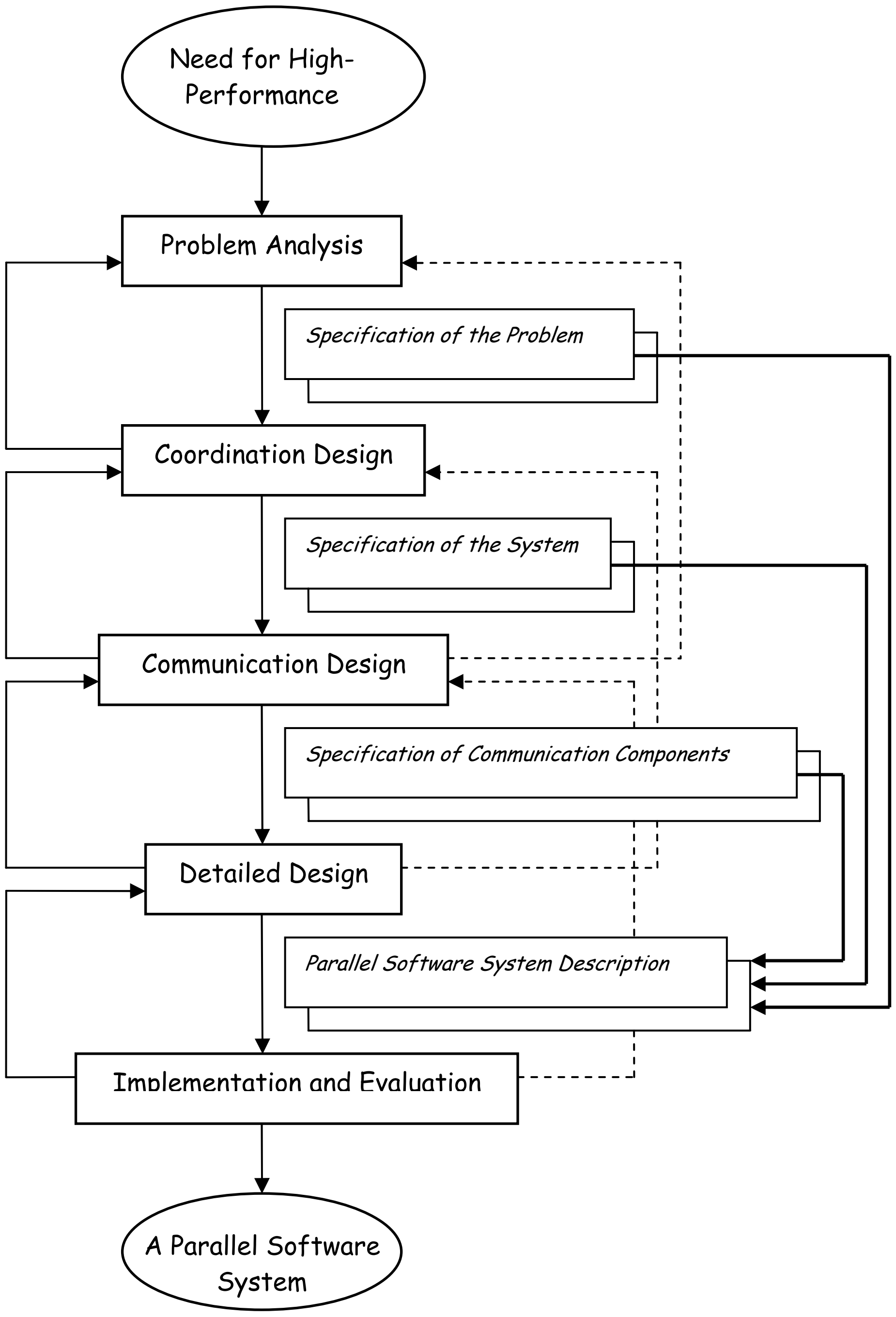

Need for High-Performance
Problem Analysis
Specification of the Problem
Coordination Design
Specification of the System
Communication Design
Specification of Communication Components
Detailed Design
Parallel Software System Description
Implementation and Evaluation
A Parallel Software System

Figure 2.

The method shown in Figure 2 is realistic, since it allows correcting errors that are discovered in a later step. The sooner an error is found, the less expensive it is to correct it. This is the reason that the method proposed here is developed to allow occasionally going back perhaps just one or two steps (as the arrows in the figure indicate).

This design method aims to arrive at a 'best' or 'optimum' parallel program as solution. However, in most practical situations, real optimum solutions are impossible to arrive to. The best that a software designer can achieve is the best among several alternatives. This raises the question: 'how to decide whether one of several solutions is the best alternative?' As shown in Figure 2, the method initiates with a specifications of the problem to be solved. This specification becomes the measure against which alternative solutions are reviewed. If a parallel design solves almost completely the problem, then it is considered better than any other parallel design which falls short. The problem specification is then a very important step in the design method.
An overview of the design method for Parallel Software Design is provided in the following sections. It is based on the general design process of parallel systems and the three different categories of Software Patterns: Architectural Patterns, Design Patterns, and Idioms. It is described in terms of concepts such as 'Problem Analysis', 'Coordination Design', 'Communication Design', 'Detailed Design', and 'Implementation and Evaluation'. These steps of the method are described in depth, organising them sequentially from top to bottom. From a Parallel Software Architecture point of view, the two most important steps are the algorithm and data analysis and the coordination design. Each of these points is covered, hence, as follows.

3. Problem Analysis

Parallel Software Design, as any design activity, goes from a statement of the problem, normally in terms of a function and a set of requirements about how such a function is to be carried out, to a form of a parallel organisation of components and a set of properties of such a form. As such, the very initial and obvious step to take is to understand the problem. Nevertheless, unfortunately this is very often ignored. Since most designers and programmers have an enormous desire of 'getting down to program', they fail to pay attention on fully understanding the problem.

The *Specification of the Problem* is the first document that the design method outputs (Figure 3). The specification document should have: (a) a description of the data to be operated on, (b) a description of the algorithm that operates on such data, (c) contextual information about the parallel platform and the programming language to be used, and (d) particular requirements about performance and cost. In such terms, this document establishes a reference against which the solution is going to be evaluated. Hence, the descriptions included in this document should attempt to answer the questions such as 'what is the problem that is to be solved?', and 'what exactly is the parallel system going to achieve?'

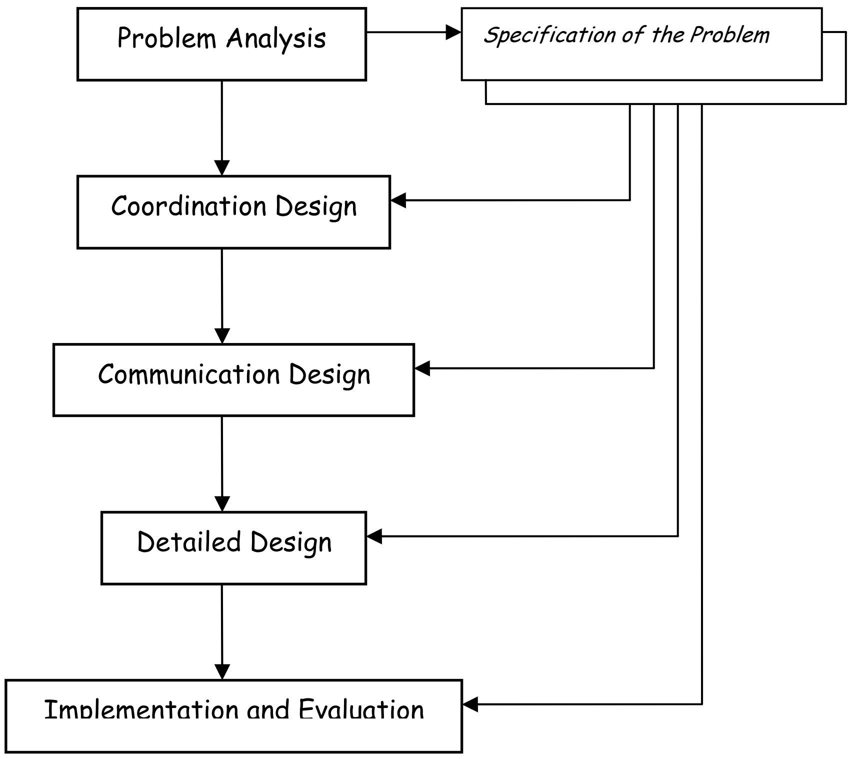


Figure 3.

The Specification of the Problem should answer as well another important question: 'how do we know if the parallel system does what it is supposed to do?' Hence, the Specification of the Problem provides a reference or criteria to verify whether or not the parallel system accomplishes the objectives it is designed for. Moreover, it may be used to generate tests in order to evaluate the very parallel system.

Also, the Specification of the Problem provides milestones that help the software designer to decide between alternatives of the design process. From beginning to end, design can be considered as a decision process, in which every decision provides with new constraints to the following decision. Thus, the Specification of the Problem works like a sieve, helping the software designer to eliminate those potential parallel solutions that are far too ambitious, require intractable parts, or fail to address important requirements. Very few parallel projects result in a successful, viable parallel system capable of addressing all requirements. However, since cost and effort of changes tend to exponentially grow while design progresses, early identifying a parallel solution that does not provide a satisfactory performance/cost balance is clearly an asset within the design development.

When writing the Specification of the Problem, the focus should be on the information provided by the user of the parallel system. Remember: at this stage the software designer aims not to solve the problem, but rather, to understand what it is about. Thus, the objective here is to define and quantify the function and requirements of the prospective parallel system, and write all them down in the form of a document: the Specification of the Problem. Also, the document should take into consideration contextual design elements, such as a simple and brief statement about the parallel hardware platform and parallel programming language used, providing references where to obtain information about them.

It tends to be difficult to write the description of all the elements that compose the Specification of the Problem, while avoiding providing a solution. When a software designer or programmer is confronted to a problem, immediately tries to solve it. This is not completely wrong if you are an experienced designer or programmer. Experienced software designers and programmers are considered so since they provide (most of the times) a 'right' solution to a problem. Nevertheless, in parallel programming, not many software designers or programmers are experienced ones. This is so since Parallel Software Design poses several variations not yet explored. They are still under research. Hence, it is advised here that before attempting to provide any idea of the solution, first understand the problem at least in terms of the data to be operated, and the algorithm to be used on the data.

As Parallel Software Design is based on decisions based on experience and information rather than on well established knowledge, many decisions should be taken by collaborating with the user. The user is normally an expert of an area which desires to obtain performance, and with that objectives, the user makes use of parallel programming. So, at this stage, the user is the main source of information, although further information is also obtained from other sources. So, the description of the problem in terms of algorithm, data, and performance/cost requirements are described, organised, and presented to the user. The objective is defining the problems completely and clearly as possible. Hence, the software designer should act here as an expert as well as a good listener. Again, do not attempt to solve the problem; understand it. This is complex, since a lot of patience and experience are required not to rush a solution. The objective is not search, provide, or evaluate different solutions, but attempt to describe the problem.

In order to derive the specification of the problem as a document, a two step procedure can be used (Figure 4). In the first step, the user's needs are written down and organised straightforward in the terms of the user into an informal 'Problem Statement' document. Here, every technical and quantifiable information is avoided. The main idea is to present the problem as simply as possible, and the way in which the user understands it.

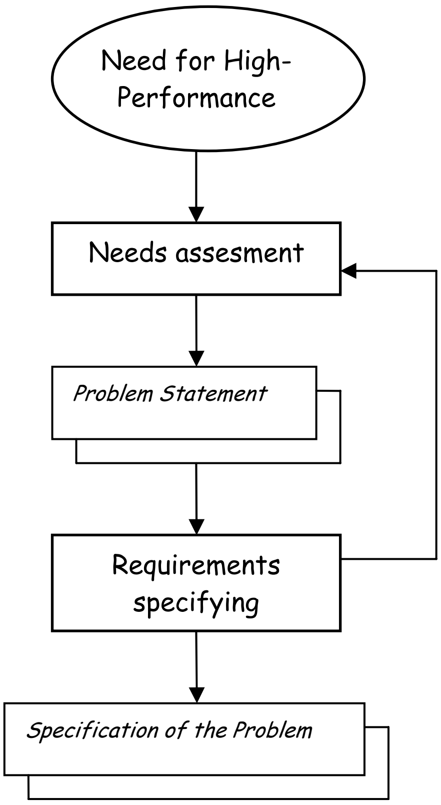


Figure 4.

In the second step, the Problem Statement is revised and re-stated, including more detailed information. The main idea is to convert this informal document into the Specification of the Problem, by including more technical and quantifiable information.

Obtaining a translation from the Problem Statement to the Specification of the Problem normally means that each need is mapped into a specification. If the Problem Statement is complete and sound, it will yield a Specification of the Problem complete and sound as well. Complete means that all needs are taken into consideration. Sound means that different needs do not contradict between them.

While developing the Specification of the Problem, the Problem Statement is revised over and over, continually consulting the user in order to make sure that real needs are covered and eliminate any inconsistency. This is the most important feature of the two steps procedure towards obtaining the Specification of the Problem. Iterating and reviewing the descriptions and definitions provided by the user is required since questions go arising as the software designer revises and re-states the user's information. As the function in terms of data and algorithm provided by the user is clarified, questions about how such a functionality has to be carried out give rise to requirements. During the iterative procedure, the software designers should take advantage in each cycle to make clarifications and propose agreements with the user, and review the information when needed.

After the iterative procedure, the Specification of the Problem is obtained as a formal document that describes as clearly as possible the issues related with the user requirements. It represents an agreement between software designer and user, which is used in later steps to aide decision making among options, as well as the criteria to judge the parallel system as a whole, whether it accomplishes or not its objectives when the parallel system is evaluated. The idea behind this document is to let everyone in the project what they are aiming to do, and how to know when it is done.
How formal the Specification of the Problem can be depends on the situation. Such a situation may go from the development of a single parallel, to providing services for parallel software development. If the parallel system is developed only for internal 'number crunching' purposes, it may be regarded as a more informal, internal document of the project, an agreement of what the project team should expect from the parallel system. Nevertheless, if the parallel system is designed for an external user, whether it is a government or a company, it normally requires further clarity in the description of what the prospective parallel system should accomplish, and thus, the document should be as

formal as possible, since it could be used even at a legal level or as part of a contract. No matter the level of formality of the Specification of the Problem, it is important that the document has the acceptance of the user and the software designer.

At this point, it may be interesting to warn about the information provided by the user. The iterative interaction with the user requires questioning over and over, always trying to specify the problem as clearly as possible. Furthermore, it is always useful to distinguish between what the user actually needs and what the user only desires. Normally, user's needs and desires are not the same, and if the Specification of the Problem is written down considering more likely desires than needs, the resulting design will fail to provide some real needs, so it will be deficient. Moreover, desires that are misguided as needs will impose an extra cost, making the design more expensive, while not providing the actual function that the user needs. Therefore, it is the software designer's responsibility to extract and differentiate user's needs and desires, and obtaining an Specification of the Problem that consider real and feasible needs. Even though it is not likely that the Specification of the Problem may cover every and all needs, it is very important for the success of the project that it addresses at least the most critical ones. The effort to get this as right as possible will be rewarded later during design steps.

Finally, the objective is to document of the Specification of the Problem. The content may vary, but in general for parallel systems, it covers:

1. Overview. The overview attempts to address the main question 'why the parallel system is needed and what it is expected to achieve?' Normally, the overview can be considered as an 'executive summary'. Hence, the overview is a summary of the whole document.
2. The Problem Statement describes the problem in the user's terms, as well as the requirements agreed with the user. It is important to present them explicitly, adding every clarification agreed between the software designer and the user.
3. The descriptions of the data and the algorithm. A description of the data to be operated on and a description of the algorithm that operates on such data. They are the basic element to start with initial decisions during coordination design.
4. Information about the parallel platform and the programming language. The performance that a parallel system is capable of achieve is directly affected by

the parallel hardware platform and the parallel programming language used. Basic information about these two issues is important, at least in the form of references to where broader descriptions of both, platform and language, can be found.

5. Quantified requirements about performance and cost. The idea behind this is to state clearly what is expected of the parallel system. This information is used later, when testing the parallel system in order to establish if it accomplishes such basic requirements or not. Also, this section may include information about other requirements (such as reliability, maintainability, or others) which are considered important, so they are taken into consideration during the design.

Every section covers textual information as well as figures, tables, or any information which may considered relevant during the design of the parallel program. The intent is to create a single document that serves as reference to the software designers as they advance to the Coordination Design, the next stage of this design method.

4. Coordination Design — Architectural Patterns

Once the problem has been analysed as a function in terms of data, an algorithm and several other issues that impact on the design, the design method advances to proper designing. From a Parallel Software Architecture point of view, this activity begins with a description of the parallel software as a coordinated system, and thus, developing the coordination of the parallel software. As it is shown in Figure 5, Coordination Design is the second step of the design method. As any design activity, the objective of the Coordination Design step is to produce a document that provides a description of the parallel software in system terms, this is, the specification of the system. This document should fully describe the parallel software as a form that carries out a function; this is, as the components or parts that compose it, as well as the particular functionalities that each one of them performs. Also, it should consider how the proposed form meets the requirements as stated in the specification of the problem.

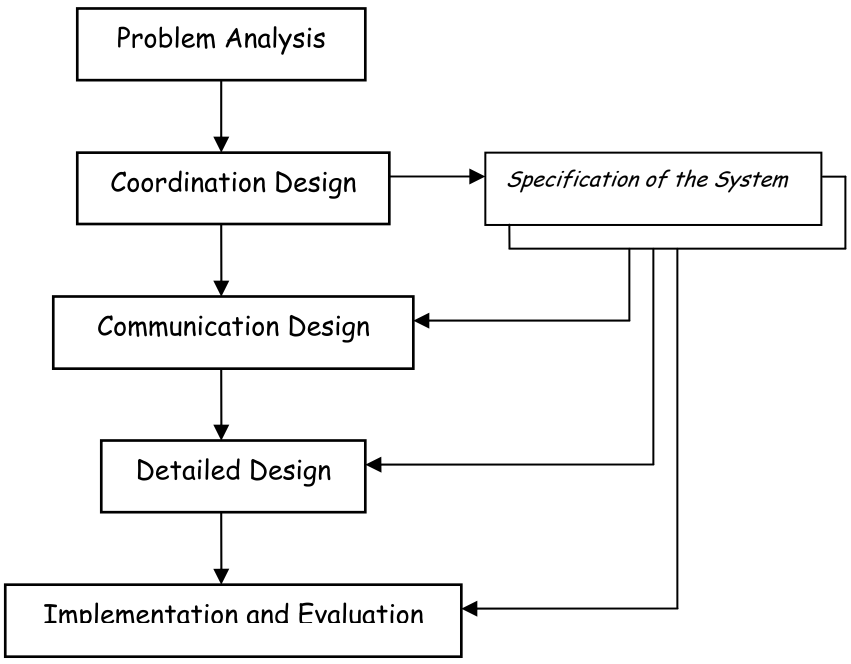


Figure 5.

Designing at the coordination level is a creative activity, involving scoping, analysis, synthesis, iterative refinement, and finally document all design decisions as the ideas that seem to pose the 'best' organisation for the problem at hand.

In a Pattern-based approach to Parallel Software Design, Architectural Patterns for Parallel Programming [OR98, Ort00, Ort03, Ort04, Ort05, Ort07a] are used here. As they specify the problem they solve as a function in terms of data and algorithm, linking it with a solution as a form describing an organisation of parallel software components that simultaneously execute. So, Architectural Patterns are used here to select a coordination for the parallel software, which at the same time represents a form of the parallel software as a whole [OR98].

The importance of Coordination Design relies on the novelty and innovation which originate in this step. Several requirements depend on the Coordination Design, requirements such as performance, cost, maintainability, reliability, and so on. Particularly, it is in this step where performance can be improved.

The division of a problem into small and manageable components is the essence of parallel programming. Software designers partition the data and/or the algorithm of the problem into smaller sub-data and/or sub-algorithms. The sub-data and/or the sub-algorithms become small enough so the processing can be faster carried out. Thus, in Parallel Software Design, the term 'Parallel Software System' describes a set of interconnected software components that cooperate to carry out a defined function. Each software component can be seen as a complete system, and it is normally referred to as a 'sub-system'.

As any design activity, Coordination Design based on Architectural Patterns follow a basic procedure. Figure 6 shows the steps of such a common procedure, as a block diagram composed of steps such as scoping, analysis, synthesis, and documentation.

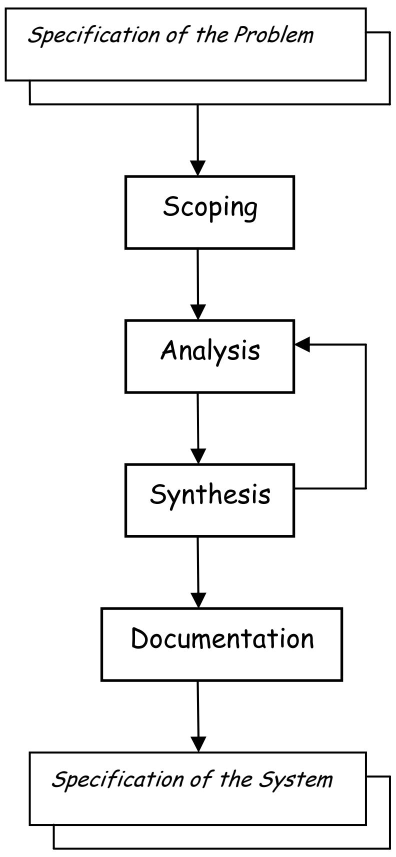


Figure 6.

The Specification of the Problem is used as input for the Coordination Design. The description about data and algorithm is used to select an Architectural Pattern that describes a potential parallel solution. The concise description included in the

Specification of the Problem is a statement of what the parallel software has to achieve. The Coordination Design using Architectural Patterns is expected to articulate the software components in enough detail that it can be actually implemented by code.
The result of the Coordination Design is a document: the *Specification of the System*, which is developed around the description provided by the Architectural Pattern. Thus, it contains a description of each software component, as well as a description of how they act simultaneously and together to work as a parallel software system. It also includes a rationale that shows how the parallel software system based on this coordination (as described by the Architectural Pattern) meets the specification of the problem.

A very first stage is to figure out even if the parallel software is necessary. If a problem has already been solved by someone else, and the parallel software is available, then the design effort is not much, mostly focusing on adjusting the parallel software to the platform at hand. In order to find a potential parallel software, it is possible to search the Web for relevant solutions. Another approach is to look for someone who has or had a similar problem, who can direct to a solution or perhaps to a vendor of a product. Nevertheless, in parallel programming, it is very unlikely to find a precise parallel software that fits to the problem at hand. Normally, a parallel software found in any of the previous ways still requires some adjustment for executing on the available platform. Moreover, it is very likely that the parallel software does not precisely meet the performance and cost requirements.

Commonly, in parallel programming, most problems have not been solved, and hence, a parallel software has to be developed. This is complex enough, so there is no obvious parallel program as a solution. In such a case, an initial parallel software is proposed, whose coordination is described in the form of an Architectural Pattern. This, effectively, divides the algorithm and/or data into a collection of well defined, coordinated software components. Later, the coordination software components are detailed and refined in terms of communication structures, as part of the next step 'Communication Design', in which some Design Patterns are proposed to act as such communication structures.

As it is shown in Figure 6, the Coordination Design step involves three main stages: scoping, synthesis, analysis, and documentation.

1. Scoping. The focus of this step is to develop the structure of a coordination as a guiding idea or principle of the parallel software system. The objective is count with a general description, which differentiates the parallel software system from its environment. Also, scoping relates with defining a concept of the parallel software system. Architectural Patterns help with his definition, by describing different types of solutions based on the division of the data, the algorithm, or both.
2. Synthesis. The coordination serves as a well-defined structure for the parallel software system. This should be described in terms of a synthesis of software components, as detailed as needed, in order to support the selected partition and allow for an analysis regarding its performance and cost properties.
3. Analysis. This step refers to determine if the parallel software system, based on the current proposed coordination, meets the performance and cost requirements as presented in the specification of the problem. The objective is to clarify if the coordination actually serves its purpose, and can be used as a base for further development towards the parallel software system.
4. Documentation. The Coordination Design process goes from synthesis to analysis, forward and backwards, until an acceptable coordination with particular properties is found. After this, the final stage in Coordination Design is to document the coordination as well as the proposed the decisions that led to it. So, it has to document the functionality of each software component and explains how these interact together. So, the Specification of the System is the output of the Coordination Design.

Refinement and elaboration of the coordination is carried out though iteratively cycle between synthesis and analysis. It is common to go through several cycles before an acceptable solution is obtained. Analysis becomes each time more and more detailed in each iterative cycle. During the first iterations, the objective is to find major deficiencies, whereas the last iterations are used to expose the performance limits. Architectural Patterns encapsulate the design experience of coordination in Parallel Software Design. They reflect the effort spent in previous parallel programs, as well as refers to the potential properties and features of the parallel software. Their objective is to easily describe the coordination of a parallel software system. Perhaps the most

important issue to keep in mind is that Architectural Patterns have to be described in such a way that the following two steps, 'Communication Design' and 'Detailed Design' can be carried out individually by a single designer or programmer.
There are some particular advises and suggestions to be taken into consideration when expressing the coordination of a parallel software system in terms of an Architectural Pattern:

- Each software component is considered as a enclosed container of code, and hence, it should be possible to implement it using a single paradigm, technology, or programming language constructions.
- Common software components are put together, so descriptions of types and classes are feasible.
- Software components should be defined so to simplify the interfaces between them. Minimal information has to be exchanged between software components.
- Always, avoid communication cycles. These are common sources of deadlock.

The Specification of the System is the final document obtained in the Coordination Design step. Such a document should serve as reference about information regarding the parallel software system, and has to be available for everyone with a stake in its development. The Specification of the System has several purposes: (a) it should allow for continue the design method into the 'Communication Design' and 'Detailed Design' steps; (b) it keeps the description of the coordination of the parallel software system, so that it can be revised and changed in response to problems found later; (c) it is the main reference to future generations of the parallel software system; and (d) it is the source of information for testing the parallel software system.

Normally, the specification of the system has the following sections:

1. The scope. This section presents the basics of operation of the parallel software system. It also includes information about the system and its surrounding environment.
2. Structure and dynamics. This section is based on the information of the Architectural Pattern used, attempting to establish the interactions among software components but in the terms of the algorithm and data at hand.

3. Functional description of software components. This section describes each software component as a participant of the Architectural Pattern, establishing its responsibilities, input and output.
4. Description of the coordination. This section describes how the coordination of the software components acts as a single entity, making the parallel software system work.
5. Coordination analysis. This section contains elements which serve to establish the advantages and disadvantages of the coordination proposed.

The specification of the system serves as an initial description of the parallel software system, and after it, the design method goes to the Communication Design step, as a refinement of the communication software components as described for the coordination used.

4.1 Architectural Patterns for Parallel Programming

The Architectural Patterns for Parallel Programming are descriptions that link a function, in terms of an algorithm and data, with a potential parallel form composed of defined software components or sub-systems connected together. Each software component has at the same time a well defined functionality. Thus, these Architectural Patterns can be considered as descriptions of well defined structures or forms in terms of the functionality of its software components or sub-systems, which simultaneously execute. The software components in these Architectural Patterns describe software components that partition the data and/or the algorithm, coordinating their activity in order to efficiently perform the function [OR98].

Architectural Patterns for Parallel Programming are used by software designers to communicate the form and structure of a parallel software [OR98]. They provide information about the problem they solve, making it a valuable piece of information for Parallel Software Design. Nevertheless, their value is not limited to communication. They are a big aid in organising ideas, as well as helping to estimate cost and effort of the parallel software development.

4.2 Classification of Architectural Patterns for Parallel Programming

The 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 [OR98].

Classification based on the order of data and operations. Requirements of order dictate the way in which a parallel process has to be performed, and therefore, impact on its Software Design. Following this, it is possible to consider that the coordination of most parallel applications fall into one of three forms of parallelism: functional parallelism, domain parallelism, and activity parallelism [CG88, Fos94, CT92, Pan96], which depend on the requirements of order of operations and data in the problem [OR98, Ort00, Ort03, Ort04, Ort05, Ort07a].

Classification based on the nature of processing elements. The nature of processing components is another classification criteria that can be used for parallel systems. Generally, components of parallel systems perform coordination and processing activities. Considering only the processing characteristic of the components, parallel systems are classified as homogenous systems and heterogeneous systems, according to the same or different processing nature of their components. This nature exposes properties that have tangible effects on their number in the system and the kind of communications among them [OR98, Ort00, Ort03, Ort04, Ort05, Ort07a].

- Homogeneous systems are based on identical components interacting in accordance with simple sets of behavioural rules. They represent instances with the same behaviour. Individually, any component can be swapped with another without noticeable change in the operation of the system. Usually, homogeneous systems have a large number of components, which communicate using data exchange operations.
- Heterogeneous systems are based on different components with specialised behavioural rules and relations. Basically, the operation of the system relies on the differences between components, and therefore, no component can be swapped with another. In general, heterogeneous systems are composed of fewer components than homogeneous systems, and communicate using function calls.

Based on these classification criteria, the five Architectural Patterns for Parallel Programming commonly used for defining the coordination organisation of parallel systems programming are classified as shown in Table 1 [OR98]:

Architectural Pattern	Type of Parallelism			Type of Processing	
	Functional	Domain	Activity	Homogeneous Processing	Heterogeneous Processing
Parallel Pipes and Filters	X				X
Parallel Layers	X			X	
Communicating Sequential Elements		X		X	
Manager Workers			X	X	
Shared Resource			X		X

Table 1.

Notice from Table 1 that there is not a pattern considered for domain parallelism and heterogeneous processing. The reason is not that there are no particular architectural patterns for such classification, but more likely, that the parallel programs that would be considered under such classification simply do not have a regular structure which could be identified by a single architectural pattern. These would require to include several organisations which solve many interesting problems, such as those commonly used in simulation. So, the present thesis focuses more precisely on those categories which can be identified to be represented as a regular structure, and described using a single architectural pattern.

4.3 Selection of Architectural Patterns

The initial selection of one or several Architectural Patterns for Parallel Programming is guided mainly by the properties used for classifying them. Based on this, a procedure for selecting an architectural pattern can be specified as follows [OR98]:

1. Analyse the design problem and obtain its specification. Analyse and specify, as precisely as possible, the problem in terms of its characteristics of order of data and computations, the probable nature of its processing components, and performance requirements. It is important to also consider the context conditions

about the chosen parallel platform and language (see step 5) that may influence the design. This stage is crucial to set up most of the basic forces to deal with during the design.

2. Select the category of parallelism. In accordance with the problem specification, select the category of parallelism —functional, domain or activity parallelism— that best describes it.
3. Select the category of the nature of the processing components. Select the nature of the process distribution —homogeneous or heterogeneous— among components that best describes the problem specification. The nature of process distribution indirectly reflects characteristics about the number of processing components and the amount and kind of communications between them in the solution.
4. Compare the problem specification with the architectural pattern's Problem section. The categories of parallelism and nature of processing components can be simply used to guide the selection of an architectural pattern. In order to verify that the selected pattern copes with the problem at hand, compare the problem specification with the Problem section of the selected pattern. More specific information and knowledge about the problem to be solved is required. Unless problems were encountered up to this point, the architectural pattern selection can be considered as completed. The design of the parallel software system continues using the selected architectural pattern's Solution section as a starting point. On the other hand, if the architectural pattern selected does not satisfactorily match aspects of the problem specification, it is possible to try to select an alternative pattern, as follows.
5. Select an alternative architectural pattern. If the selected pattern does not match the problem specification at hand, try to select another pattern that alternatively may provide a better approach when it is modified, specialised or combined with others. Checking the Examples, Known Uses and Related Patterns sections of other pattern descriptions may be helpful for this. If an alternative pattern is selected, return to the previous step to verify it copes with the problem specification.

If the previous steps do not provide a result, even after trying some alternative patterns, stop searching. The architectural patterns here do not provide a coordination organisation that can help to solve this particular problem. It is possible to look at other more general pattern languages or systems [GHJV95, POSA1, POSA2, POSA4, PLoP1, PLoP2, PLoP3, PLoP4, PLoP5] to see if they contain a pattern that can be used. Or the alternative is trying to solve the design problem without using Software Patterns.

5. Communication Design — Design Patterns

Coordination Design expresses the function of a parallel software system in terms of interacting software components which simultaneously execute, exchanging data as the processing builds up. This data exchange is performed by programmed constructions that follow one of many communication structures, which depends on the Architectural Pattern selected for the coordination, as well as the type of communication and data to be exchanged. So, some descriptions contained in the Specification of the System are required.

The information about the type of coordination structure from the Coordination Design is combined with information about the parallel hardware platform and the available parallel programming language, in order to design the communication components. These communication components along with the processing components compose the coordination of the parallel software system. Nevertheless, they have different purposes:

- Processing components are those which effectively perform a transformation or operation on data, and thus, they are developed as software components that enclose a particular function.
- Communication components allow communication between the processing components.

As Figure 7 shows, the design of the communication components (or simply, Communication Design) is the third step of the design method, following the Coordination Design. The objective of the Communication Design step is to document the descriptions of the communication components of the parallel software. Such descriptions are gathered together into a single document: the *Specification of Communication Components*. This document contains the description of the software components as sub-systems or sub-form that allow for communication and interaction between processing components. The communication components are developed mainly based on the particular characteristics available from the type of memory organisation of the hardware platform (whether shared memory or distributed memory), the available communication mechanisms of the parallel programming language, and requirements regarding the coordination organisation from the previous step. All these are proposed again as a form for the communication components that attempt to meet the requirements stated in the Specification of the Problem.

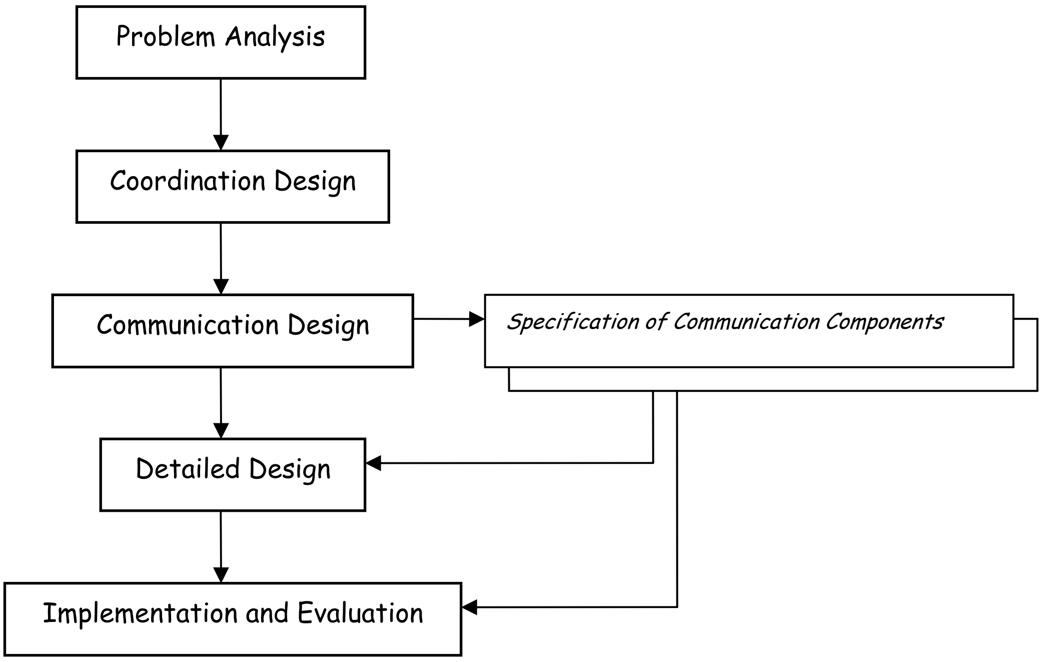

Problem Analysis
Coordination Design
Communication Design
Specification of Communication Components
Detailed Design
Implementation and Evaluation

Figure 7.

The design of the communication components involves again the activities of scoping, analysis, synthesis, refinement, and document the actual design decisions as the description of components that allow for communication between parallel software components. Such design decisions are supported at this point by the Design Patterns for Communication Components [Ort07b], as part of the Pattern-based Parallel Software Design Method . These Design Patterns specify the problem they solve as a need for communication between parallel software components which depend on (a) the selected coordination for the parallel software system under design, (b) the memory organisation of the hardware platform, and (c) the type of synchronisation proposed. Thus, these Design Patterns link these requirements with a solution, as a form describing an organisation of software components that allow for communication between parallel software components.

Therefore, Design Patterns are used here to select an organisation of software components for communicating the parallel software processing components. Hence, communicating and processing software components compose the parallel software system as a whole.

The importance of Communication Design relies on using experience about the design and implementation of communication software components. Even though they do not impact the parallel software system as a whole, they do affect the form of communication sub-systems. Several communication requirements as considered during the Coordination Design, such as communication and synchronisation between communicating components, depend on the organisation of communication components.

The communication between parallel software processing components is another important feature of parallel programming. Software designers make use of particular organisations of software components in order to achieve a certain type of communication. Separating the software components of a parallel software system into processing and communication components allows for the reuse of these communication components in other parallel software systems.

In Software Design, a 'Software System' is described as a set of interconnected software components that cooperate to carry out a defined function. Such an interconnection is carried out by communication components, defining the cooperation between software components. Each communication component is normally referred to as a 'sub-system'. At this level, Software Design describe the implementation of a communication software sub-system, which usually is developed as a set of encapsulated components that only carry out a communication functionality.
As any design activity, Communication Design based on Design Patterns for Communication Components follow the design basic procedure composed of steps such as scoping, analysis, synthesis, and documentation (Figure 8).

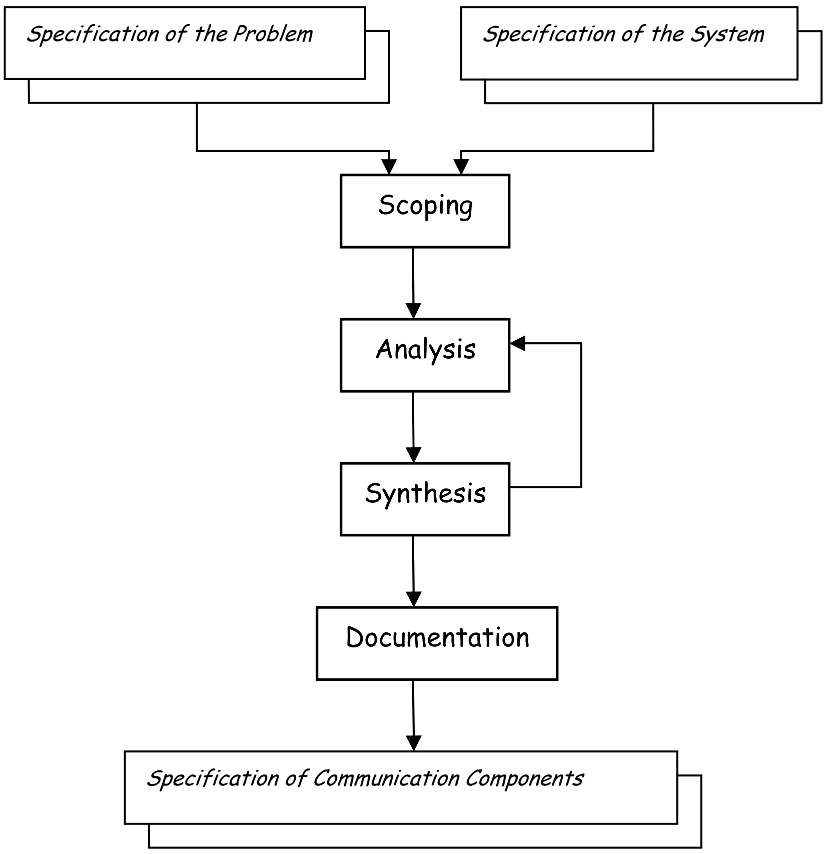

Specification of the Problem
Specification of the System
Scoping
Analysis
Synthesis
Documentation
Specification of Communication Components

Figure 8.

The Specification of the Problem and the Specification of the System are used as inputs for Communication Design. From the Specification of the Problem, the description of the parallel hardware platform, regarding its memory organisation, and the communication primitives from the parallel programming language, are taken into consideration here. This information, along with the description of the coordination selected in the Coordination Design stage, is used to select a design pattern that describes a potential communication solution. Thus, the Communication Design using design patterns is aims to articulate software components as a communication sub-system in enough detail that they can be actually implemented using code.

The outcome of the Communication Design is the *Specification of the Communication Components*, a document that describes the communication software sub-systems based on the design pattern selected. Such a document describes these sub-systems in terms of software components that allow the data exchange between parallel software components, along with the rationale about how the communication sub-systems based on this design pattern meets the communication requirements as described in the Specification of the Problem and the Specification of the System. Later, from the Specification of the Communication Components, such components are detailed and refined in terms of synchronisation mechanisms as part of the next step 'Detailed Design', in which some idioms are used to actually perform communication.

As Figure 8 shows, the Communication Design step also involves the three main stages as in any design activity: scoping, synthesis, analysis, and documentation.

1. Scoping. This step proposes an initial structure for the communication sub-system as a principle of the communication components of the parallel software system. The objective is to obtain a description of the communication sub-system. Design Patterns for Communication Components help to obtain this description, describing different types of communication structures as solutions based on the selected coordination, the memory organisation, and the synchronisation mechanisms included in the parallel programming language.
2. Synthesis. The communication sub-systems attempts to provide a well-described communication structure for the parallel software system. Such a structure is described as a synthesis of software components, supporting the selected type of communication and allowing an analysis regarding its properties.
3. Analysis. This step determines whether the communication sub-system based on the current proposed communication structure meets the requirements as needed by the coordination and presented in the specification of the problem. The objective is to check if the communication actually accomplishes its purpose, and can be used as a base for further development towards a more detailed and complete design and implementation of the parallel software system.
4. Documentation. The Communication Design goes iteratively from synthesis to analysis, until an adequate communication sub-system is found. After this, the communication design finishes by actually documenting the communication structure with all the design decisions that led to it, into the Specification of the Communication Components. This document describes the functionality of each software component, explaining their interaction to carry out the communication between parallel software components.

The Specification of the Communication Components is the document obtained from the Communication Design step, and it is added with the Specification of the System in order to provide a more detailed design for the parallel software system in development. So, the Specification of the Communication Components is expected to serve as the reference regarding the communication sub-systems, as part of the coordination of the parallel software system, and as any other specification in the method, it has to be available for everyone with a stake in its development. The Specification of the Communication Components has several objectives: (a) it should allow to go on with the next design step in the method, 'Detailed Design'; (b) it keeps the description of the communication sub-systems of the parallel software system, so that such sub-systems can be revised and changed in response to problems found later; and (c) it should help for testing the communication sub-systems of the parallel software system.

Normally, the specification of the system has the following sections:

1. The scope. This section presents the basic information about the parallel hardware platform and the programming language, as well as the selected coordination, relevant for choosing a particular communication structure.
2. Structure and dynamics. This section takes information of the Design Pattern, expressing the interaction between software components that carry out the communication between parallel software components.
3. Functional description of software components. This section describes each software component of the communication sub-system as a participant of the Design Pattern, establishing its responsibilities, input and output.
4. Description of the communication. This section describes how the communication sub-system acts as a single entity, allowing the exchange of information between parallel software components.
5. Communication analysis. This section contains issues about the advantages and disadvantages of the communication structure proposed.

The Specification of the Communication Components is a description of the communication sub-systems of the parallel software system. Following the design method, the next step is the Detailed Design step, as a refinement of the communication software components with synchronisation mechanisms.

5.1. Design Patterns for Communication Components

The Design Patterns for Communication Components are descriptions that link a communication functionality with a potential software sub-system form for communication software components which connect and communicate parallel software components. Each communication software component has a well defined functionality within the communication. Thus, these Design Patterns can be seen as descriptions of well defined structures or forms for communication sub-systems, which connect parallel software components that simultaneously execute. The software components in every Design Pattern describe a form in which components allow for a type of communication or data exchange, coordinating the activity between parallel software components.

Design Patterns for Communication Components are used by software designers to describe the form and structure of communication software components. Each Design Pattern provides information about the communication it allows, making it a valuable piece of information for Parallel Software Design.

The application of the Design Patterns for Communication Components directly depends on the Architectural Pattern for Parallel Programming which they are part of, detailing a communication and synchronisation function as a local problem, and providing a form as a local solution of software components for such a communication problem.

5.2. Classification of Design Patterns for Communication Components

The Design Patterns for Communication Components are classified taking into consideration several characteristics of the communication they perform, as well as contextual features. Hence, these Design Patterns are defined and classified according to:

- **The Architectural Pattern of the overall parallel software system.** The communication components have to be designed to allow communications in parallel systems based on an Architectural Pattern such as Parallel Pipes and Filters [OR98, Ort05], Parallel Layers [OR98, Ort07a], Communicating Sequential Elements [OR98, Ort00], Manager-Workers [OR98, Ort04], or Shared Resource [OR98, Ort03]. The type of parallelism used in the overall parallel software system is an important contextual indicator of the type of communication component to be designed.

- **The memory organisation of the parallel hardware platform.** The communication components are designed and implemented through programming mechanisms that cope with a parallel hardware platform with (a) shared memory, or (b) distributed memory [And91, Har98, And00]. The type of memory organisation is an indicator of the kind of programming mechanisms to be used when designing and implementing communication components.
- **The type of synchronisation.** Depending on the memory organisation, communication components are implemented through programming mechanisms that involve (a) synchronous communications, or (b) asynchronous communications.

Based on this classification criteria, Table 2 presents the Design Patterns for Communication Components, classified regarding the parallelism of the overall parallel software system, the memory organisation of the parallel hardware platform, and the type of synchronisation used for their implementation [Ort07b].

Design Pattern	**Architectural Pattern**	**Type of Synchronisation**		**Memory Organisation**	
		Sync.	Async.	Shared Memory	Distributed Memory
Shared Variable Pipe pattern	Parallel Pipes and Filters		X	X	
Multiple Local Call pattern	Parallel Layers	X		X	
Message Passing Pipe pattern	Parallel Pipes and Filters		X		X
Multiple Remote Call pattern	Parallel Layers	X			X
Shared Variable Channel pattern	Communicating Sequential Elements		X	X	
Message Passing Channel pattern	Communicating Sequential Elements		X		X
Local Rendezvous pattern	Manager-Workers or Shared Resource	X		X	
Remote	Manager-	X			X

Rendezvous pattern	Workers or Shared Resource				

Table 2.

5.3. Selection of Design Patterns for Communication Components

The selection of one or several Design Patterns for Communication Components is mainly guided by the classification schema. Based on this, a procedure for selecting a design pattern can be specified as follows:

1. From the Architectural Pattern to be refined and detailed, select the Design Patterns which provide communication components that allow the coordination as described by the Architectural Pattern, and check the kind of communication that best checks it.
2. Based on the memory organisation of the parallel hardware platform to be used, select the nature of the communicating components for such memory organisation —shared variable or message passing. The nature of the communicating components directly impact on the way in which the processing components are communicated, as well as the amount and kind of communications between them in the solution.
3. Select the type of synchronisation required for the communication. Normally, synchronous and asynchronous communications are available for most applications. Nevertheless, the type of synchronisation could be a difficult issue to deal with during implementation, particularly if it does not allow a flexibility when coordinating the activities within the Architectural Pattern used. Depending on the kind of coordination developed, failure in the type of synchronisation available may cause from delays in communication to complete deadlock of the whole application.
4. Once a Design Pattern is selected as a potential solution, compare the communication specification with its Context and Problem sections. Unless any problem has been found up to now, the design pattern selection can be considered as completed. The design of the parallel software system continues using the selected Design Pattern’s Solution section as a starting point for communication design and implementation. On the other hand, if the Design Pattern selected does not satisfactorily match aspects of the communication specification, it is possible to try to select an alternative design pattern, as follows.

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

If after attempting a few times with the previous steps do not yield a simple result, even trying some alternative design patterns, perhaps it is time to stop searching. The design patterns presented here most likely do not provide a communication structure that can help to solve this particular communication problem. It is possible to search other more general pattern languages or systems [GHJV95, POSA1, POSA2, POSA4, PLoP1, PLoP2, PLoP3, PLoP4, PLoP5] to see if they contain a pattern that can be used. Or the alternative is trying to solve the communication problem without using Design Patterns.

6. Detailed Design — Idioms

After the Communication Design step, the next step in the Pattern-based Parallel Software Design Method is the Detailed Design. This refers to take the characteristics described in the Specification of the Problem, the Specification of the System, and the Specification of the Communication Components, and generate the code for the required synchronisation mechanisms, depending on the parallel programming language at hand. Hence, this step is called Detailed Design since the communication sub-systems (described in the Communication Design step) of the coordination (as defined in the Coordination Design step) are actually structured into synchronisation and communication mechanisms of a real parallel programming language, such as semaphores, critical regions, monitors, message passing, or remote procedure calls.
Figure 9 shows the relation of the Detailed Design step with the other steps of the design method. As within the other design steps, the Detailed Design has the objective of producing a document which describes how the communication sub-systems within the coordination are designed and implemented using the primitives of the programming language at hand, this is, the initial code that allows for the information exchange between parallel software components of the parallel software system. Thus, such a code is made

part, along with the Specification of the Problem, Specification of the System, and the Specification of the Communication Components, to compose a single full document that describes the whole parallel system architecture: the *Parallel Software System Description*. This document is expected to fully describe the parallel software system at the three different levels of design: coordination, communication, and synchronisation mechanisms, or, in Software Pattern terms, Architectural Patterns, Design Patterns, and Idioms. Notice that the relation between the solution structures proposed at each level of design have a strict 'contains in' relation with the structures in levels above it. The Parallel Software System Description is a description of the whole parallel software system, composed of different levels of design and abstraction, as well as the forms (structures) and functionalities (dynamics) of the software components that constitute it, and how these are gathered together so the parallel software system acts as a whole, complete entity. Also, it should consider how the parallel software system meets its requirements, as stated in the Specification of the Problem document.

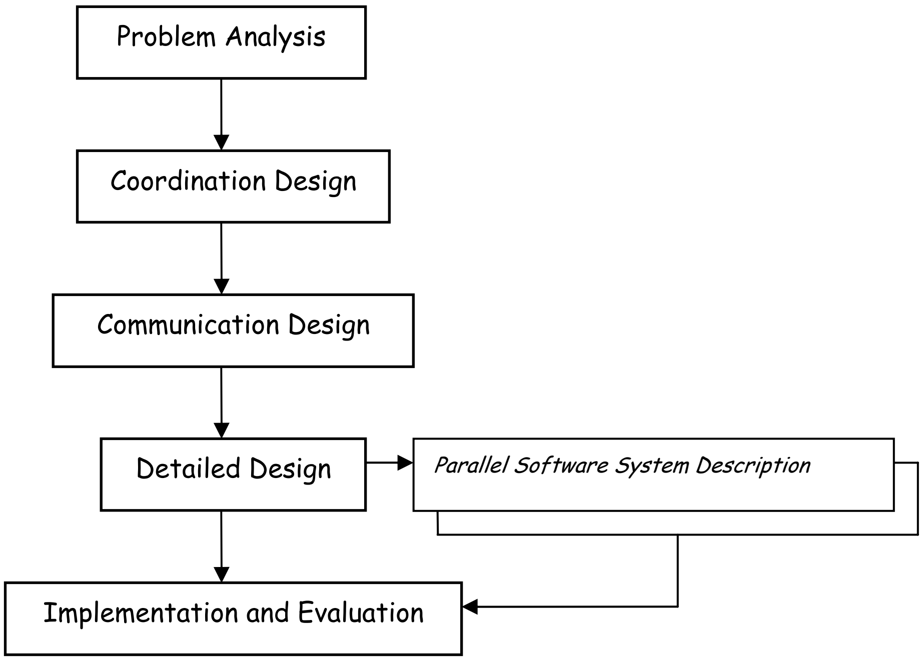


Figure 9.

Designing at the level of coding the synchronisation mechanisms again is based on the general design, involving about scoping, analysis, synthesis, refinement, and documenting

(Figure 10). Moreover, in this pattern-based approach to Parallel Software Design, idioms are proposed as the low level patterns used here.

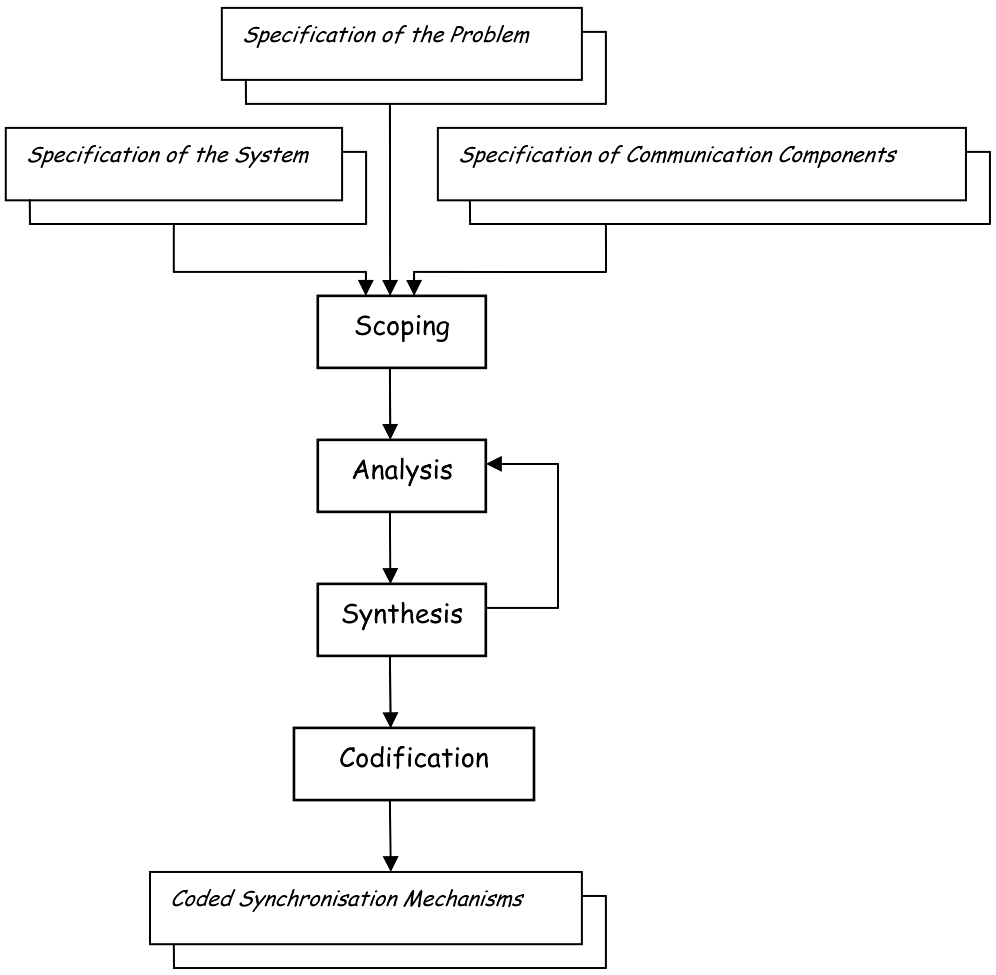

Specification of the Problem
Specification of the System
Specification of Communication Components
Scoping
Analysis
Synthesis
Codification
Coded Synchronisation Mechanisms

Figure 10.

The Detailed Design step takes as input the Specification of the Problem, the Specification of the System, and the Specification of the Communication Components, since this step requires a lot of information of the design decisions taken in all previous steps, so these are taken into consideration when coding the synchronisation mechanisms. These are the programming elements that actually perform the communication and coordination between parallel software components. At this step, the Specification of the Problem provides information about the parallel hardware platform and the programming language, which determines several characteristics of the synchronisation mechanisms to be used. The Specification of the System provides a description about how the parallel software components coordinate in order to perform the processing as a whole, and thus, provide information about what should be expected from the synchronisation mechanisms. Finally, the Specification of the Communication Components provide the programming place where the synchronisation components are actually used. As it can be observed, all this design effort focuses on creating a coordination for the Parallel Software System.

The result of the Detailed Design is not properly a document, but a coded specification of the synchronisation mechanisms. This coded specification is added along with the rest of the documents into a single design document, in order to provide a complete description of the parallel software system. Such a document contains a description of the parallel software system, as well as descriptions at several levels of detail about the coordination, communication, and synchronisation used in this design. It also provide how all components acts simultaneously and together to coordinate the simultaneous execution of the parallel software system.

Commonly, the type of low level design and implementation presented in the Detailed Design stage is covered in most parallel programming publications, and as such, they are the primary source to search for idioms. As it is shown in Figure 10, the detailed design step involves scoping, synthesis, analysis, and codification.

1. Scoping. The focus of this step in the Detailed Design is to define the environment in which the synchronisation mechanisms are used within the communication components or sub-systems. Idioms help with this definition, by describing different types of coded solutions.
2. Synthesis. An initial code for the synchronisation mechanisms coordination serves as basic programming structure for the communication and synchronisation between parallel software components. This should describe a few lines of code which actually perform the communication, ant it should be as detailed as needed, in order to allow for an analysis regarding its functionality and some of its properties.
3. Analysis. The objective here is to check if the communication is actually and adequately performed using the provided code, helping to detect and correct problems in the code. This seems something easy, but if in the final implementation of the communication components, these do not act as expected, a lot of problems whose source is unknown may arise. So, the code for communication components

should be tested, in order to understand as clearly as possible how it allows for the communication.

4. Codification. Just as every design process, the Detailed Design iterates through synthesis and analysis until an acceptable code with a defined communication and synchronisation functionality is found. This code is kept as part of the design of the parallel software system, as well as part of the system itself. This is the first coding within the Pattern-based Method, and it is aimed to carry out coordination activities.

The Parallel Software System Description is the actual result of the design effort, but it is still incomplete. In order to finish the method and obtain a working parallel software system, it is needed to still provide an implementation for the processing components of the parallel software system. Nevertheless, note that parallel design issues have been addressed here, and the design and implementation of processing software components can be carried out using concepts and techniques from sequential programming, as described in the following step, Implementation and Evaluation.

6.1. Some Idioms for Synchronisation Mechanisms

Idioms for Synchronisation Mechanisms specify how synchronisation mechanisms are used within a piece of code (and hence, they are dependent of the programming language used) serving as a coded solution to a particular problem of representing a synchronisation mechanism in a particular programming language. Thus, Idioms for Synchronisation Mechanisms are used here to deal with implementation of the communication sub-systems.

Idioms for Synchronisation Mechanisms link a well defined function with a piece of actual code in a parallel programming language that carries out such function. Moreover, this function is generally used over and over through the code of a parallel program. Thus, these idioms represent descriptions of well defined code structures or forms in terms of the functionality they capture. They are used as the basic blocks to implement the actual communication software components.

Idioms for Synchronisation Mechanisms can be used by software designers in parallel programming in order to communicate a coded form or structure of synchronisation mechanisms within a parallel program. Hence, idioms are the basic element for programming activities, making them important pieces of information for Parallel Software Design.

Here, just some Idioms for Synchronisation Mechanisms are presented. This is so since there is a large amount of related patterns and previous work on the issues presented here. Nevertheless, the objective here is to present these idioms within the context of adding synchronisation mechanisms to the communication components, which act as part of a larger coordination for a parallel software system. As such, the idioms here provide a coded form to the synchronisation mechanisms, as local implementation solutions for within a communication assembly.

6.2. Classification of Some Idioms for Synchronisation Mechanisms

The Idioms for Synchronisation Mechanisms presented here are classified taking into consideration the memory organisation, as well as the type of communication:

- **The memory organisation.** The idioms can be used within a memory organisation with shared or distributed memory. This use implies that idioms can be classified regarding their use to communicate between components using *(a)* shared variable or *(b)* message passing and remote procedure call [And91, Har98, And00]. These two ways of communication indicate the kind of programming synchronisation mechanisms to be used when designing and implementing communication components.
- **The type of communication.** Idioms are able to implement synchronisation mechanisms between components regarding two types of communication: *(a)* data exchange, or *(b)* function call. Data exchange implies that there is an actual pass of data from one component to another. Function call implies that a component invokes a function within another component. Function call can be used to implement data exchange.

Based on these two characteristics as classification criteria, Table 4.3 presents these synchronisation mechanisms, classified regarding the memory organisation and the type of communication.

Idiom	Type of Communication	Communication Mechanism

	Data Exchange	Function Call	Shared Variable	Message Passing
Semaphore idiom	X		X	
Critical Region idiom		X	X	
Monitor idiom		X	X	
Message Passing idiom	X			X
Remote Procedure Call idiom		X		X

Tabla 3.

When used within the Design Patterns for Communication Components, one or several of these idioms can be applied, in order to achieve the synchronisation features as required by the communication components.

6.3. Selection of Some Idioms for Synchronisation Mechanisms

The selection of one or several idioms for synchronisation mechanisms is guided mainly by the classification schema explained before. Based on this, a simple procedure for selecting an idiom can be considered as follows:

1) From the Design Patterns for Communication Components to be refined and detailed using synchronisation mechanisms, select the idiom which provide the synchronisation as required by the communication sub-system as described by the Design Pattern, and check if the synchronisation that best fits in it.
2) Based on the memory organisation of the parallel hardware platform to be used —shared memory or distributed memory—, select the type of synchronisation mechanism for such memory organisation —shared variable or message passing. The memory organisation directly impacts on the use of the synchronisation mechanisms by which the processing components synchronise their actions, as well as the amount and kind of communications between them in the solution.
3) Select the type of synchronisation required for the communication, and verify the way in which it is available in the parallel programming language to be used in the implementation. Normally, most programming languages include communication primitives which allow synchronous and asynchronous communications. The type of

synchronisation could represent a difficult issue to deal with during implementation, particularly if it has particularities of its implementation within the programming language which can only be noticed when executing communication activities. Thus, depending on the implementation of communication primitives, failure in understanding how they synchronise may cause from poor communication to deadlock of the whole parallel application.

4) After checking the previous steps, compare the synchronisation specification with the Context and Problem sections of the potential idiom to select. Unless a problem has arisen by now, the selection of idioms can be considered finished. The design of the parallel software system continues using the Solution section of the selected idiom for designing and implementing the synchronisation mechanisms in the programming language available. However, if the selected idiom does not satisfactorily match the synchronisation specification, try to select an alternative idiom, as described in the following steps.
5) Select an alternative idiom. If the selected idiom does not match the synchronisation specification, look for one or more idioms that alternatively may provide a better approach when modified, specialised or combined with other idioms. So, it is possible to review other sections of the idioms, such as Examples, Known Uses and Related Patterns, which may help with the synchronisation problem at hand. Normally, an alternative idiom can be selected. Therefore, it is possible to return to the previous step in order to verify if it copes with the synchronisation specification.

If after attempting with the previous steps do not yield a simple result, even trying some alternative idioms, it is very likely that the idioms here do not cover a synchronisation mechanism for the particular problem here. So, consider searching in other more general pattern languages or systems [GHJV95, POSA1, POSA2, POSA4, PLoP1, PLoP2, PLoP3, PLoP4, PLoP5] for a pattern that can be used here. Or the alternative is solving the synchronisation problem without using Software Patterns.

7. Implementation and Evaluation

The final step of the Pattern-based Parallel Software Design Method is the Implementation and Evaluation step. In this stage, all decisions regarding parallel execution and communication have been solved in previous steps, so now it is time for implementing the processing software components which actually carry out the computations and simultaneously execute. The processing components are actually inserted into the coordination structure. This structure is composed of communication software components, which are implemented using synchronisation mechanisms.

In this step, implementation means building and including the actual sequential code within the parallel software components, as described in the Parallel Software System Description, which entails the Specification of the Problem, the Specification of the System, the Specification of the Communication Components, and the coding for the synchronisation mechanisms. Also, as an integral part of this step, an evaluation of the Parallel Software System is proposed in order to test if such a system actually performs as required. In fact, the evaluation normally begins with testing and recording the performance of the different components of the Parallel Software System. Starting with the coordination software components in order to test if parallel execution and communication are carried out as outlined, and adding later the processing software components to test if such implementations really produce reasonable results. Once, the software components have been evaluated, the Parallel Software System is tested against the requirements proposed in the Specification of the Problem. It is expected that, if such requirements are accomplished by the actual properties of the Parallel Software System, then the whole development task involving design and implementation of a working Parallel Software System is done (Figure 11). In any other case, the method proposed here allows to back-track through the documentation of the design decisions taken, and correct or improve the design and implementation. Software Patterns, and specifically Design Patterns and Idioms , developed elsewhere in the Pattern Community literature, can be used at this stage in order to design and implement the processing software components [GHJV95, PLoP1, PLoP2, PLoP3, PLoP4, PLoP5, POSA1, POSA2, POSA4].

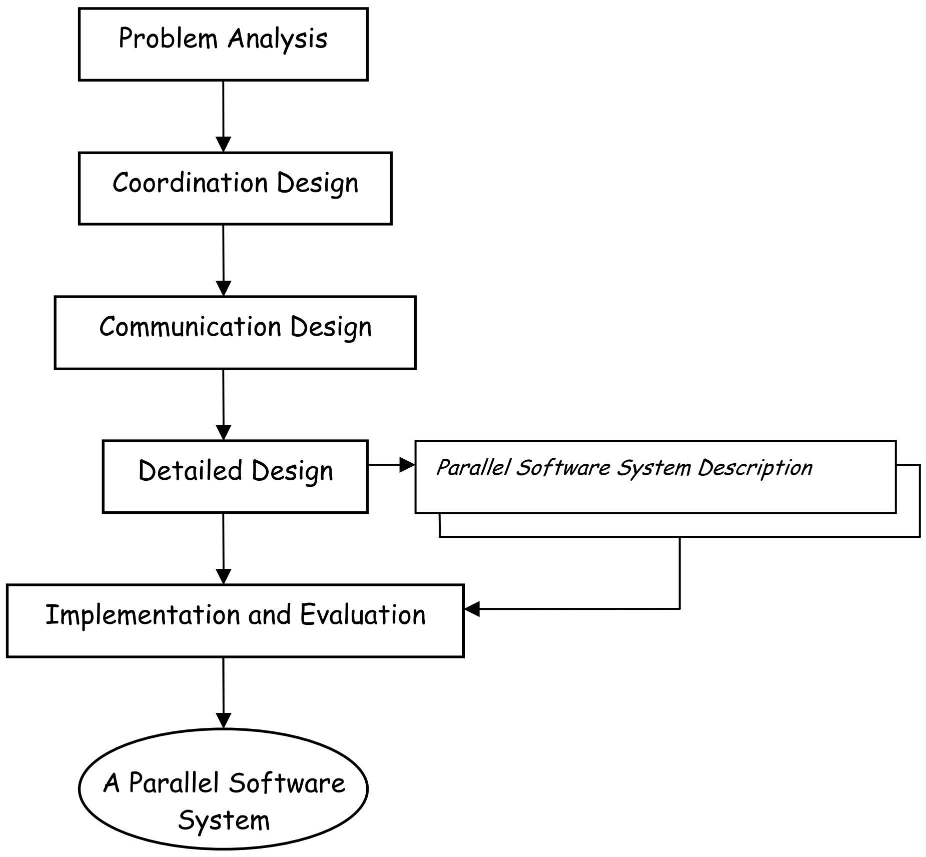

Problem Analysis
Coordination Design
Communication Design
Detailed Design
Parallel Software System Description
Implementation and Evaluation
A Parallel Software System

Figure 11.

Figure 12 shows the activities required to take the Parallel Software System Description, and turn it into a working Parallel Software System by implementing its sequential parts of code. Notice that this stage requires that the implementation of each processing software component, and when these are inserted into the coordination structure, then the output is a resulting Parallel Software System. The final activity is involved with evaluating such a system, normally against performance requirements. The evaluation is composed by a series of tests, and these tests determine whether or not the Parallel Software System and its design are acceptable.

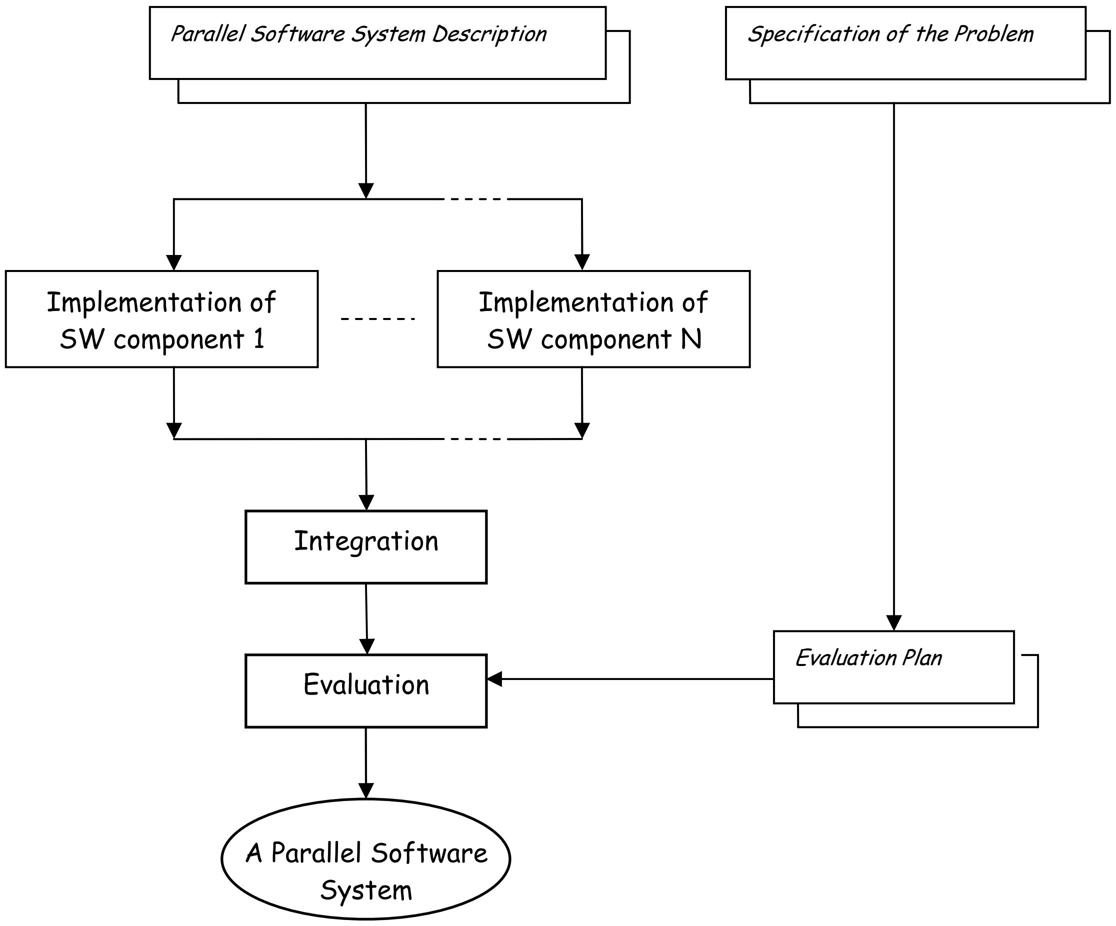

Parallel Software System Description
Specification of the Problem
Implementation of SW component 1
Implementation of SW component N
Integration
Evaluation
Evaluation Plan
A Parallel Software System

Figure 12.

Due to the importance of evaluation, normally it is necessary to develop an *Evaluation Plan* in advance, from the Specification of the Problem. In order to find any incorrect interpretation in the Specification of the Problem, in the best case the Evaluation Plan is commonly developed by someone outside the software design team. Nevertheless, if the Specification of the Problem has been thoroughly developed and taken into consideration through the design and implementation, it should be not so difficult to prepare a series of tests which verify if the Parallel Software System is ready or not.

8. Summary

This chapter introduces a Pattern-based Parallel Software Design Method, which attempts to serve as a guidance to follow through the development of a complete parallel software applications, commencing with the need for high-performance from a Problem Description, and finishing with a complete Parallel Software System. This is the framework in which all the design (and sometimes some implementation) is contained, in the form of several Software Patterns that are used for the design and implementation of the coordination.