CHAPTER 6

PARALLEL DESIGN PATTERNS

Design patterns have been introduced in the ’90 to “describe simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time. Hence, they aren’t the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form” [47].

Parallel design patterns [66] have been introduced in early ’00, with the intent to model solutions to recurring parallel problems. They represent a software engineering perspective on the parallel computing problems. The way they are presented—grouped in layers, with each layer including patterns of interest when looking at the parallel programming problems from a particular abstraction level—represents a kind of methodology suitable to support the parallel application developer in the design of efficient applications.

From our structured parallel programming course perspective, parallel design patterns represent a kind of “engineered way” of considering the problems related to parallelism exploitation. This is the reason why we introduce and shortly discuss parallel design patterns here.

6.1 DESIGN PATTERNS

A design pattern is a representation of a common programming problem along with a tested, efficient solution for that problem [47]. Design patterns are usually presented in an object oriented programming framework, although the idea is completely general and it can be applied to different programming models.

A design pattern includes different elements. The more important are: a name, denoting the pattern, a description of the problem that pattern aims to solve, and a description of the solution. Other items such as the typical usage of the pattern, the consequences achieved by
using the pattern, and the *motivations* for the pattern are also included. Table 6.1 details the elements included in the description of a design pattern in [47].

Therefore a design pattern, *per se*, is not a programming construct, as it happens in case of the algorithmic skeletons. Rather, design patterns can be viewed as “recipes” that can be used to achieve efficient solutions to common programming problems.

Typical design patterns include patterns commonly used in wide range of object-oriented applications, such as

- the *Proxy* pattern, providing local access to remote services (Fig. 6.1 shortly outlines the proxy pattern as described in [47]),
- the *Iterator* pattern, providing (sequential) access to the items of a collection,

as well as more sophisticated patterns, such as

- the *Mediator* pattern, encapsulating the interactions happening between a set of *colleague* objects,
- the *Strategy* pattern, providing encapsulation of a set of similar services along with methods to transparently pick up the best service provider,
- the *Observer* (or *Publish/Subscribe*) pattern, used to structure asynchronous, event based applications,
- the *Facade* pattern, providing a simplified interface to some kind of large code/class ensemble.

Overall, the programmer using design patterns to structure the sequential skeleton of his/her application will produce more robust and maintainable code than the programmer using a more traditional–non design pattern based–object oriented approach.

In the original work discussed in Gamma’s book [47], a relatively small number of patterns is discussed. The debate is open on the effectiveness of adding more and more specialized patterns to the design pattern set [1]. This situation closely resembles the one found in the skeleton community some years ago. Skeleton designers recognized that a small set of skeletons may be used to model a wide range of parallel applications [21, 41] but some

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name &amp; classification</td>
<td>The name of the pattern and the class the pattern belongs to</td>
</tr>
<tr>
<td>Intent</td>
<td>The problem solved by the pattern</td>
</tr>
<tr>
<td>Also known as</td>
<td>The other names/terms used to describe the pattern</td>
</tr>
<tr>
<td>Motivation (forces)</td>
<td>The reason(s) why the pattern is introduced</td>
</tr>
<tr>
<td>Applicability</td>
<td>The typical situations where the pattern may be exploited</td>
</tr>
<tr>
<td>Structure</td>
<td>The structure of the pattern in terms of the component classes/class diagrams</td>
</tr>
<tr>
<td>Participants</td>
<td>The classes involved</td>
</tr>
<tr>
<td>Collaboration</td>
<td>The interactions among the involved classes</td>
</tr>
<tr>
<td>Consequences</td>
<td>The kind of consequences deriving from pattern application</td>
</tr>
<tr>
<td>Implementation</td>
<td>Known implementation</td>
</tr>
<tr>
<td>Sample code</td>
<td>Sample code or pseudo code relative to the implementation and usage of the patterns</td>
</tr>
<tr>
<td>Known uses</td>
<td>Examples of applications/algorithms/kernels using the pattern</td>
</tr>
<tr>
<td>Related patterns</td>
<td>Other patterns that may be of interest for or used with the current pattern</td>
</tr>
</tbody>
</table>

Table 6.1 Elements of a design pattern
Parallel design patterns have been introduced [66] with the same aims of (sequential) design patterns, namely to describe solutions to recurrent problems in the context of interest. The context of interest in this case if obviously parallel software design rather than simple object-oriented software design.

As it happened in the case of classical design patterns, a parallel design pattern is described by means of a precise set of elements, namely:

- **Problem** – the problem to be solved.
- **Context** – the context where the pattern may be most suitably applied.
- **Forces** – the different features influencing the parallel pattern design.
- **Solution** – a description of one or more possible solutions to the problem solved by the pattern.

Sample parallel design patterns, as an example, solve typical parallelism exploitation problems, such as *divide & conquer*, *pipeline* and *master/worker* solving respectively the problem of efficient implementation of recursive, divide and conquer computations, of staged computations and of computations split in a number of independent tasks.
Parallel design patterns have been used to implement structured parallel programming frameworks very similar to the algorithmic skeleton ones [63, 48, 67] and a notable research activity has been generated that eventually led to the recognition of the fact that parallel design patterns may be a key point in the solution of the problems related to the development of efficient parallel software. The Berkeley report [20] published in late '00s clearly states that

Industry needs help from the research community to succeed in its recent dramatic shift to parallel computing.

and lists among the other principles to be followed to efficiently address the problem of designing and developing efficient software for parallel machines the following principle:

Architecting parallel software with design patterns, not just parallel programming languages.

In particular, the authors recognize that

Computer scientists are trained to think in well-defined formalisms. Pattern languages encourage a less formal, more associative way of thinking about a problem. A pattern language does not impose a rigid methodology; rather, it fosters creative problem solving by providing a common vocabulary to capture the problems encountered during design and identify potential solutions from among families of proven designs.

A comprehensive list of parallel design pattern papers and projects may be found at the web page http://www.cs.uiuc.edu/homes/snir/PPP/. The rest of this Section summarizes the parallel design patterns as presented in [66].

### 6.2.1 The parallel design pattern design spaces

One of the most interesting aspects related to parallel design patterns as described in [66] is the partitioning of the design space of a parallel application in four separate but related design spaces:

- The **Finding concurrency** design space, including the parallel design patterns modelling the different kinds of parallel/concurrent activities structures that may be suitably used in a parallel application.
- The **Algorithm structure** design space, including the parallel design patterns modelling recurring parallel algorithm patterns.
- The **Supporting structure** design space, including the parallel design pattern that can be used to implement the algorithmic patterns in the previous design space.
- The **Implementation mechanisms** design space, eventually including the patterns modelling the base mechanisms used to implement a parallel application.

From our algorithmic skeleton perspective, the first two design spaces correspond to what we called “concurrent activity graph” definition/design, while the last two correspond to the “implementation” of the concurrent activity graph. The importance of this design space structuring relies in the “separation of concerns” implicitly assumed in between the programmers using the parallel patterns: application programmers—the ones supposedly using the patterns in the finding concurrency and algorithm structure design spaces—will only use their (application) domain specific knowledge to find out the most proper combination of high level patterns modelling parallelism exploitation in the application at hand. And this process will not require any knowledge relative to the underlying target architecture. Then the system programmers—the ones that are experts in the target architecture features—may exploit the design patterns in the supporting structure and implementation mechanisms.
spaces to properly implement the patterns used at the higher level by the application programmers. This reflects the situation in the algorithmic skeleton world where application users basically use pre-defined skeletons that the system programmers are in charge to implement efficiently onto the target architecture.

In the following sections, we will discuss in detail the different patterns in the different design spaces, pointing out, if the case, similarities and differences with similar concepts/abstractions in the algorithmic skeleton universe. Fig. 6.2 shows the global design space, outlining the different patterns and pattern groups in the four design spaces.

### 6.2.2 Finding concurrency design space

The patterns in the “finding concurrency design space” may be used to start designing a parallel application, right after having considered the problem to be solved with the application and identified the most computationally intensive parts to be parallelized.

There are three different groups of patterns in this space. Two of them are actually related to the ways used to implement the parallel application (the decomposition and dependency analysis patterns) and one related to the fact the “finding concurrency” space is such that a solution if often found iterating with different steps through the decomposition and dependency analysis patterns (the design evaluation pattern).

In particular, the Decomposition patterns are used to decompose the problem into pieces that can execute concurrently, while the Dependency analysis patterns help grouping the tasks to be executed and analyzing the dependencies among these tasks. The Design evaluation pattern is only used to “guide the algorithm designer through an analysis of what has been done so far before moving to the patterns in the algorithm structure design space” [66].

The decomposition patterns include just two patterns:

- the Task decomposition pattern, modelling the division of a complex application in a set of tasks that can be executed concurrently, and
- the Data decomposition pattern, modelling the division of the input data into subsets that can be (relatively) independently computed to obtain the final result.

The main forces influencing the design of these patterns are flexibility, efficiency and simplicity. Flexibility is needed to adapt the application design to different implementation requirements. Efficiency is needed as we are targeting efficient parallel application development. Last but not least, simplicity is required to help the application programmers to use to pattern in their parallel applications: too complicated patterns (or libraries, objects or APIs) have no sufficient appeal to convince the application programmers they are worth being used and exploited.

The dependency analysis patterns include three different patterns, instead:

- the Group tasks pattern aimed at modelling the more convenient grouping of tasks such that the management of dependencies is simplified,
- the Order tasks pattern aimed at figuring out how tasks (or groups of tasks) may be ordered to satisfy the application constraints related to task execution, and
- the Data sharing pattern aims at modelling the accesses to a shared data structure.

E.G. Let’s have a look in more detail to one of these patterns, which are not as obvious as the ones in the decomposition patterns set: the order task pattern. The order task pattern considers groups of tasks, as output by the group tasks pattern and tries to find the dependencies among those groups of tasks in order to determine an order of execution of the groups of
Figure 6.2  Design spaces & Parallel design patterns
Figure 6.3  Algorithm structure design space: alternative pattern groups

The overall output resulting from the analysis of the “finding concurrency” design space is a decomposition of the problem into different design elements, namely i) a task decomposition identifying the tasks that can be executed concurrently, ii) a data decomposition that identifies the data local to each one of the tasks, iii) a way of grouping tasks and ordering the tasks groups in such a way temporal and data dependencies are satisfied and iv) an analysis of the dependencies among the tasks.

6.2.3  Algorithm structure design space

The output from the “finding concurrency” design space is used in the “algorithm structure design space” to refine the design of our parallel application and to figure out a parallel program structure closer to an actual parallel program suitable to be run on a parallel target architecture.

There are three major ways of organizing a parallel algorithm:

- **by task**, that is considering the tasks as the base thing to be computed in parallel within the parallel algorithm,
- **by data decomposition**, that is considering the decomposition of data into (possibly disjoint) subsets as the base items to be processed in parallel within the parallel algorithm, or
- **by flow of data**, that is considering the algorithm defined flow of data as the rule dictating what are the steps to computed concurrently/in parallel within the parallel algorithm.

The three ways of organizing the parallel algorithm are somehow alternatives. While in the “finding concurrency” design space the application programmer is supposed to go through all the groups of patterns, here the programmer is required to choose one of the three alternatives and exploit on of the parallel design patterns in the group (see Fig. 6.3).

The **Organize by tasks** pattern group includes two patterns:

- the **Task parallelism** pattern, managing the efficient execution of collections of tasks.
- The proposed solution to implement the pattern works out three different points: how
PARALLEL DESIGN PATTERNS

tasks are defined, the dependencies among tasks and the scheduling of the tasks for concurrent execution.

- the Divide&conquer pattern, implementing the well know divide and conquer recursive solution schema.

The Organize by data decomposition pattern group includes two patterns:

- the Geometric decomposition pattern modelling all those computations where a given “geometric shape” of the concurrent executors may be recognized (linear vector, bi-dimensional array), and the parallel algorithm may be defined by defining the kind of computation(s) being performed at the shape items (vector items, array rows, array columns, array items, ...).

- the Recursive data pattern, modelling those parallel computations structured after the structure of some recursively defined data structure, e.g. a list–defined as a null list or an item followed by a list–or a tree–defined as a node or a node with a (given) number of “son” trees.

Eventually, the Organize by flow of data pattern group also hosts two patterns:

- the Pipeline pattern, where the flow of data is traversing a linear chain of stages, each representing a function computed on the input data–coming from the previous stage–whose result is delivered to the next stage.

- the Event-based coordination where a number of semi-independent concurrent activities interact in an irregular way and interactions are determined by the flow of data among the concurrent activities.

E.G.D> In order to fix the concepts, let’s analyze more in detail the “recursive data” pattern. The problem solved by the pattern is the implementation of (efficient and parallel) operations over recursively defined data structures. The context includes all those situations where the data contained in a recursively defined data structure (e.g. tree or list) are processed as a whole or in pieces. Tree structured data, as an example, may be processed to obtain a new data tree with all the nodes/leaves substituted by the result of a function f applied to the old nodes/leaves. Or we may process the tree structured data to search for a particular node, traversing the tree with a pre-fix algorithm. A number of problems have to be taken into account when describing this pattern. The most notable include data representation—usually pointers are used, which are neither too much portable nor practical to use for parallel processing of all the items in the data structure, as following pointers is usually a fairly sequential process, as an example—and dimensioning of the concurrent activities used to implement recursive data processing—e.g. the dept of the tree may be unknown and not computable up to the end of the computation. The forces driving the pattern design are different. First, we have to take into account the data representation problem. We need to define a proper representation, suitable to be used for parallel processing, may be radically different from the representation used for the recursive data structure in sequential programming. As an example a list may be represented with a vector rather than with the usual linked list data structure, to ensure the possibility to perform concurrent item accesses, even if insertion and removal of elements may require more complex implementations. Second, we must ensure the data representation is not only functional to the parallel processing needs, but it is should be also easy to understand and maintain. Last but not least, a tradeoff should be looked for in between the amount of concurrency exploited in the recursive data structure processing and the amount of overhead–concurrent activity setup, communication, synchronization–introduced. The solutions adopted within the pattern include structuring the pattern computation in phases, namely:

- data decomposition: the data is restructured and decomposed as most suitable for being processed by the pattern
Figure 6.4 Recursive data pattern: sample usage (reduce of a tree)

- **structure**: figure out some kind of “collective” and parallel operation to be applied on the data decomposition results, possibly in a loop, to compute the expected results.
- **synchronization**: figure out the kind of synchronization needed to implement the (loop of) collective operations leading to the final result.

As an example, consider the problem of computing a “reduce” with an operator $\oplus$ over a binary tree. We can decompose the tree into the sub trees rooted at level $i$, then apply sequentially the “reduce” $\oplus$ operation onto all the sub trees in parallel and eventually sequentially compute the “reduce” $\oplus$ operation on the partial results computed at the different sub trees. In this case, the data “data decomposition” phase consist in obtaining the sub tree vector out of the original tree, the “structure” phase consist in applying—in parallel—the reduce over the sub trees and the “synchronization” phase involves waiting the termination of the computation of the sub tree reduces before actually starting computing the root reduce (see Fig. 6.4).

### 6.2.4 Supporting structure design space

After having explored different possibilities to find concurrency and to express parallel algorithms in the “finding concurrency” and “algorithm structure” design spaces, implementation is taken into account with two more design spaces. The first one is called **Supporting structures** design space. This space starts investigating those structures—patterns—suitable to support the implementation of the algorithms planned when exploring the “algorithm structure” design space. Two groups of patterns are included in the design space. The first one is related to program structuring approaches—this is the Program structures pattern group—while the second one is related to the commonly used shared data structures—this is the **Data structures** pattern group. The **Program structures** group includes four patterns:

- the **SPMD** pattern, modelling the computation where a single program code is run on different input data,
- the **Master/Worker** pattern, modelling concurrent execution of a *bag of tasks* on a collection of identical workers,
- the **Loop parallelism** pattern, modelling the concurrent execution of distinct loop iterations, and
Figure 6.5  Relationships between the Supporting Structures patterns and Algorithm Structure patterns (from [66]).

- the Fork/Join pattern, modelling the concurrent execution of different portions of the overall computation that proceed unrelated up to the (possibly coordinated) collective termination.

These patterns are well known in the parallel computing community. The SPMD pattern is the computational model used by MPI and one of the most popular patterns used to structure parallel computations with the master/worker. Loop parallelism has been exploited in vector architectures, and it is currently one of the main sources of parallelism in both OpenMP and GPUs. Last but not least, the fork/join pattern perfectly models the pthread_create/pthread_join model of POSIX threads.

The Data structures group includes three patterns:

- the Shared data pattern models all those aspects related to the management of data shared among a number of different concurrent activities,

- the Shared queue pattern models queue data types implemented in such a way the queues may be accessed concurrently, and

- the Distributed array pattern, models all the aspects related to the management of arrays partitioned and distributed among different concurrent activities.

Also these patterns are well known in the parallel computing community. Shared data is managed in a number of different parallel applications and the correct and efficient management of the shared data is usually the more time and effort consuming activity in the whole parallel application development/design process. Shared queues are used to support interaction of concurrent activities in different contexts, from threads to processes and concurrent activities running on CPU co-processors. Eventually, distributed arrays are often used to implement data structures that are logically shared among concurrent activities but may be somehow partitioned in such a way single one of the concurrent activities “owns and manages” a single portion of the distributed array.

The authors of [66] classify the different patterns in this design space—in fact called “supporting structures” design space—with respect to their suitability to support the implementation of the different patterns in the “algorithm structure” design space.

As an example, Task parallelism is well supported by the four patterns in the Program structures group, whereas the Recursive data pattern is only (partially) supported by the SPMD and Master/Worker pattern (see Tab. 6.5, taken from the parallel patterns book).

In fact, by considering the “supporting structures” design space as the implementation layer of the “algorithm structure” design space, we de facto make a strong analogy with what happened in the algorithmic skeletons frameworks, where skeletons represent the “algorithm structure” patterns and the implementation templates represent the “supporting...
The algorithm structure framework (corresponding to the task parallelism pattern) and the "master/worker" pattern (corresponding to one of the implementation templates used to implement task farms, maps, and other kind of skeletons) has no correspondence in the algorithm skeleton framework where often the “supporting structures” master/worker pattern is claimed to be a skeleton, that is an “algorithm structure” pattern.

We want to point out here another interesting fact. Table 6.6 reports the pages used to describe the patterns in the four design spaces in [66]. The high level patterns and the low level ones are described using more or less the same space used to describe the patterns in the “supporting structure” design space. This is a clear sign that “there is more to say” on the supporting structures, as in parallel programming this is the only design space explored in depth along with the “implementation mechanisms” space. However, this latter space is a kind of “RISC” layer only providing very basic patterns used to build the full range of “supporting structures” patterns and therefore it may be described much more concisely.

Consider the “SPMD” pattern. The problem solved with the pattern is the correct and efficient implementation of parallel computations where programmer provide a single program for all the concurrent activities. The single program may differentiate the instructions to execute depending on the concurrent activity index. The context where this pattern is used includes the large majority of the massively parallel applications, in particular all those developed using MPI. These applications are usually implemented with a single program such as the one outlined in Fig. 6.7. The program is executed on all the target machine processing elements. Each instance of the program receives a different integer id and the code executed at the different PEs is therefore differentiation with the switch statement. The forces driving the pattern design are the usual ones of most of the “supporting structures” patterns.
patterns, namely: portability, to ensure pattern reusability, scalability, to support different parallelism degrees according to the target architecture features, and efficiency, to ensure performance. A viable solution to SPMD pattern design consists in structuring the pattern with the phases outlined in the MPI code sketch above:

- Initialize: setting up all the necessary infrastructure for the pattern execution
- Unique identifier: getting the id necessary to differentiate the code within the unique program
- Run the program on all the different processing elements: run the program in such a way the all the concurrent program activities are completed
- Finalize: clean up everything and carefully terminate all the activities.

6.2.5 Implementation mechanisms design space

The second (and lower level) design space related to implementation of parallel applications is called Implementation mechanisms design space. This design space includes the patterns modelling the base mechanisms needed to support the parallel computing abstractions typical of parallel programming, that is:

- concurrent activities
- synchronizations
- communications.

In fact, the design pattern hosts only three distinct patterns:

- the UE Management pattern, related to the management of the Units of Execution (processes, threads)
- the Synchronization pattern, handling all those aspects related to ordering of events/computations in the UE used to execute the parallel application
- the Communication pattern, handling all the aspects related to the communications happening in between the different UE implementing the parallel application.

The “UE management” pattern deals with all the aspects related to the management of the concurrent activities in a parallel application. Therefore creation, run and termination of concurrent activities are taken into account. Although in [66] only threads and processes are taken into account, the “UE management” pattern may be obviously adapted to handle the concurrent activities placed on CPU co-processors, e.g. the GPU kernels. The “synchronization” pattern deals with all aspects related to synchronization of concurrent activities and therefore covers aspects such as lock/fence\textsuperscript{98} mechanisms, higher level mutual exclusion constructs (e.g. monitors) and collective synchronizations (e.g. barriers). Last but not least, the “communication” pattern deals with the aspects related to data exchange among concurrent activities and therefore covers aspects related to different kinds of point-to-point (e.g. send, receive, synchronous and asynchronous) and collective communications (e.g. broadcast, scatter, gather, reduce).

\textsuperscript{98}mechanisms ensuring that at a given point the view of the shared memory items is consistent across all EU involved in a parallel application
6.3 SAMPLE PARALLEL DESIGN PATTERN USAGE: IMAGE PROCESSING APPLICATION DESIGN

We illustrate the kind of design methodology related to parallel design patterns summarizing different steps in the design of a parallel application. We chose an application whose goal is to process--in parallel--a stream of images in such a way i) a “truecolor” filter is applied to each image—the colors are normalized—and ii) the image is sharpened—an edge enhance filter is applied.

6.3.1 Application analysis

The first step is not related to the design patterns, but rather is meant to identify the actual needs of the application in terms of parallel computing. As we want to process a (possibly long) stream of images and we know the images are high resolution images, we can simply evince that the computational load will be huge due to two distinct facts: i) the large number of images being processed, and ii) the large number of pixels relative to each one of the images. Therefore our aim will be to parallelize as much as possible both the processing of different images in the stream (which are independent of each other) and of different parts of the same image (if possible).

6.3.2 Exploring the “finding concurrency” design space

We first consider the patterns in “decomposition” pattern group. As different images appearing onto the input stream are independent, we can use the “task decomposition” pattern to model parallel execution of our application over the different images of the stream.

When moving considering the problem of parallel computation of the single image, we may consider to use the “data decomposition” pattern to implement both filters, the color normalization and the edge sharpening ones.

We then consider the “dependency analysis” pattern group. Here we may use the “order task” pattern to ensure that the processed images are output in the same order the corresponding unprocessed images appeared onto the input stream. We can also consider using the “data sharing” pattern to ensure access to shared data or portions of the images when we implement the two filters. As an example, we should consider the possibility to compute some general—i.e. referred to the whole image—color parameters and then to use this parameters to normalize independently different sub-portions of the image in the color normalization filter.

The “design evaluation” pattern may be used to evaluate the concurrency found in this step, before actually moving to explore the “algorithm structure” design space. Within the pattern, it is suggested to consider three different perspectives (forces) in order to come to a good “concurrent decomposition” of our application:

**Suitability for target platform** Although we still move at a very high level of abstraction, some general features of the target architecture should already be taken into account: i) number of processing elements (CPU/core, co-processor cores), ii) mechanisms supporting data sharing (shared memory, distributed shared memory, message passing) and iii) overhead related to the set up of concurrent activities. In particular, we must ensure

- we do not plan to use a number of concurrent activities larger than the number of available processing elements. In our case this means we should not plan to use one processing element per input image, as an example. In fact, by recognizing (task order pattern) that the edge filter is to be applied after the color one, we can plan to use a couple of processing elements to implement the processing of
a single image, and then we may assume to use more of this processing element pairs to implement in parallel the processing relative to different images of the input stream, up to the point we use all the available processing elements or we succeed processing all the available images before new ones are available onto the input stream.

- that the planned shared data structures may be actually reasonably implemented in terms of the mechanisms available to support sharing. In our case this means that we look at the target architecture features and in case it does not support primitive sharing mechanisms (e.g. shared memory or distributed shared memory) we rely only on communications to implement our shared data structures, e.g. assuming they are encapsulated in a “sharing server” answering all the read/write requests to the shared data structures.

- and that the overhead necessary to implement all the planned concurrent activities does not exceed the benefits coming from the parallel execution of our application. In our case this means we should be careful not setting up a concurrent activity when the time spent in computing the results of the concurrent activity sequentially is comparable to the time used to set up the concurrent activity itself.

**Design quality** In this case we need to ensure our concurrent activity plan ensures flexibility, efficiency and simplicity. In our case, flexibility is ensured by the possibility to vary different parameters, such as the parallelism degree in the “task decomposition” and “data decomposition” pattern, as well as in the possibility to collapse either level, that is to apply only the “task decomposition” pattern (therefore to exploit only stream parallelism) or to apply only the “data decomposition” pattern (that is to apply only data parallelism). Simplicity comes from the clean design ensured by the proper usage of the two patterns mentioned above. Efficiency comes from the considerations relative to the “suitability for target platform” and may only be stated in principle: it is well known that a good design may be completely impaired when the wrong implementation mechanisms are chosen to implement it, and therefore the actual efficiency will be clear only after having explored the whole set of design spaces.

**Preparation for the next phase** We must ensure that our concurrent activity plan is suitable to be worked out by the patterns in the “algorithm structure” design space. In our case, the regular decomposition “task+data” figured out should ensure the “algorithm structure” space may be efficiently explored.

Fig. 6.8 illustrates the results of the exploration of the “finding concurrency” design space.

### 6.3.3 Exploring the “algorithm structure” design space

Moving to the algorithm structure design space, we decide the abstract algorithms to be used to implement the concurrent activities identified in the previous step. As an example:

- we decide to use a “task parallelism” pattern from the “organize by task” group to implement the processing of different images of the input stream

- we decide to use a “geometric decomposition” pattern from the “organize by data decomposition” group to implement the processing of the single filter on the single image

- we decide to use a “pipeline” pattern from the “organize by flow of data” group to implement the sequence of the two filters applied onto one input image.
Figure 6.8  Finding concurrency design space explored: the patterns used are outlined in red/bold, the relation with the concurrent activities individuated are show as dotted lines.

Figure 6.9  Algorithm structure design space explored: the patterns used are outlined in red/bold, the relation with the concurrent activities individuated are show as dotted lines.
By picking up these patterns (see Fig. 6.9) we obviously must consider all the pattern related aspects. Remember a design pattern is a kind of “recipe” solving a particular problem. You—the application programmer—are charged of the task of instantiating the recipe to the case at hand.

6.3.4 Exploring the “supporting structures” design space

Eventually, we start considering more concrete implementation aspects by exploring the “supporting structures” design space.

As far as the “program structures” pattern group is concerned, we should look for patterns modelling the different concurrent activities individuated so far. The task parallel computation of different pictures of the input stream will be most suitably implemented using a “master/worker” pattern. Each worker will be in charge of processing a single image. The number of workers used—the parallelism degree—will determine the performance of our application. When considering the implementation of the different filter stages according to the “data decomposition” pattern, we have two choices that are worth being explored. We can use the classic “SPMD” pattern to model data parallel processing of the single filter. But we may also choose to use the “loop parallelism” pattern for the very same purpose. In case we use modern multi core architectures, the “loop parallelism” pattern may be a good choice, due to the facilities included in different programming environments (e.g. SSEx/AVX exploitation within g++ or the #pragma parallel for directive in OpenMP) to efficiently support this kind of parallelism. In case we use COW/NOW architectures, instead, the choice will be definitely to use the “SPMD” paradigm which is the one supported in MPI.

While considering the “data structures” pattern group, instead, we will probably eventually pick up the “shared queue” pattern to communicate (partially computed) images across different stages/workers, and the “distributed array” pattern to support data parallel implementation of the two distinct image filters.
6.3.5 Exploring the “implementation mechanisms” design space

Eventually, we have to explore the design space related to the implementation mechanisms. Let’s assume we are targeting a cluster of multi core processing elements running a POSIX compliant operating system. A possible choice of patterns in this design space implementing the patterns chosen in the “supporting structures” design space will be the following one:

- both threads and processes will be used in the “UE mechanisms” pattern. In particular, we may choose to use processes to implement the master/worker pattern and threads to implement the data parallel computation of the two filter stages.

- locks and mutexes from the “synchronization” patterns will be used to implement the “shared queue” and the “distributed array” patterns at the single (multi core) processing element level.

- asynchronous point-to-point communications from the “communication” pattern will be used to implement the steps needed to feed workers in the master/worker pattern processing different images and stages in the pipeline processing different filters on the same image.

6.4 COMPARING PARALLEL DESIGN PATTERNS AND ALGORITHMIC SKELETONS

After having discussed how parallel design patterns are structured and how they can be used to design and implement parallel applications, let’s try to set up a kind of “summary” of different features relative to both design patterns and to algorithmic skeletons, in the perspective of better understanding the pros and cons of both approaches. Fig. 6.11 summarizes how different features are supported in parallel design pattern and algorithmic skeleton (typical) frameworks.

The overall statement we can make out of this comparison is the following one:

Algorithmic skeletons and design patterns tackle the same problem(s) from different perspectives: high performance computing and software engineering, respectively.

Algorithmic skeletons are aimed at directly providing pre-defined, efficient building blocks for parallel applications to the application programmer, while design patterns are aimed at providing “recipes” to program those building blocks, at different levels of abstraction.

A large number of algorithmic skeleton frameworks are available, while design pattern related methodologies are used in different—apparently non structured—parallel programming frameworks.

From this summary two important considerations may be made:

- on the one side, the pattern methodology may be used to improve the design of parallel applications with algorithmic skeleton based frameworks

- on the other side, some formalization of the design pattern concepts such that the “solution” element of the pattern may be provided as one or more library entries/object classes could support the definition of a pattern/skeleton programming environment/framework merging the more evident advantages of both worlds.

It is worth pointing out that the design patterns listed in Sec. 6.2 may be extended considerably and with profit. A number of different parallel design patterns have been actually proposed and used, which are not included in [66]. Just to make an example,

99 e.g. see the ones described in the works collected at the http://www.cs.uiuc.edu/homes/snir/PPP/web site
<table>
<thead>
<tr>
<th>Algorithmic skeletons</th>
<th>Parallel design patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Communities</strong></td>
<td>High performance computing</td>
</tr>
<tr>
<td><strong>Active since</strong></td>
<td>Software engineering</td>
</tr>
<tr>
<td><strong>Style</strong></td>
<td>Declarative, library calls</td>
</tr>
<tr>
<td><strong>Abstractions</strong></td>
<td>Common, reusable, complete, nestable</td>
</tr>
<tr>
<td><strong>Level of abstraction</strong></td>
<td>Abstract parallelism/concurrency patterns, Implementation patterns</td>
</tr>
<tr>
<td><strong>Nesting</strong></td>
<td>Very high</td>
</tr>
<tr>
<td><strong>Code reuse</strong></td>
<td>Allowed, encouraged</td>
</tr>
<tr>
<td><strong>Expandability</strong></td>
<td>Poor/none, some frameworks support</td>
</tr>
<tr>
<td><strong>Programming methodology</strong></td>
<td>Instantiate and combine skeletons from a kind of “skeleton palette”</td>
</tr>
<tr>
<td><strong>Portability</strong></td>
<td>Ensured by different implementations of the skeleton framework (or different implementation template libraries)</td>
</tr>
<tr>
<td><strong>Layering</strong></td>
<td>Usage layering: two tier model with task parallel skeletons at top, and data parallel skeletons at the bottom level</td>
</tr>
<tr>
<td><strong>Maintained programming frameworks</strong></td>
<td>Multiple, C/C++, Java, Ocaml based, targeting COW/NOW, multi cores, multi cores with GPUs</td>
</tr>
<tr>
<td><strong>Notable frameworks</strong></td>
<td>P3L, ASSIST, Muesli, SkeTo, Skandium, FastFlow, SkePu, Mallba, OSL</td>
</tr>
<tr>
<td><strong>Automatic performance tuning</strong></td>
<td>Possible, demonstrated, both static (performance models and template assignment) and dynamic (behavioural skeletons)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>Assessed. Comparable with the ones achieved using classical, non structured programming environments.</td>
</tr>
<tr>
<td><strong>Time to deploy</strong></td>
<td>Greatly reduced w.r.t. classical parallel programming frameworks, both design and implementation/tuning/debugging time</td>
</tr>
<tr>
<td><strong>Debugging</strong></td>
<td>Debugging of sequential code portions (only) is needed</td>
</tr>
</tbody>
</table>

**Figure 6.11** Summary of features in parallel design patterns and algorithmic skeletons
consider the “pipeline” pattern included in “organize by flow of data” pattern group in
the “algorithm structure” design space. The “supporting structures” design space does not
provide a specific pattern suitable to be used to implement a pipeline algorithm. Although
both the “SPMD” and the “master/worker” patterns may be adapted to execute “pipeline”
stage tasks, a “pipeline” “supporting structure” pattern would be greatly useful and not
too much difficult to introduce.

Actually, different authors–including the author of these notes–recognize that only three
fundamental “supporting structures” are needed to implement any parallel application, pro-
vided a convenient number of “connector patterns” are also provided. These fundamental
“supporting structure” patterns are the “string of workers”–a number of concurrent activi-
ties processing relatively independent tasks in parallel–the “chain of workers”–a number of
concurrent activities each processing results from the previous one and delivering results to
the next one in the chain–and feedback loops–communications channels driving back some
results to previously executed concurrent activities. The “connector patterns” needed to set
up proper networks of concurrent activities out of these supporting structure patterns i.e. to
manage communication and synchronization among the concurrent activities–include point-
to-point and collective communications, lock/mutex/semaphores and barriers and logically
shared data structures.