CENG-505

Parallel Computing – I

**Design Patterns for Parallel Programming**

by

Çağlar GÜNEL

200871202

Table of Contents

[INTRODUCTION 1](#_Toc281759356)

[PATTERN LANGUAGE AND DESIGN SPACES 4](#_Toc281759357)

[Finding Concurrency Design Space 5](#_Toc281759358)

[Task Decomposition Pattern 7](#_Toc281759359)

[Data Decomposition Pattern 7](#_Toc281759360)

[Algorithm Structure Design Space 8](#_Toc281759361)

[Task Parallelism Pattern 8](#_Toc281759362)

[Divide and Conquer Pattern 9](#_Toc281759363)

[Geometric Decomposition Pattern 9](#_Toc281759364)

[Recursive Data Pattern 9](#_Toc281759365)

[Pipeline Pattern 9](#_Toc281759366)

[Event-Based Coordination Pattern 10](#_Toc281759367)

[Supporting Structures Design Space 10](#_Toc281759368)

[SPMD Pattern 10](#_Toc281759369)

[Loop Parallelism Pattern 11](#_Toc281759370)

[Fork/Join Pattern 11](#_Toc281759371)

[Shared Queue Pattern 11](#_Toc281759372)

[Implementation Mechanisms Design Space 11](#_Toc281759373)

[UE Management 11](#_Toc281759374)

[Synchronization 12](#_Toc281759375)

[Communication 12](#_Toc281759376)

[REFERENCES 12](#_Toc281759377)

# INTRODUCTION

Software developers continuously solve many problems. However, the problems that parallel software designers face are very complex. An important difference between parallel software design and other software design is that parallel problems are solved by applying very special scientiﬁc and mathematical approaches.

A **design pattern** describes a good solution to a recurring problem in a particular context. The pattern is described in format that includes the pattern name, a description of the context, the forces (goals and constraints), and the solution. The idea is to record the experience of experts in a way that can be used by others facing a similar problem. In addition to the solution itself, the name of the pattern is important and it enhances communication between designers in the same area.

Design patterns: [3]

* Provide a cookbook to systematically guide programmers
  + Decompose, Assign, Orchestrate, Map
  + Can lead to high quality solutions in some domains
* Provide common vocabulary to the programming community
  + Each pattern has a name, providing a vocabulary for discussing solutions
* Helps with software reusability and modularity
  + Written in prescribed format to allow the reader to quickly understand the solution and its context
* Otherwise, too difficult for programmers and software will not fully exploit parallel hardware.

Most parallel software design problems are very complex that designers cannot initially see the solution. To produce a parallel program, designers need to proceed step-by-step. A general design process proposed in [1] that is used for parallel software design is shown in Figure 1.

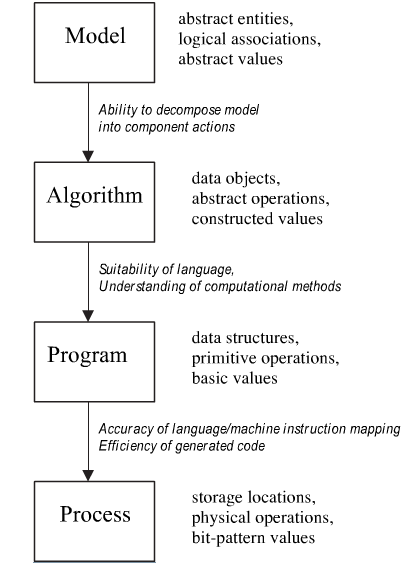


Figure 1. Describing Parallel Systems at Different Levels of Abstraction [1]

The design process can often be characterized using four levels of abstraction according to [1]. Each level deﬁnes its own collection of data and a set of operations or manipulations applicable to that data, as described below:

* **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, irrespective of the computer system on which it will execute. Descriptions are often made in natural language or diagrams. At the model level, software designers may start to notice portions of the solution that are **candidates for parallelization**. Since scientiﬁc problems require improvement in performance, software designers have to focus their attention on computationally intensive parts of the implementation.
* **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 a speciﬁc range of values. Typically, algorithmic solutions are conceived as a group of sequential steps whose descriptions are made in a notation appropriate for the model from the previous level, rather than precisely related to the computing environment in which the solution will execute. The algorithmic speciﬁcation nevertheless reﬂects the fact that the solution will be obtained 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 proceed concurrently.
* **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 formalism, but at the same time, attempts to provide expressiveness, generality and portability. This phase is often the most challenging, since software designers 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 designers are conﬁdent that the solution works.
* **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 can execute simultaneously, depending on the programming language description of the previous level.

This approach is very similar to the patterns and the four design spaces proposed in [2].

# PATTERN LANGUAGE AND DESIGN SPACES

A pattern language is something more than a catalog of patterns. In a pattern language, the patterns are organized into a structure that leads the user through the collection of patterns in such a way that complex systems can be designed using the patterns. At each decision point, the designer selects an appropriate pattern. Each pattern leads to other patterns, resulting in a final design in terms of a web of patterns. Thus, a pattern language embodies a design methodology and provides domain-specific advice to the application designer.

T.G. Mattson, et.al. [2] specifies a **pattern language** for parallel programming that provides several benefits. The immediate benefits are a way to disseminate the experience of experts by providing a catalog of

* good solutions to important problems,
* an expanded vocabulary,
* a methodology for the design of parallel programs.

This guides a developer in the process of developing a parallel program. The programmer provides an understanding of the problem to be solved, goes through the pattern language, and then obtains a detailed parallel design

The pattern language is organized into four design spaces:

* Finding Concurrency
* Algorithm Structure
* Supporting Structures
* Implementation Mechanisms

as shown in Figure 2.

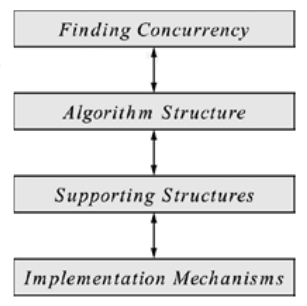


Figure . Overview of the Pattern Language [2]

The Finding Concurrency design space helps to structure the problem to **expose exploitable concurrency**. The designer working at this level focuses on high-level algorithmic issues and tries to expose potential concurrency.

The Algorithm Structure design space helps to structure the algorithm to **take advantage of potential concurrency**. That is, the designer who is working at this level tries to use the concurrency exposed in working with the Finding Concurrency patterns. The Algorithm Structure patterns describe overall strategies for exploiting concurrency.

The Supporting Structures design space represents an intermediate stage between the Algorithm Structure and Implementation Mechanisms design spaces. Two important groups of patterns in this space are those that represent program-structuring approaches and those that represent commonly used shared data structures.

The Implementation Mechanisms design space is concerned with how the patterns of the higher-level spaces are mapped into particular programming environments. We use it to provide descriptions of common mechanisms for process/thread management (for example, creating or destroying processes/threads) and process/thread interaction (for example, semaphores, barriers, or message passing). The items in this design space are **not presented as patterns** because in many cases they map directly onto elements within particular parallel programming environments. They are included in the pattern language anyway, however, to provide a complete path from problem description to code.

### Finding Concurrency Design Space

Programmers should start their design by analyzing the problem to expose exploitable concurrency. The design space in which this analysis is carried out is called the **Finding Concurrency** design space. The patterns in this design space will help identify and analyze the exploitable concurrency in a problem. After this is done, one or more patterns from the **Algorithm Structure** space can be chosen to help design the appropriate algorithm structure to exploit the identified concurrency.

An overview of this design space and its place in the pattern language is shown in Figure 3.

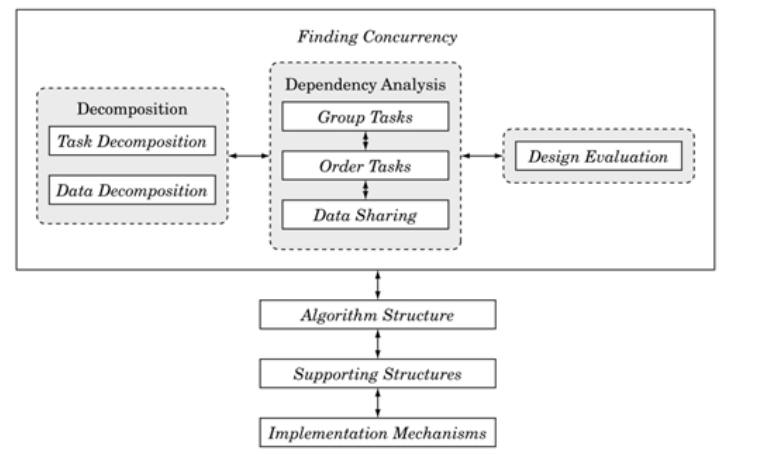


Figure 3. Finding Concurrency [2]

Before starting to work with the patterns in this design space, the algorithm designer must first consider the problem to be solved and make sure the effort to create a parallel program will be justified: Is the problem large enough and the results significant enough to justify expending effort to solve it faster? If so, the next step is to make sure the key features and data elements within the problem are well understood. Finally, the designer needs to understand which parts of the problem are most computationally intensive, because the effort to parallelize the problem should be focused on those parts.

After this analysis is complete, the patterns in the Finding Concurrency design space can be used to start designing a parallel algorithm. The patterns in this design space can be organized into three groups.

* **Decomposition Patterns**: The two decomposition patterns, Task Decomposition and Data Decomposition, are used to decompose the problem into pieces that can execute concurrently.
* **Dependency Analysis Patterns**: This group contains three patterns that help group the tasks and analyze the dependencies among them: Group Tasks, Order Tasks, and Data Sharing. Nominally, the patterns are applied in this order. In practice, however, it is often necessary to work back and forth between them, or possibly even revisit the decomposition patterns.
* **Design Evaluation Pattern**: The final pattern in this space guides the algorithm designer through an analysis of what has been done so far before moving on to the patterns in the Algorithm Structure design space. This pattern is important because it often happens that the best design is not found on the first attempt, and the earlier design flaws are identified, the easier they are to correct. In general, working through the patterns in this space is an iterative process.

The first step in designing a parallel algorithm is to decompose the problem into elements that can execute concurrently. We can think of this decomposition as occurring in two dimensions.

* The **task-decomposition dimension** views the problem as a stream of instructions that can be broken into sequences called **tasks** that can execute simultaneously. For the computation to be efficient, the operations that make up the task should be largely independent of the operations taking place inside other tasks.
* The **data-decomposition dimension** focuses on the data required by the tasks and how it can be decomposed into distinct chunks. The computation associated with the data chunks will only be efficient if the data chunks can be operated upon relatively independently.

#### Task Decomposition Pattern

**Problem**: How can a problem be decomposed into tasks that can execute concurrently?

**Context**: Every parallel algorithm design should start from a good understanding of the problem being solved. The designer must understand which are the computationally intensive parts of the problem, the key data structures, and how the data is used as the problem's solution unfolds. Fundamentally, every parallel algorithm involves a collection of tasks that can execute concurrently. The challenge is to find these tasks and craft an algorithm that lets them run concurrently.

In some cases, the problem will naturally break down into a collection of independent (or nearly independent) tasks, and it is easiest to start with a task-based decomposition. In other cases, the tasks are difficult to isolate and the decomposition of the data (as discussed in the Data Decomposition pattern) is a better starting point. It is not always clear which approach is best, and often the algorithm designer needs to consider both.

**Solution**: The key to effective task decomposition is to ensure that the tasks are sufficiently independent so that managing dependencies takes only a small fraction of the program's overall execution time. It is also important to ensure that the execution of the tasks can be evenly distributed among the ensemble of PEs (the load-balancing problem).

In a task-based decomposition, we look at the problem as a collection of distinct tasks, paying particular attention to:

* The actions that are carried out to solve the problem. (Are there enough of them to keep the processing elements on the target machines busy?)
* Whether these actions are distinct and relatively independent.

#### Data Decomposition Pattern

**Problem:** How can a problem's data be decomposed into units that can be operated on relatively independently?

**Context:** A data-based decomposition is a good starting point if the following is true:

* The most computationally intensive part of the problem is organized around the manipulation of a large data structure.
* Similar operations are being applied to different parts of the data structure, in such a way that the different parts can be operated on relatively independently.

For example, many linear algebra problems update large matrices, applying a similar set of operations to each element of the matrix. In these cases, it is straightforward to drive the parallel algorithm design by looking at how the matrix can be broken up into blocks that are updated concurrently.

**Solution:** A few common examples include the following.

* **Array-based computations**: Concurrency can be defined in terms of updates of different segments of the array. If the array is multidimensional, it can be decomposed in a variety of ways (rows, columns, or blocks of varying shapes).
* **Recursive data structures**: We can think of, for example, decomposing the parallel update of a large tree data structure by decomposing the data structure into sub trees that can be updated concurrently.

### Algorithm Structure Design Space

Our goal in the Algorithm Structure design space is to refine the design and move it closer to a program that can execute tasks concurrently by mapping the concurrency onto multiple UEs running on a parallel computer.

There are many ways to define an algorithm structure. Most follow one of six basic design patterns. These patterns make up the Algorithm Structure design space. An overview of this design space and its place in the pattern language is shown in Figure 4.

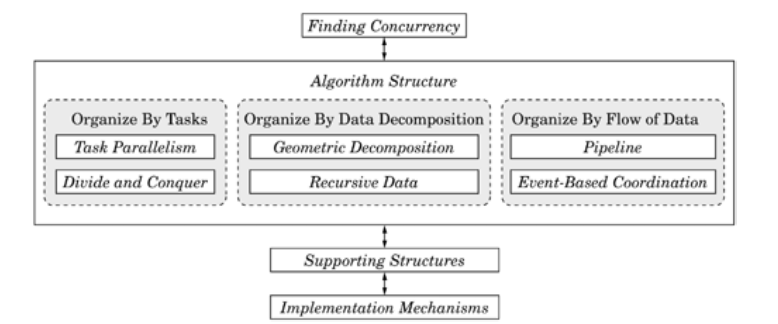


Figure 4. Algorithm Structure [2]

#### Task Parallelism Pattern

When the design is based directly on the tasks, the algorithm is said to be a task parallel algorithm. The problem can be decomposed into a collection of tasks that can execute concurrently. The tasks can be completely independent or there can be dependencies among them. In most cases, the tasks will be associated with iterations of a loop, but it is possible to associate them with larger scale program structures as well.

Most problems for which this pattern is applicable fall into the following two categories:

* **Embarrassingly parallel** problems are those in which there are no dependencies among the tasks. A wide range of problems fall into this category, ranging from rendering frames in a motion picture to statistical sampling in computational physics.
* **Replicated data or reduction** problems are those in which dependencies can be managed by "separating them from the tasks" by replicating the data at the beginning of computation and combining results when the termination condition is met.

#### Divide and Conquer Pattern

The divide-and-conquer strategy is employed in many sequential algorithms. With this strategy, a problem is solved by splitting it into a number of smaller sub problems, solving them independently, and merging the sub solutions into a solution for the whole problem. The sub problems can be solved directly, or they can in turn be solved using the same divide-and-conquer strategy, leading to an overall recursive program structure.

The potential concurrency in this strategy is not hard to see. Because the sub-problems are solved independently, their solutions can be computed concurrently.

#### Geometric Decomposition Pattern

For arrays and other linear data structures, we can often reduce the problem to potentially concurrent components by decomposing the data structure into contiguous substructures, just like dividing a geometric region into sub regions. For arrays, this decomposition is along one or more dimensions, and the resulting sub arrays are usually called blocks.

#### Recursive Data Pattern

The most challenging part of applying this pattern is restructuring the operations over a recursive data structure into a form that exposes additional concurrency.

After the concurrency has been exposed, it is not always the case that this concurrency can be effectively exploited to speed up the solution of a problem. This depends on a number of factors including how much work is involved as each element of the recursive data structure is updated and on the characteristics of the target parallel computer.

#### Pipeline Pattern

An assembly line is a good analogy for this pattern. Suppose we want to manufacture a number of cars. The manufacturing process can be broken down into a sequence of operations each of which adds some component, say the engine or the windshield, to the car. An assembly line (pipeline) assigns a component to each worker. As each car moves down the assembly line, each worker installs the same component over and over on a succession of cars. After the pipeline is full (and until it starts to empty) the workers can all be busy simultaneously, all performing their operations on the cars that are currently at their stations.

#### Event-Based Coordination Pattern

Some problems are most naturally represented as a collection of semi-independent entities interacting in an irregular way. What this means is perhaps clearest if we compare this pattern with the Pipeline pattern. In the Pipeline pattern, the entities form a linear pipeline, each entity interacts only with the entities to either side, the flow of data is one-way, and interaction occurs at fairly regular and predictable intervals. In the Event-Based Coordination pattern, in contrast, there is no restriction to a linear structure, no restriction that the flow of data be one-way, and the interaction takes place at irregular and sometimes unpredictable intervals.

As a real-world analogy, consider a newsroom, with reporters, editors, fact-checkers, and other employees collaborating on stories. As reporters finish stories, they send them to the appropriate editors; an editor can decide to send the story to a fact-checker (who would then eventually send it back) or back to the reporter for further revision. Each employee is a semi-independent entity, and their interaction (for example, a reporter sending a story to an editor) is irregular.

### Supporting Structures Design Space

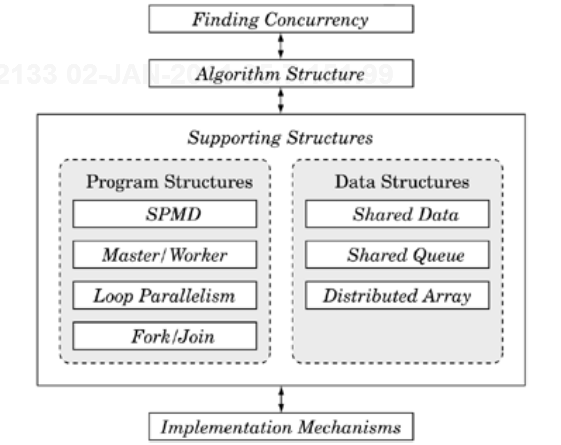


Figure 5. Supporting Structures [2]

#### SPMD Pattern

The programmer must manage multiple tasks running on multiple UEs. In addition, these tasks and UEs interact, either through exchange of messages or by sharing memory. In spite of these complexities, the program must be correct, and the interactions must be well orchestrated if excess overhead is to be avoided.

Fortunately, for most parallel algorithms, the operations carried out on each UE are similar. The data might be different between UEs, or slightly different computations might be needed on a subset of UEs (for example, handling boundary conditions in partial differential equation solvers), but for the most part each UE will carry out similar computations.

This pattern is by far the most commonly used pattern for structuring parallel programs. It is particularly relevant for MPI programmers and problems using the Task Parallelism and Geometric Decomposition patterns.

#### Loop Parallelism Pattern

This pattern addresses ways to structure loop-based programs for parallel computation. When existing code is available, the goal is to "evolve" a sequential program into a parallel program by a series of transformations on the loops. Ideally, all changes are localized to the loops with transformations that remove loop-carried dependencies and leave the overall program semantics unchanged.

#### Fork/Join Pattern

In some problems, the algorithm imposes a general and dynamic parallel control structure. Tasks are created dynamically (that is, forked) and later terminated (that is, joined with the forking task) as the program continues to execute.

Examples include recursively generated task structures, highly irregular sets of connected tasks, and problems where different functions are mapped onto different concurrent tasks. In each of these examples, tasks are forked and later joined with the parent task (that is, the task that executed the fork) and the other tasks created by the same fork. These problems are addressed in the Fork / Join pattern.

#### Shared Queue Pattern

Effective implementation of many parallel algorithms requires a queue that is to be shared among UEs. The most common situation is the need for a task queue in programs implementing the Master / Worker pattern.

### Implementation Mechanisms Design Space

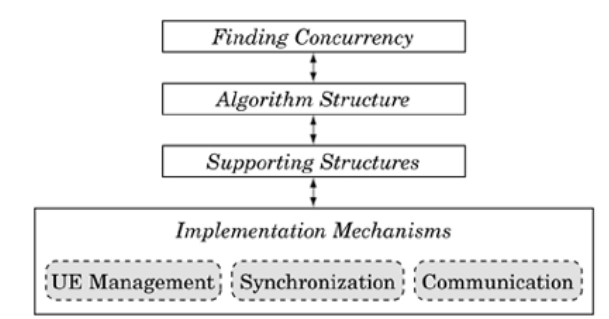


Figure 6. Implementation Mechanisms [2]

#### UE Management

The creation, destruction, and management of the processes and threads used in parallel computation.

#### Synchronization

Enforcing constraints on the ordering of events occurring in different UEs. This is primarily used to ensure that shared resources are accessed by a collection of UEs in such a way that the program is correct regardless of how the UEs are scheduled.

#### Communication

Deals with the exchange of information between UEs.

# REFERENCES

1. Patterns for Parallel Software Design, J.L. Ortega-Arjona, 2010
2. Patterns for Parallel Programming, T.G. Mattson, et.al., 2004
3. Multicore Programming Primer, [MIT Open Courseware](http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-189-multicore-programming-primer-january-iap-2007/index.htm), 2007