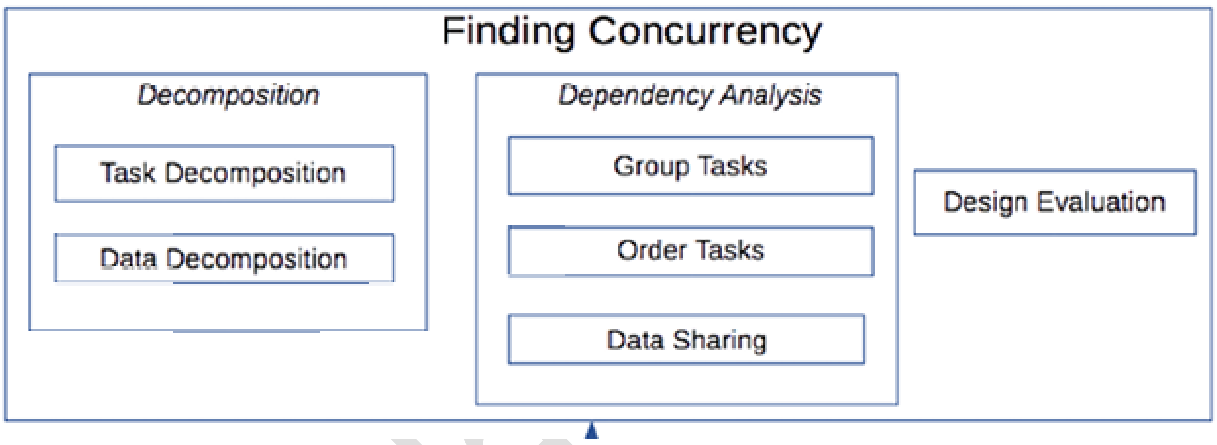
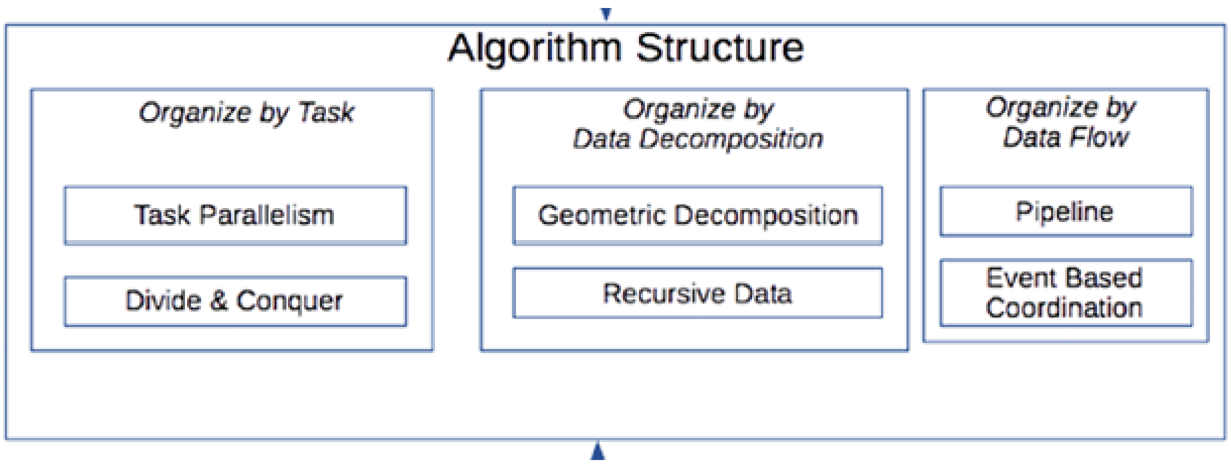
Design Pattern – is a representation of a common programming problem along with a tested, efficient solution for that problem.

They reflect untold **redesign and recoding** as developers have struggled for greater use and flexibility in their software.

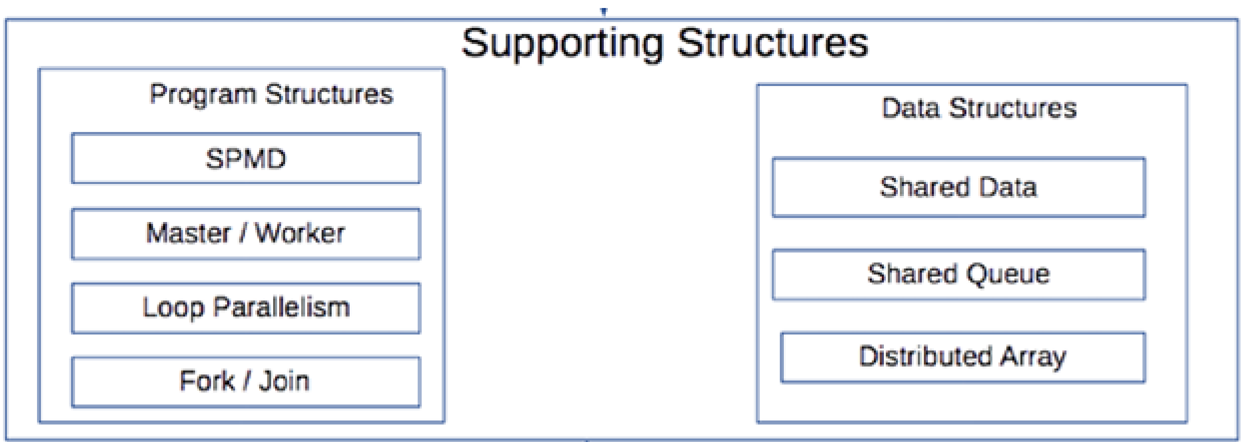
Parallel Design Pattern Design Spaces - Interesting aspects related to parallel design patterns is the **partitioning of the design** of a parallel application into 4 separate but related *design spaces* that roughly coincide with the step in creating a parallel program	
Finding concurrency	This design spaces is "**concerned with structuring the problem to expose exploitable concurrency"**. Programmer take a problem & search out the areas.
Algorithm Structure	In this design space, the programmer attempts to find and structure the algorithms that can take advantage of the exposed concurrency.
Supporting Structure	Begins to **map the algorithms** to data structures & to **more detailed program structures** like loops and recursion.
Implementation Mechanism	Finally, the design is **mapped** into particular parallel programming frameworks.



Decomposition Dimension – includes just 2 meta-patterns that are used to find and divide the problem into parts that can execute concurrently:	
Task Decomposition	View a complex algorithm as a set of instructions that can be **grouped into a set of tasks** that can be executed concurrently.
Data Decomposition	Takes the data used by the program and attempts to **divide it into chunks** that can be used by each of the tasks.
Dependency Analysis – includes 3 different meta-patterns whose job it is to group the tasks found above & to analyze the dependencies between them:	
Group Tasks	Aims at modeling the more convenient grouping of tasks such that the management of **dependencies is simplified**
Order Tasks	Aims at figuring out **how tasks (or groups of tasks) may be ordered** to satisfy the application constraints related to task execution.
Data Sharing	Aims to model the **accesses to a shared data structure.**



Algorithm Structure – output from the *finding concurrency* design space is used in the *algorithm structure* design space to refine the design of our **concurrent tasks** and to create a **parallel program structure** closer to an actual parallel program suitable to be run on a parallel target architecture.	
Task Decomposition	The tasks themselves drive your design. That is, consider the tasks that can be **computed in parallel**, which tasks in your set are concurrent and then how they're enumerated, linearly or recursively. *Organize by tasks* • **Tasks Parallelism -** Pattern that governs the efficient execution of collections of tasks. • **Divide & Conquer -** Problem is **divided** up into a number of smaller, **identical sub-problems**, which are then solved, & the solutions are **combined** into a **single final** solution for the original problem.
Data Decomposition	Data can be distributed into **discrete data sets** & the entire problem can be solved by operating on each data set independently, then choose the Geometric Decomposition meta-pattern. If the data is **organized recursively** (say, as a binary tree), choose the Recursive Data meta-pattern. *Organize by data* • **Geometric Decomposition -** pattern represents all those computations where the algorithm is recognized as a series of computations on some core data structure and where that data structure is inherently **linear in nature**, such as an array, table, or matrix. • **Recursive Data -** This pattern works with those parallel computations created to work with some recursively **defined data structure**, where the data appears to be acted upon sequentially.
Organize by flow of data	You would consider using the *flow of data* when the organizing principle of algorithms is how the flow of data imposes an ordering on the tasks that make up the algorithm. If the flow of data is **one-way & consistent,** then the Pipeline meta-pattern is the choice. If the flow of data is **dynamic or unpredictable**, then you want the Event-based Coordination meta-pattern. • **Pipeline** - This pattern is for 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. • **Event-Based Coordination -** This pattern is for where a number of semi-independent concurrent activities interact in an **irregular way**, and interactions are determined by the flow of data between the concurrent activities.



Supporting Structures – Investigating those structures/patterns suitable to **support the implementation of the algorithms** planned when exploring the *algorithm structure* design space. **2 Groups of meta-patterns** are included in this design space. The 1st group is related to how to structure the program in order to **maximize parallelism-**the *program structures* meta-pattern group-and the second is related to **commonly used shared data structures**: the *data structures* meta-pattern group.

Program Structures	• **Single Program, Multiple Data (SPMD)** – all the processing elements (PEs) run the same program in parallel, but each PE has it's **own subset of the data.** • **Master/Worker –** Sets up a number of **concurrent Worker threads** or processes & a **single bag of tasks**. Workers will continue to take tasks out of the bag & execute them until the bag is empty. Bag of tasks is typically implemented as a shared queue. • **Loop Parallelism –** How to execute an algorithm with one or more compute-intensive loops. Describes how to create a parallel program where the **distinct interactions** of the loop are executed in parallel. • **Fork/Join –** A single thread or process will **fork off** some number of sub-processes that will all execute in parallel. The originating process will typically wait until the child process all **join before resuming** it's own execution.
Data Structures	• **Shared Data –** Managed in a number of **different parallel applications.** • **Shared Queue –** Are used to **support the interaction of concurrent activities** in different contexts, from threads to processes & concurrent activities running on CPU co-processors. • **Distributed Array –** Meta-pattern models all the aspects of the **parallel program** related to the management of **arrays partitioned** and **distributed** among different concurrent activities.

Implementation Mechanisms
Process/Thread Management
Synchronization
Communication

Implementation Mechanisms – 2nd (and lowest-level) design space, which includes the meta-patterns representing the **base mechanisms** related to support parallel computing abstractions typical of parallel programming:

- Concurrent activities
- Synchronization
- Communication

UE Management	Related to the **management** of the *units of execution* (process, threads). Deals with all the aspects related to the concurrent activities in a parallel application, including their creation, destruction, and management.
Synchronization	Related to **ordering of events/computations** in the UE used to execute the parallel application/
Communication	Deals with the aspects related to **data exchange among concurrent activities** & therefore convers aspects related to different kinds of point-to-point message passing & multi-point or collective communications where multiple UEs are involved in a single communication event.

2. List of Parallel Patterns

Embarrassingly Parallel	Not really a pattern, this is rather a **class of problems**. Examples include the **trapezoid rule program** to compute pi in Chapter 12, where we could compute the areas of any number of trapezoids in parallel.
Master/Worker	A **single Master** task will set up **a number of concurrent Worker** threads or processes and a single bag of tasks. Workers will continue to take tasks out of the bag & execute them **until the bag is empty** or some other ending condition has occurred.
Map & Reduce	Map pattern is likely the simplest pattern. The functions must have **no side-effects** & must be **identical** & **independent.**
MapReduce	Variation of using the Map & Reduce patterns combines the 2 to accomplish a common task. **Main goal** is to **input, process,** and **generate.** Has **3 Essential functions** • **Mapping -** The program **divides** up a large data set into N discrete & independent subsets, each of which will be processed on a single processor. • **Shuffle –** The program **extracts** similar (key, value) pairs & assigns them to a new processor where the reduce operation will happen • **Reduce –** The elements in an input **data set are combined** into a **single result** that's output.
Divide & Conquer	Problem are solved independently & the partial solutions combined back
Fork/Join	Number of threads **will vary** as the program executes. Program execution and then **wait for them** to finish before proceeding. The originating process will typically **wait** until the child **processes all join** before resuming it's own execution. This pattern also ahs the possibility of having ***nested parallel execution*** *regions* that can complicate the performance of the program.

A Last Word on Parallel Design Patterns

1. Identify concurrency
2. Split the program into the concurrent pieces
3. Split the data if the updated algorithm calls for it
4. Execute the concurrent pieces in parallel
5. Put all the answers back together to make a final answer.