Design patterns for HPC: an introduction

Paolo Ciancarini – paolo.ciancarini@unibo.it
Department of Informatics – University of Bologna

Motivation and Concept

- Software designers should exploit reusable design knowledge promoting
  - Abstraction and elegance
  - Flexibility and modularity
  - Uncoupling and cohesion
- Problem: capturing, communicating, and reuse design knowledge

Patterns for HPC

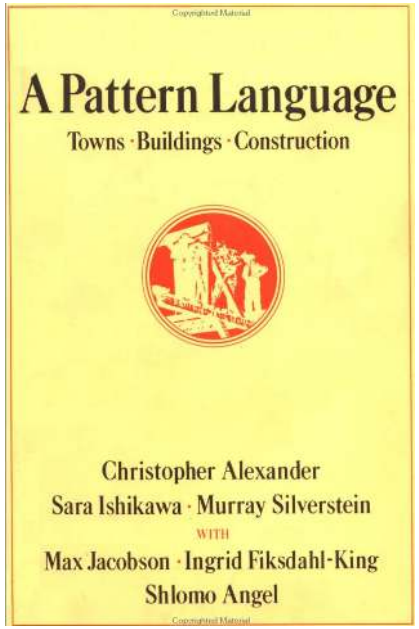
Which technologies improve the productivity of HPC software development?

- Parallel programming languages and libraries,
- Object Oriented scientific programming, and
- parallel run-time systems and tools

The success of these activities requires some understanding of common patterns used in the development of HPC software:

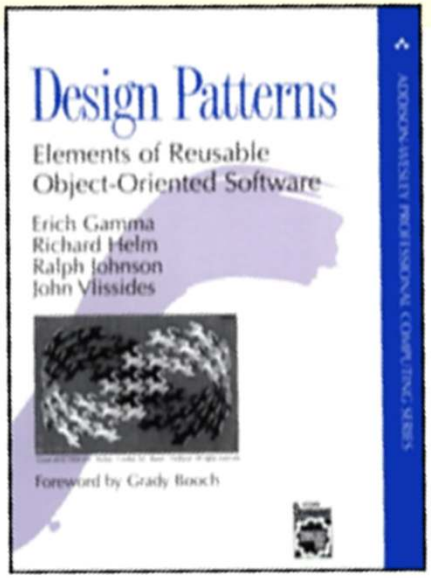
- patterns used for the coding of parallel algorithms,
- their mapping to various architectures, and
- performance tuning activities

A discipline of design



- Christopher Alexander's approach to (civil) architecture:
  - A design pattern "describes a problem which occurs over and over, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." *A Pattern Language,* Christopher Alexander
- A pattern language is an organized way of tackling an architectural problem using patterns

- Gamma used patterns to bring order to the chaos of object oriented design.
- The book turned object oriented design from "an art" to a systematic design discipline



What is a design pattern?

A design pattern

- Is a reusable solution to a common problem in software design
- Abstracts a *recurring design structure*
- Includes class or object
  - dependencies
  - structures
  - interactions
- Names and specifies the design structure explicitly
- Distils design experience

What is a design pattern?

A design pattern is a description of a reusable design:

1. Name
2. Problem
3. Solution
4. Consequences and trade-offs of application

- Language- and implementation-independent
- A “micro-architecture”
- Adjunct to existing methods (eg. UML, C++, etc.)
- No mechanical application
  - The solution needs to be translated by the developer into concrete terms in the application context

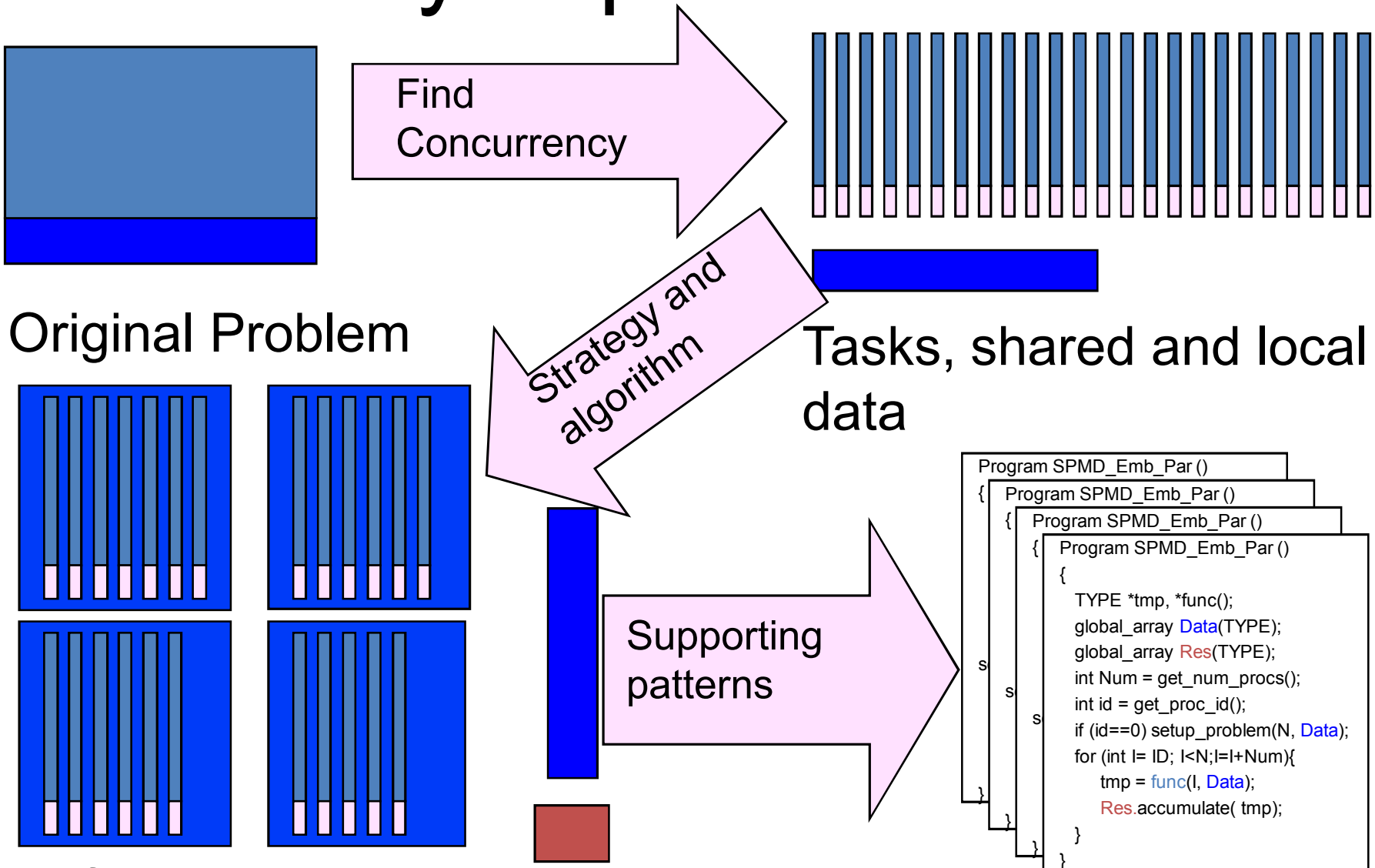
Getting started with parallel algorithms

- Concurrency is a general concept
  - … multiple activities that can occur and make progress at the same time.
- A parallel algorithm is any algorithm that uses concurrency to solve a problem of a given size in less time
- Scientific programmers have been working with parallelism since the early 80's
  - Hence we have almost 30 years of experience to draw on to help us understand parallel algorithms.

Basic approach

- Identify the concurrency in your problem:
  - Find the tasks, data dependencies and any other constraints.
- Develop a strategy to exploit this concurrency:
  - Which elements of the parallel design will be used to organize your approach to exploiting concurrency.
- Identify and use the right algorithm pattern to turn your strategy into the design of a specific algorithm.
- Choose the supporting patterns to move your design into source code.
  - This step is heavily influenced by the target platform

Concurrency in parallel software



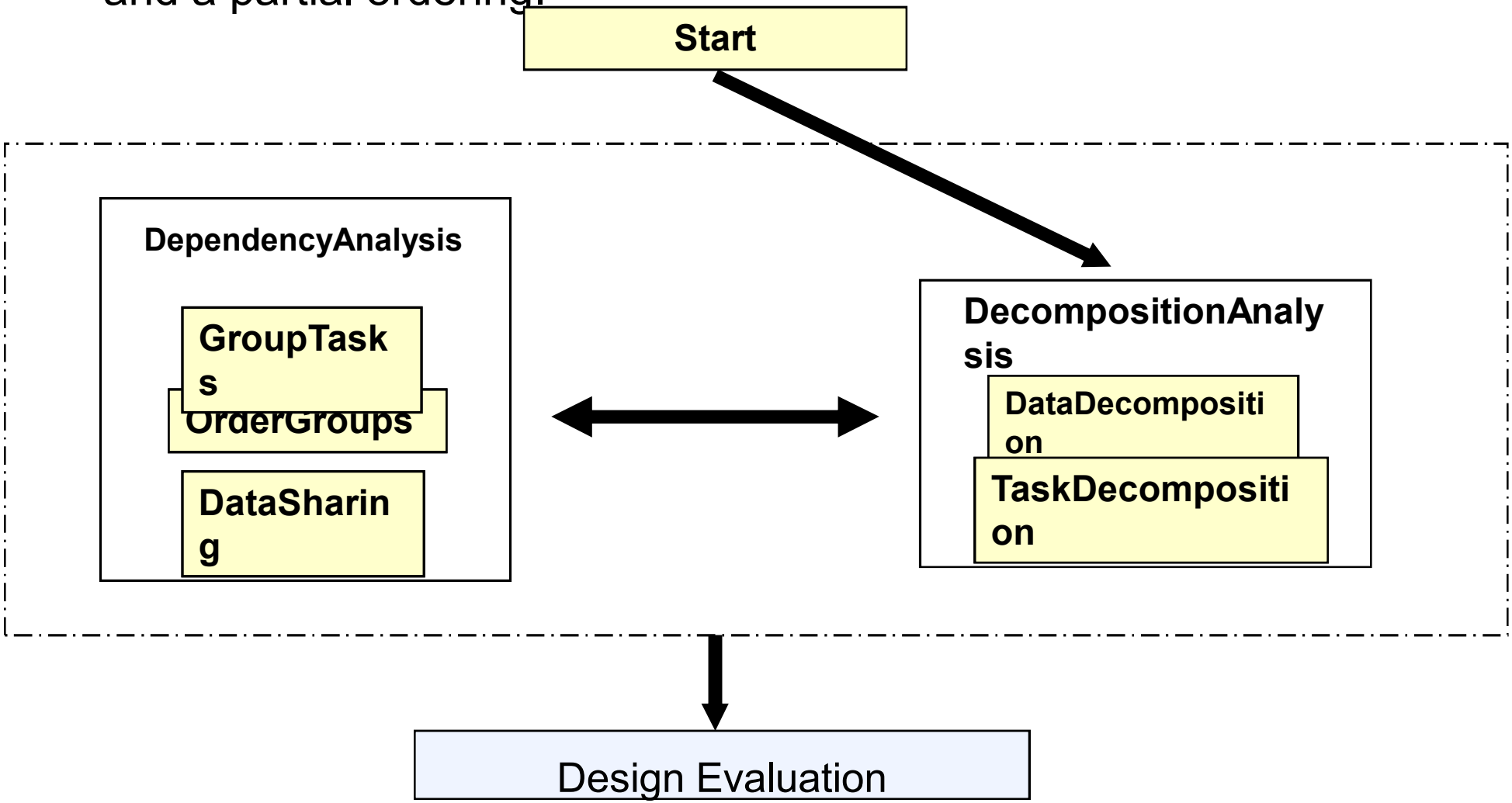
Exploiting concurrency: the big picture

Exploiting concurrency "always" involves the following 3 steps:

1. Identify the concurrent tasks
   - A task is a logically related sequence of operations.
2. Decompose the problem's data to minimize overhead of sharing or data-movement between tasks
3. Describe dependencies between tasks: both ordering constraints and data that is shared between tasks.

Exploiting concurrency

Start with a specification that solves the original problem -- finish with the problem decomposed into parallel tasks, shared data, and a partial ordering.



Example: File Search

- You have a collection of thousands of data-files
  - Input: A string
  - Output: The number of times the input string appears in the collection of files.
- Finding concurrency
  - Task decomposition: The search of each file defines a distinct task.
  - Data decomposition: Assign each file to a task
  - Dependencies:
    - A single group of tasks that can execute in any order … hence no partial orders to enforce
    - Data Dependencies: A single global counter for the number of times the string is found

Summary

- The fundamental insight to remember is that **ALL** parallel algorithms require a task decomposition **AND** a data decomposition.
  - Even so-called "strictly data parallel" algorithms are based on implicitly defined tasks.

Strategies for exploiting concurrency

- Given the results from your "concurrency" analysis, there are many different ways to turn them into a parallel algorithm.
- In most cases, one of three strategies are used
  - Agenda parallelism: The collection of tasks that are to be computed.
  - Result parallelism: Updates to the data.
  - Specialist parallelism: The flow of data between a fixed set of tasks.

Ref: N. Carriero and D. Gelernter, **How to Write Parallel Programs: A First Course**, 1990.

Agenda parallelism

- For agenda parallelism, the parallel algorithm is naturally expressed in terms of the actions to be carried out by the program … i.e. the program′s ″agenda″.
  - Each ″action″ defines the task and this is the natural way to think about the concurrency.
- These tasks may be:
  - Statically defined up front.
  - Dynamically generated as the computation proceeds.
  - Spawned as a recursive data structure is traversed.
- Algorithms based on Agenda parallelism center on how the tasks are created, managing their execution to balance the load among all processing elements, and correct (and scalable) combination of results into the final answer.

Result parallelism

- In result parallelism, the algorithm is designed around what you will be computing, i.e., the data decomposition guides the design of the algorithm.
- These problems revolve around a central data structure that hopefully presents a natural decomposition strategy.
- For these algorithms, the resulting programs focus on breaking up the central data structure and launching threads or processes to update them concurrently.

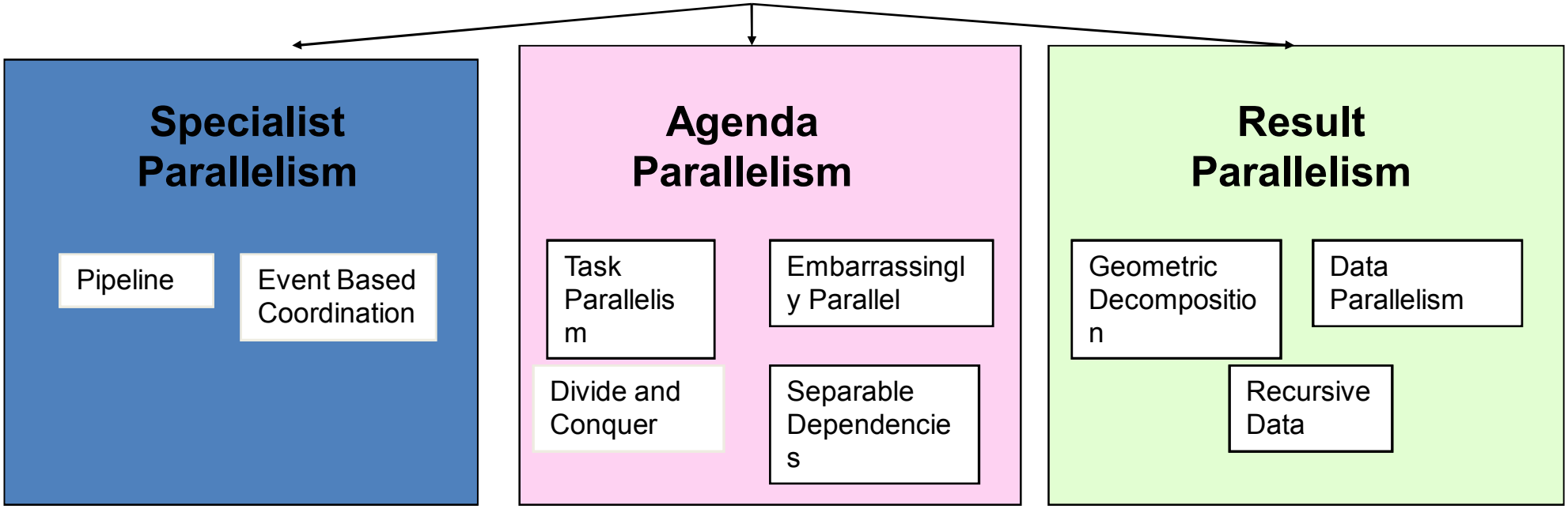
Specialist Parallelism

- Specialist parallelism occurs when the problem can be defined in terms of data flowing through a fixed set of tasks.
- This strategy works best when there are a modest number of compute intensive tasks. When there are large numbers of ne grained tasks, its usually better to think of the problem in terms of the agenda parallelism strategy.
- An extremely common example of specialist parallelism is the linear pipeline of tasks, though algorithms with feedback loops and more complex branching structure fit this strategy as well.

The Algorithm Design Patterns

Start with a basic concurrency decomposition

- A problem decomposed into a set of tasks
- A data decomposition aligned with the set of tasks … designed to minimize interactions between tasks and make concurrent updates to data safe.
- Dependencies and ordering constraints between groups of tasks.



Specialist Parallelism: Algorithm Patterns

- A fixed set of tasks that data flows through.
- Pipeline:
  - A collection of tasks or pipeline stages are defined and connected in terms of a fixed communication pattern. Data flows between stages as computations complete; the parallelism comes from the stages executing at the same time once the pipeline is full. The pipeline stages can include branches or feedback loops.
- Event-based coordination:
  - This pattern defines a set of tasks that run concurrently in response to events that arrive on a queue.

Agenda Parallelism: Algorithm Patterns

- These patterns naturally focuses on the tasks that are exposed by the problem.
  - Task parallel:
    - The set of tasks are defined statically or through iterative control structures. The crux of this pattern is to schedule the tasks so the computational load is evenly spread between the threads or processes and to manage the data dependencies between tasks.
  - Embarrassingly parallel:
    - This is a very important instance of the task parallel pattern in which there are no dependencies between the tasks. The challenge with this pattern is to distribute the tasks so the load is evenly balanced among the processing elements of the parallel system.
  - Separable dependencies:
    - A sub-pattern of the task parallel pattern in which the dependencies between tasks are managed by replicating key data structures on each thread or process and then accumulating results into these local structures. The tasks then execute according to the embarrassingly parallel pattern and the local replicated data structures are combined into the final global result.
  - Recursive algorithms:
    - Tasks are generated by recursively splitting a problem into smaller sub-problems. These sub-problems are themselves split until at some point the generated sub-problems are small enough to solve directly. In a divide and conquer algorithm, the splitting is reversed to combine the solutions from the sub-problems into a single solution for the original problem.

Result Parallelism: Algorithm Patterns

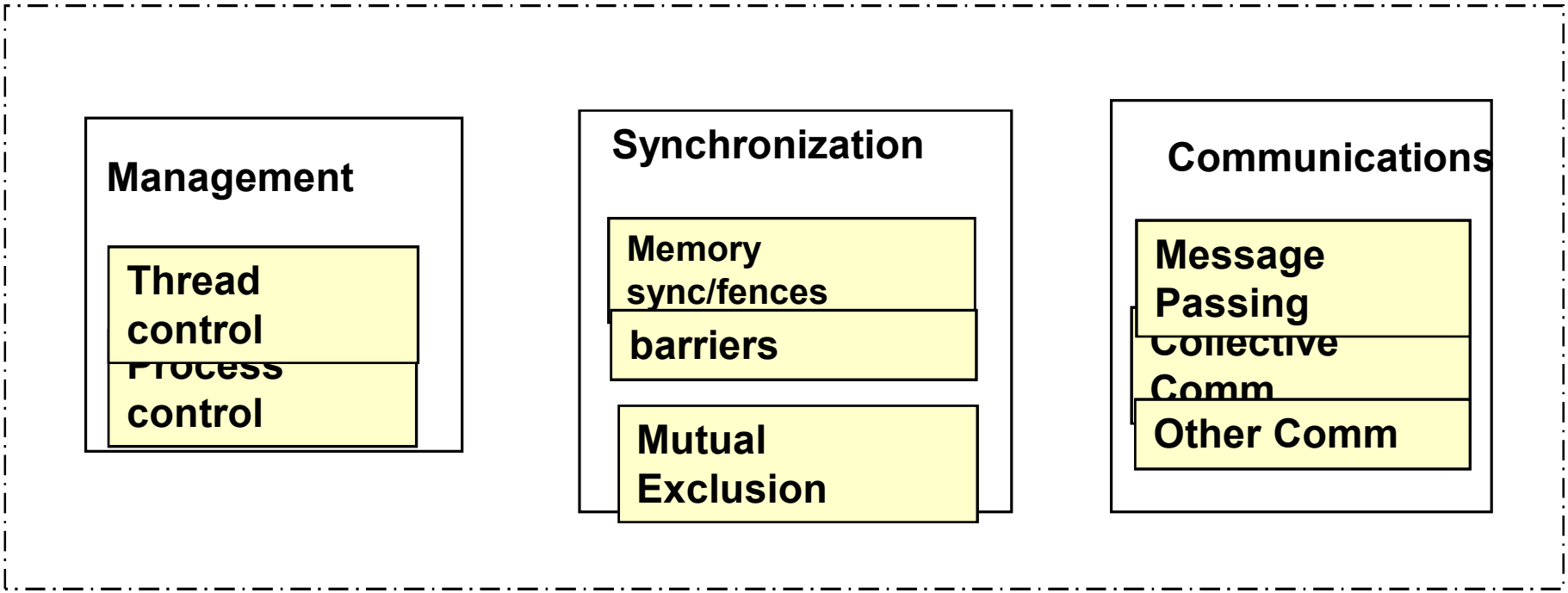
- The core idea is to define the algorithm in terms of the data structures within the problem and how they are decomposed.
  - Data parallelism:
    - A broadly applicable pattern in which the parallelism is expressed as streams of instructions applied concurrently to the elements of a data structure (e.g., arrays).
  - Geometric decomposition:
    - A data parallel pattern where the data structures at the center of the problem are broken into sub-regions or tiles that are distributed about the threads or processes involved in the computation. The algorithm consists of updates to local or interior points, exchange of boundary regions, and update of the edges.
  - Recursive data:
    - A data parallel pattern used with recursively defined data structures. Extra work (relative to the serial version of the algorithm) is expended to traverse the data structure and define the concurrent tasks, but this is compensated for by the potential for parallel speedup.

Supporting Patterns

- Fork-join
  - A computation begins as a single thread of control. Additional threads are created as needed (forked) to execute functions and then when complete terminate (join). The computation continues as a single thread until a later time when more threads might be useful.
- SPMD
  - Multiple copies of a single program are launched typically with their own view of the data. The path through the program is determined in part base don a unique ID (a rank). This is by far the most commonly used pattern with message passing APIs such as MPI.
- Loop parallelism
  - Parallelism is expressed in terms of loops that execute concurrently.
- Master-worker
  - A process or thread (the master) sets up a task queue and manages other threads (the workers) as they grab a task from the queue, carry out the computation, and then return for their next task. This continues until the master detects that a termination condition has been met, at which point the master ends the computation.
- SIMD
  - The computation is a single stream of instructions applied to the individual components of a data structure (such as an array).
- Functional parallelism
  - Concurrency is expressed as a distinct set of functions that execute concurrently. This pattern may be used with an imperative semantics in which case the way the functions execute are defined in the source code (e.g., event based coordination). Alternatively, this pattern can be used with declarative semantics, such as within a functional language, where the functions are defined but how (or when) they execute is dictated by the interaction of the data with the language model.

The Implementation Mechanisms

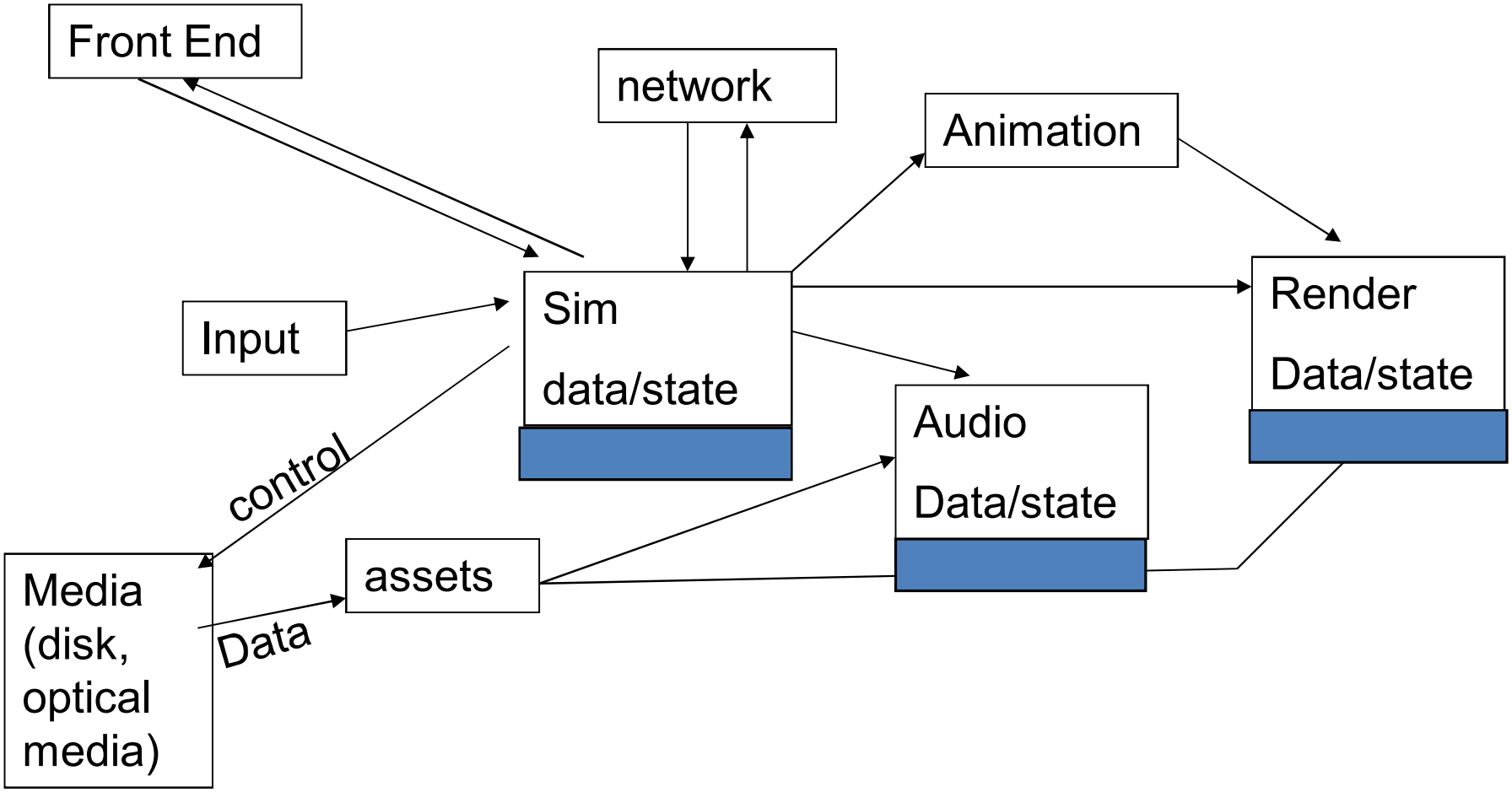
The low level building blocks parallel computing.



Example: A parallel game engine

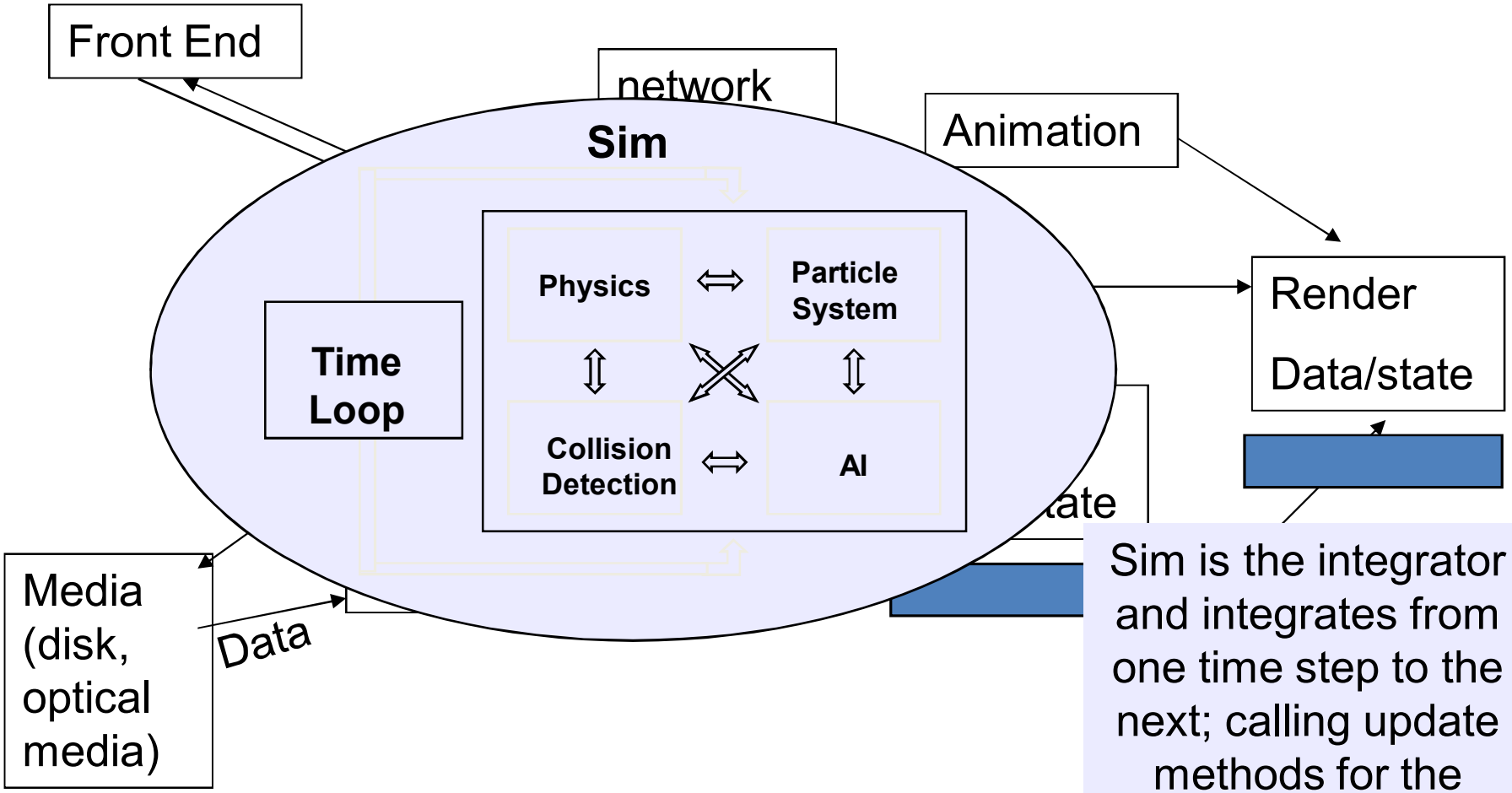
- Computer games represent an important yet VERY difficult class of parallel algorithm problems.
- A computer game is:
  - An HPC problem … physics simulations, AI, complex graphics, etc.
  - Real time programming … latencies must be less than 50 milliseconds and ideally MUCH lower (16 millisecs … to match the frame update rate for satisfactory graphics).
- The computational core of a computer game is the Game-engine.

The heart of a game is the “Game Engine”



game state in its internally managed data structures.

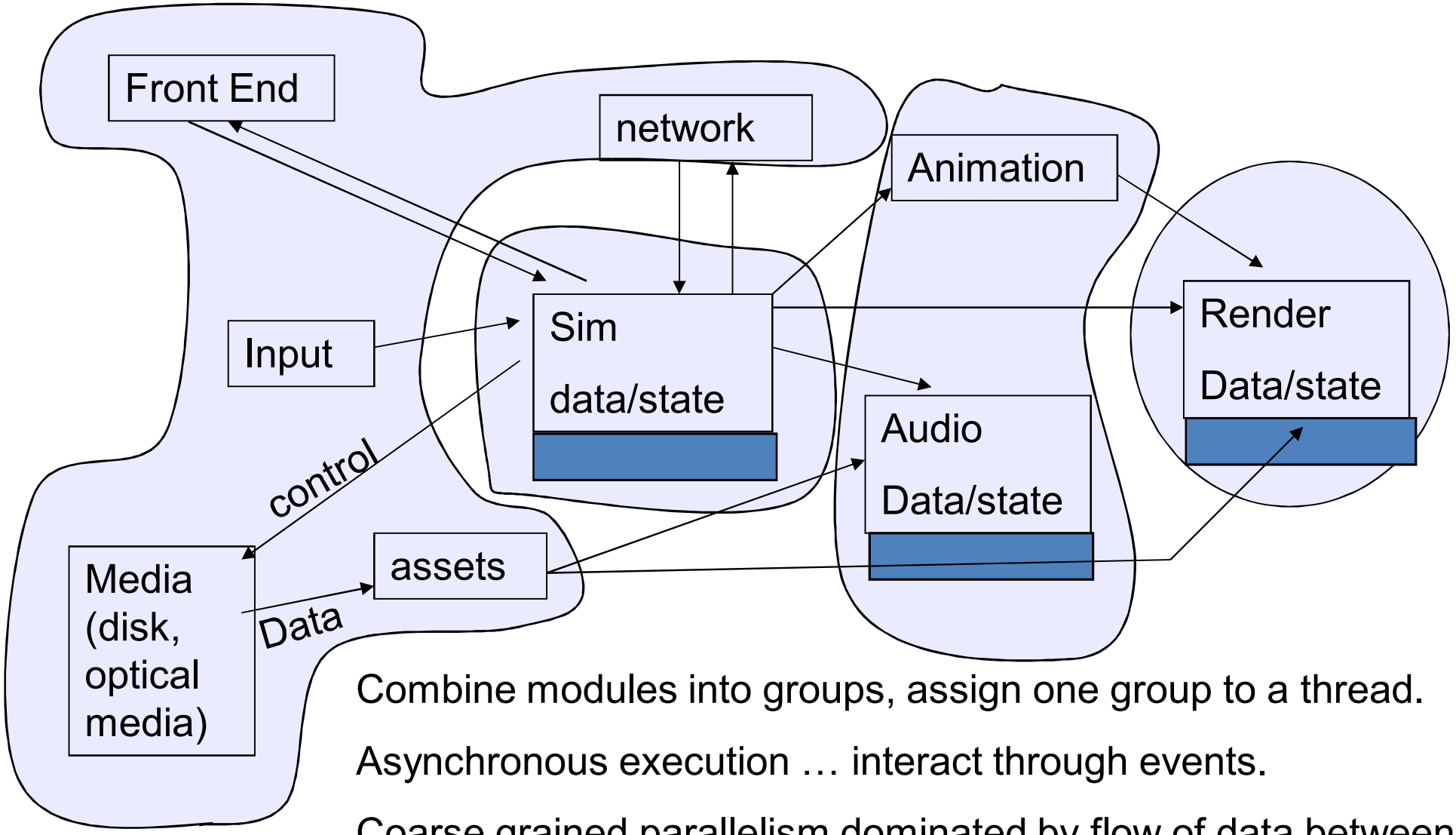
The heart of a game is the “Game Engine”



Sim is the integrator and integrates from one time step to the next; calling update methods for the other modules inside a central time loop.

game state in its internally managed data structures.

Finding concurrency: Specialist parallelism



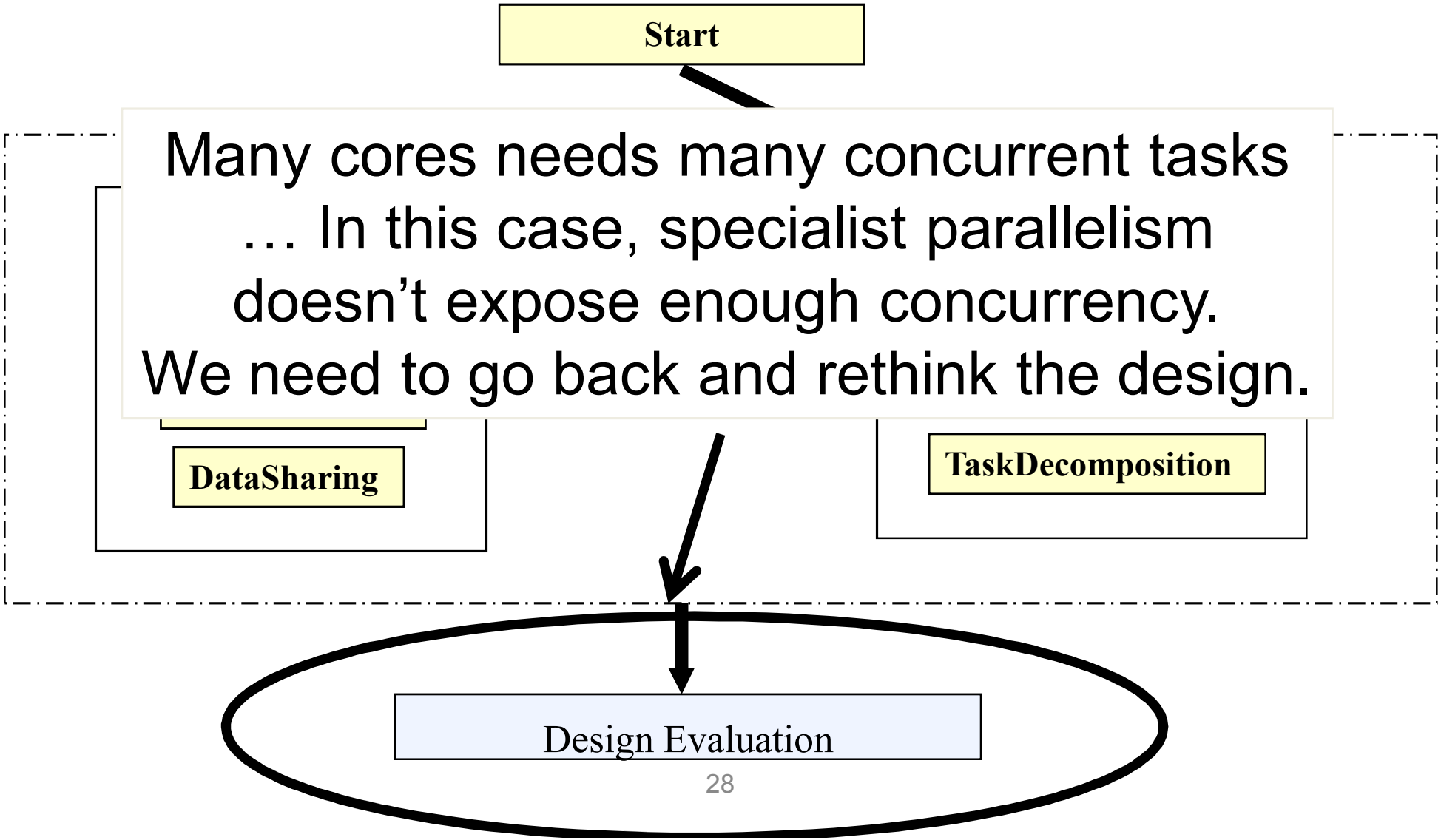
Combine modules into groups, assign one group to a thread.

Asynchronous execution … interact through events.

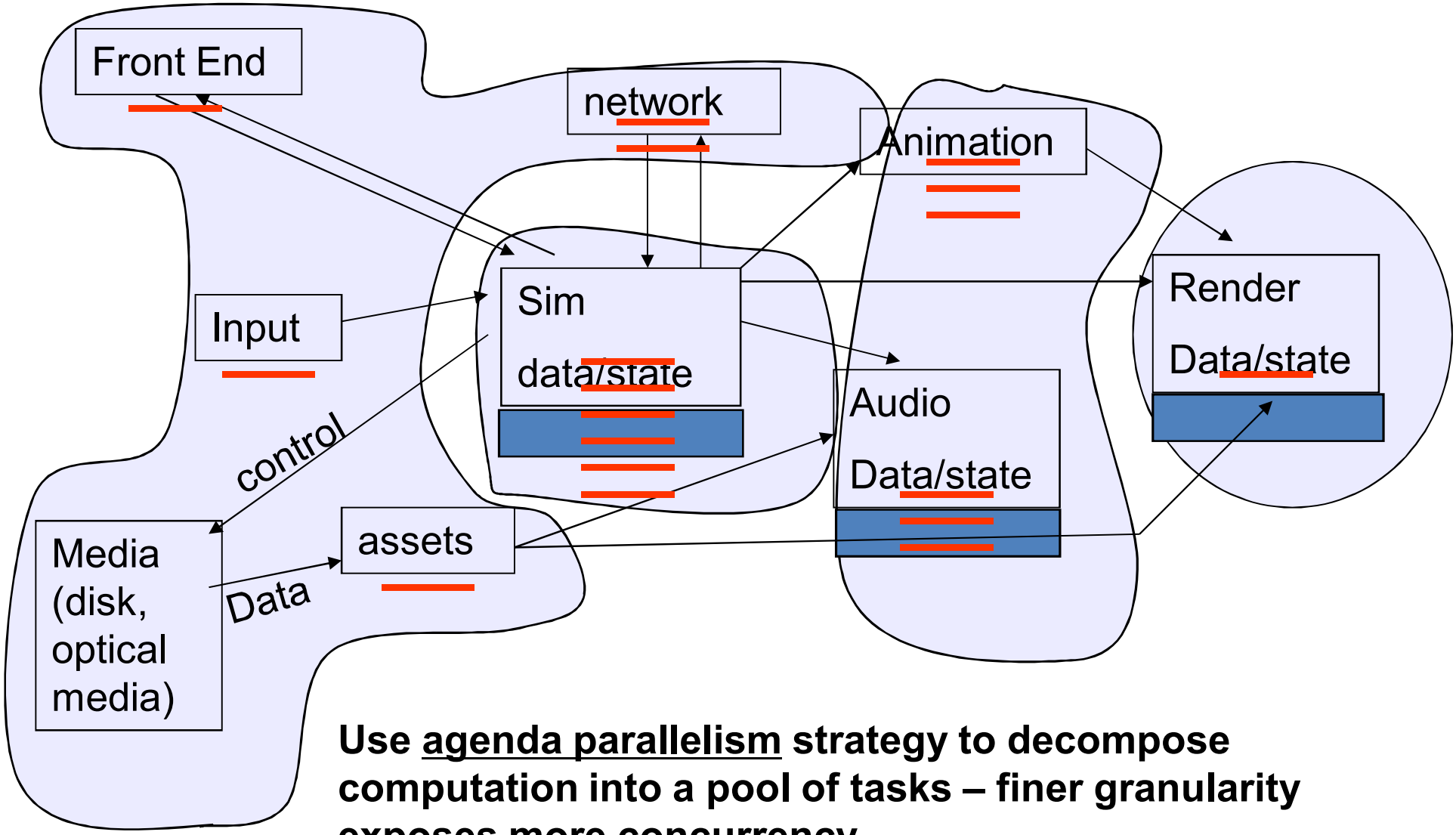
Coarse grained parallelism dominated by flow of data between groups … Specialist parallelism strategy.

The Finding Concurrency Analysis

Start with a specification that solves the original problem -- finish with the problem decomposed into tasks, shared data, and a partial ordering.



More sophisticated parallelization strategy

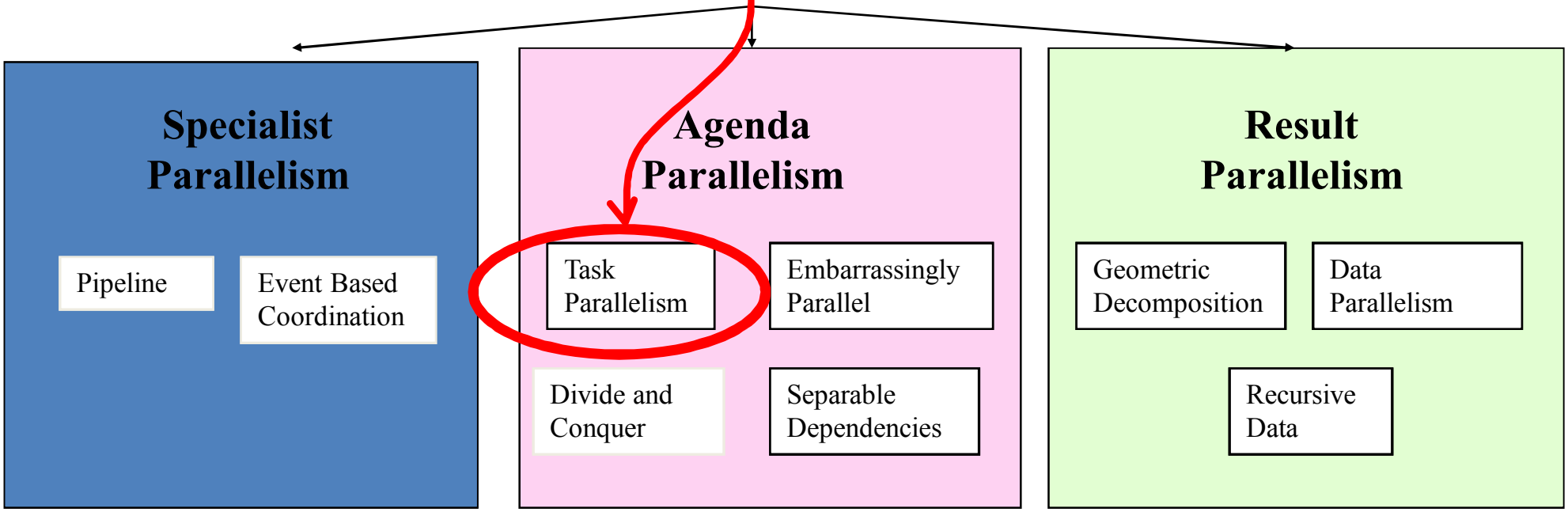


Use <u>agenda parallelism</u> strategy to decompose computation into a pool of tasks – finer granularity exposes more concurrency.

The Algorithm Design Patterns

Start with a basic concurrency decomposition

- A problem decomposed into a set of tasks
- A data decomposition aligned with the set of tasks … designed to minimize interactions between tasks and make concurrent updates to data safe.
- Dependencies and ordering constraints between groups of tasks.



Supporting Patterns

- Fork-join
  - A computation begins as a single thread of control. Additional threads are created as needed (forked) to execute functions and then when complete terminate (join). The computation continues as a single thread until a later time when more threads might be useful.
- SPMD
  - Multiple copies of a single program are launched typically with their own view of the data. The path through the program is determined in part base don a unique ID (a rank). This is by far the most commonly used pattern with message passing APIs such as MPI.
- Loop parallelism
  - Parallelism is expressed in terms of loops that execute concurrently.
- Master-worker
  - A process or thread (the master) sets up a task queue and manages other threads (the workers) as they grab a task from the queue, carry out the computation, and then return for their next task. This continues until the master detects that a termination condition has been met, at which point the master ends the computation.
- SIMD
  - The computation is a single stream of instructions applied to the individual components of a data structure (such as a
- Functional parallelism
  - Concurrency is exp ay be used with an imperative semantics in which case the way the functions execute are defined in the source code (e.g., event based coordination). Alternatively, this pattern can be used with declarative semantics, such as within a functional language, where the functions are defined but how (or when) they execute is dictated by the interaction of the data with the language model.

Tasks vary widely so you need a supporting pattern that helps with dynamic load balancing.

MapReduce and Hadoop

based on material by

K. Madurai and B. Ramamurthy

Big-data

- Data mining huge amounts of data collected in a wide range of domains from astronomy to healthcare has become essential for planning and performance
- We are in a knowledge economy
  - Data is an important asset to any organization
  - Discovery of knowledge; Enabling discovery; annotation of data
- We are looking at newer
  - programming models, and
  - Supporting algorithms and data structures

What is MapReduce?

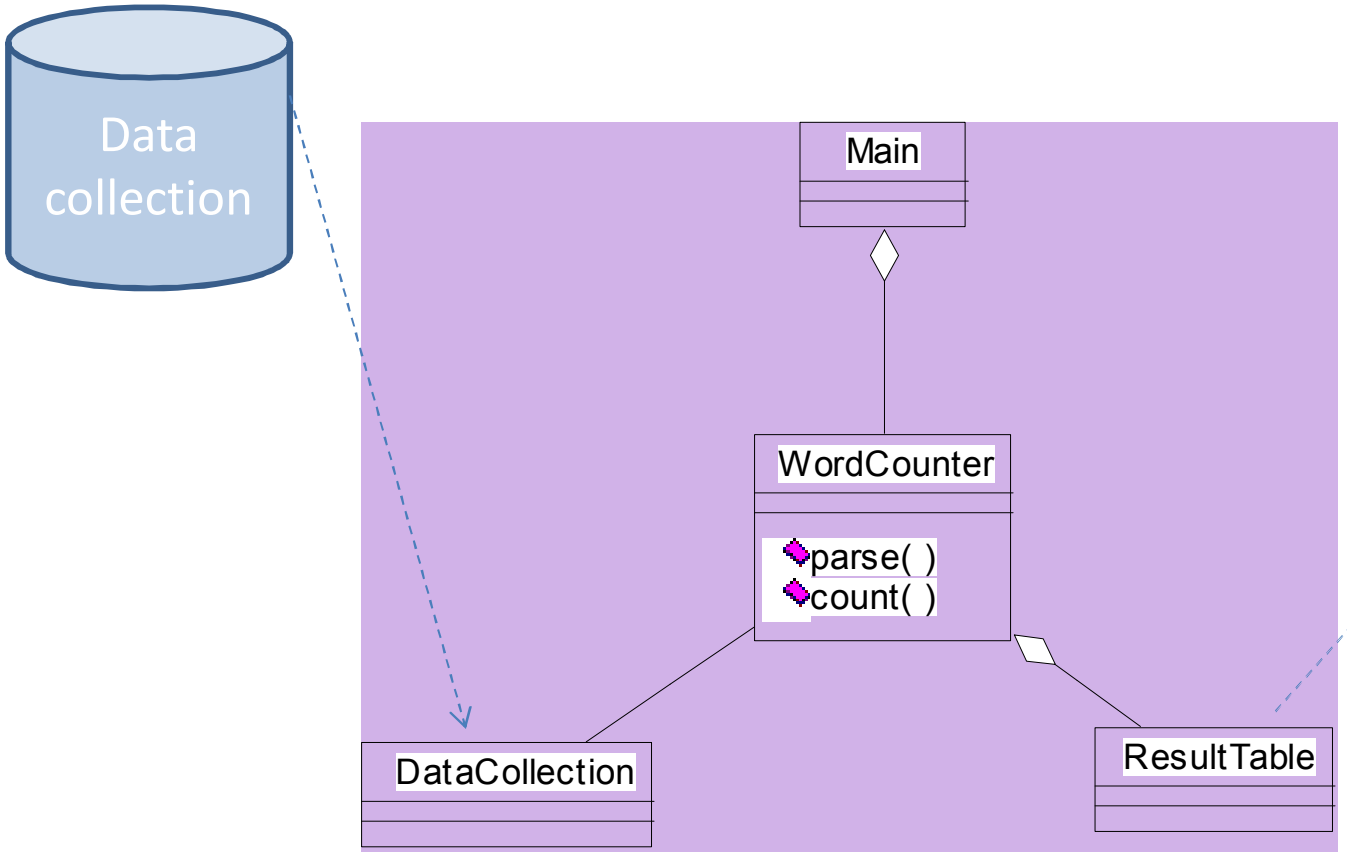
- MapReduce is a programming model
  - Google has used successfully in processing its "big-data" sets (~ 20000 peta bytes per day)
- Users specify the computation in terms of a map and a reduce function
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine failures, efficient communications, and performance issues

Towards MapReduce

- Consider a large data collection:
  - {web, weed, green, sun, moon, land, part, web, green, …}
  - Problem: Count the occurrences of the different words in the collection

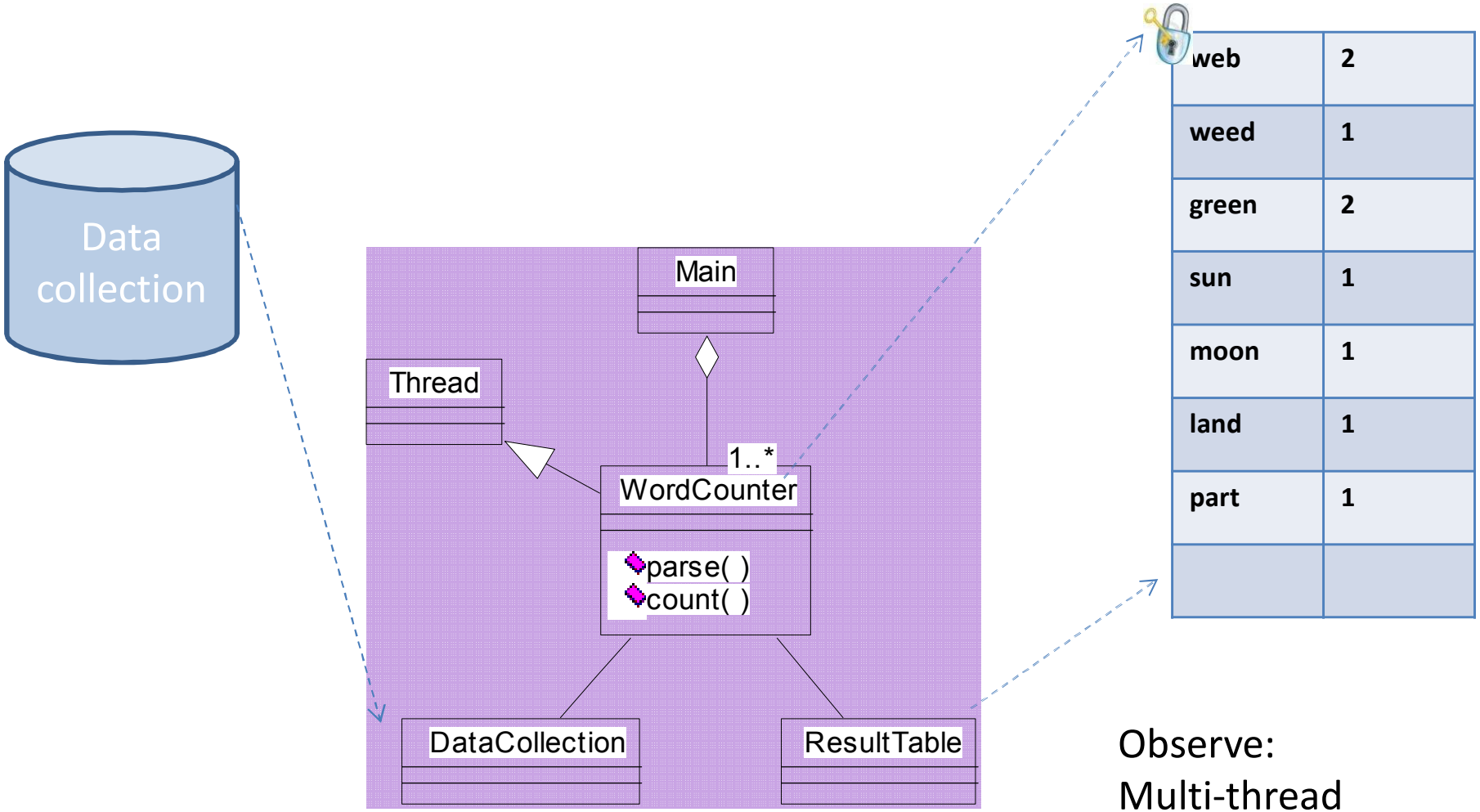
Word Counter and Result Table

{web, weed, green, sun, moon, land, part,
web, green,...}



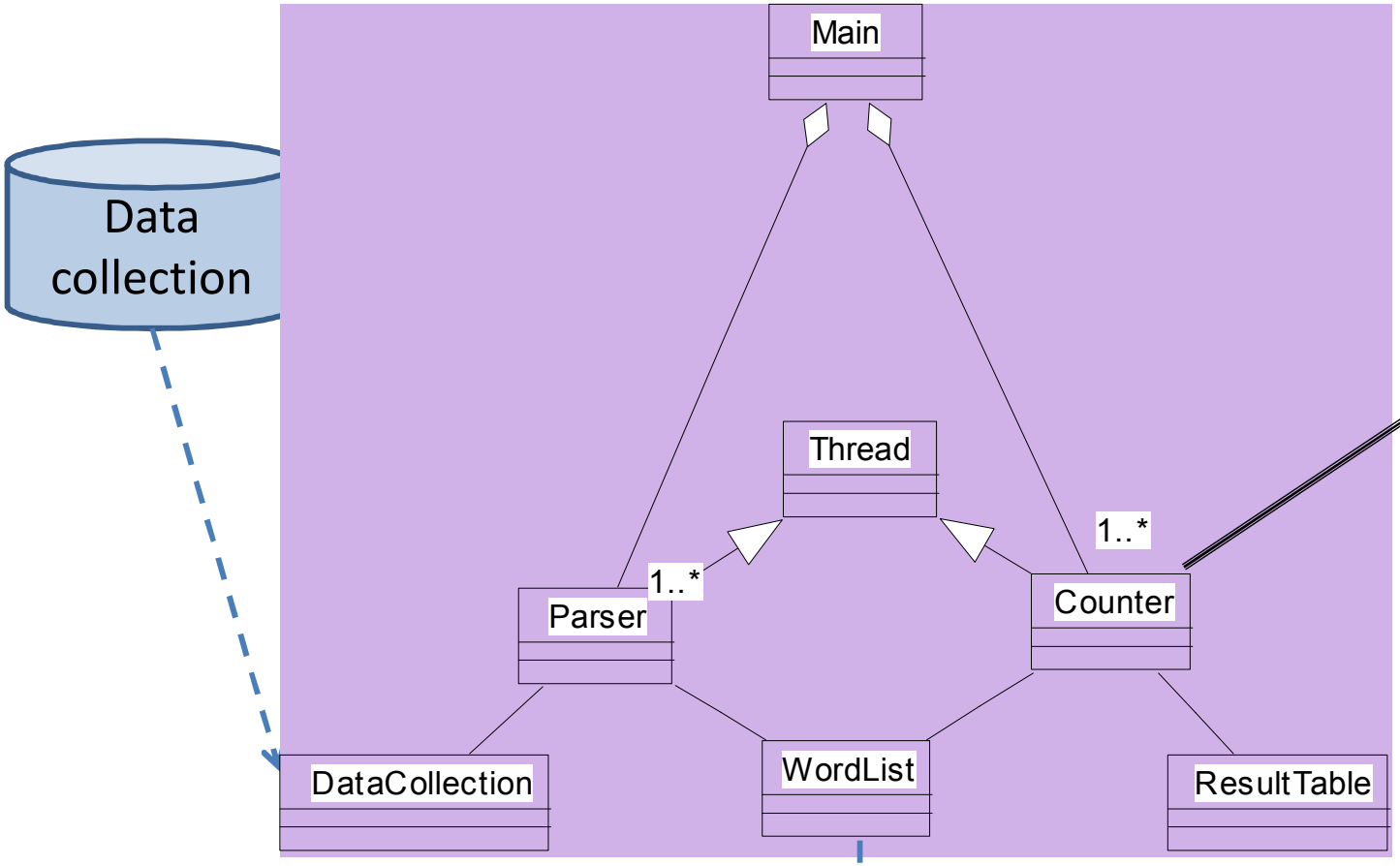
web	**2**
weed	**1**
green	**2**
sun	**1**
moon	**1**
land	**1**
part	**1**

Multiple Instances of Word Counter



Observe:
Multi-thread
Lock on shared data

Improve Word Counter for Performance



No need for lock

web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

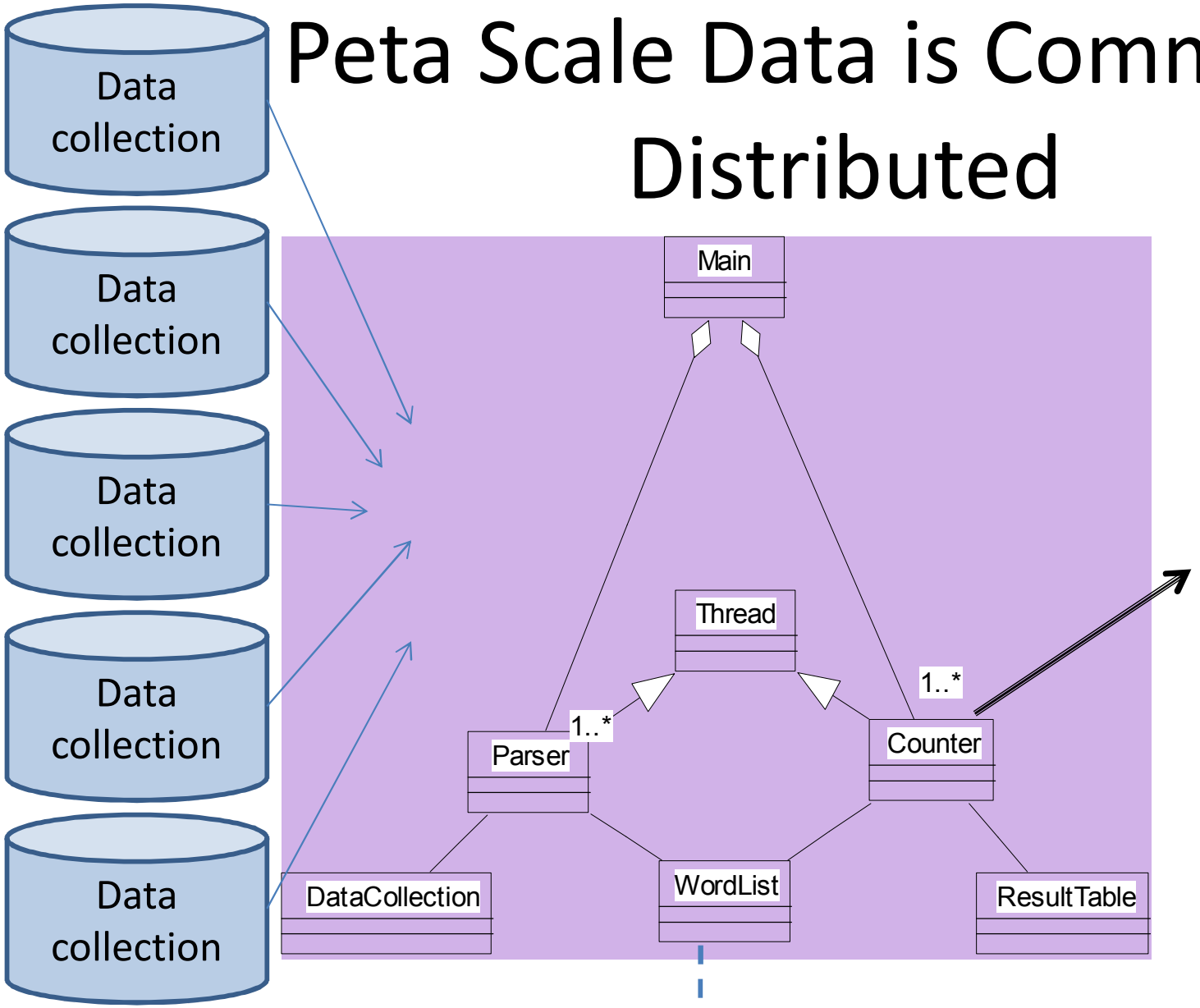
Separate counters

KEY	web	weed	green	sun	moon	land	part	web	green	.....
VALUE										

Addressing the Scale Issue

- Single machine cannot serve all the data: you need a distributed special (file) system
- Failure is norm and not an exception
  - File system has to be fault-tolerant: replication, checksum
- Data transfer bandwidth is critical (location of data)
- Critical aspects: fault tolerance + replication + load balancing, monitoring
- Exploit parallelism afforded by splitting parsing and counting

Peta Scale Data is Commonly Distributed

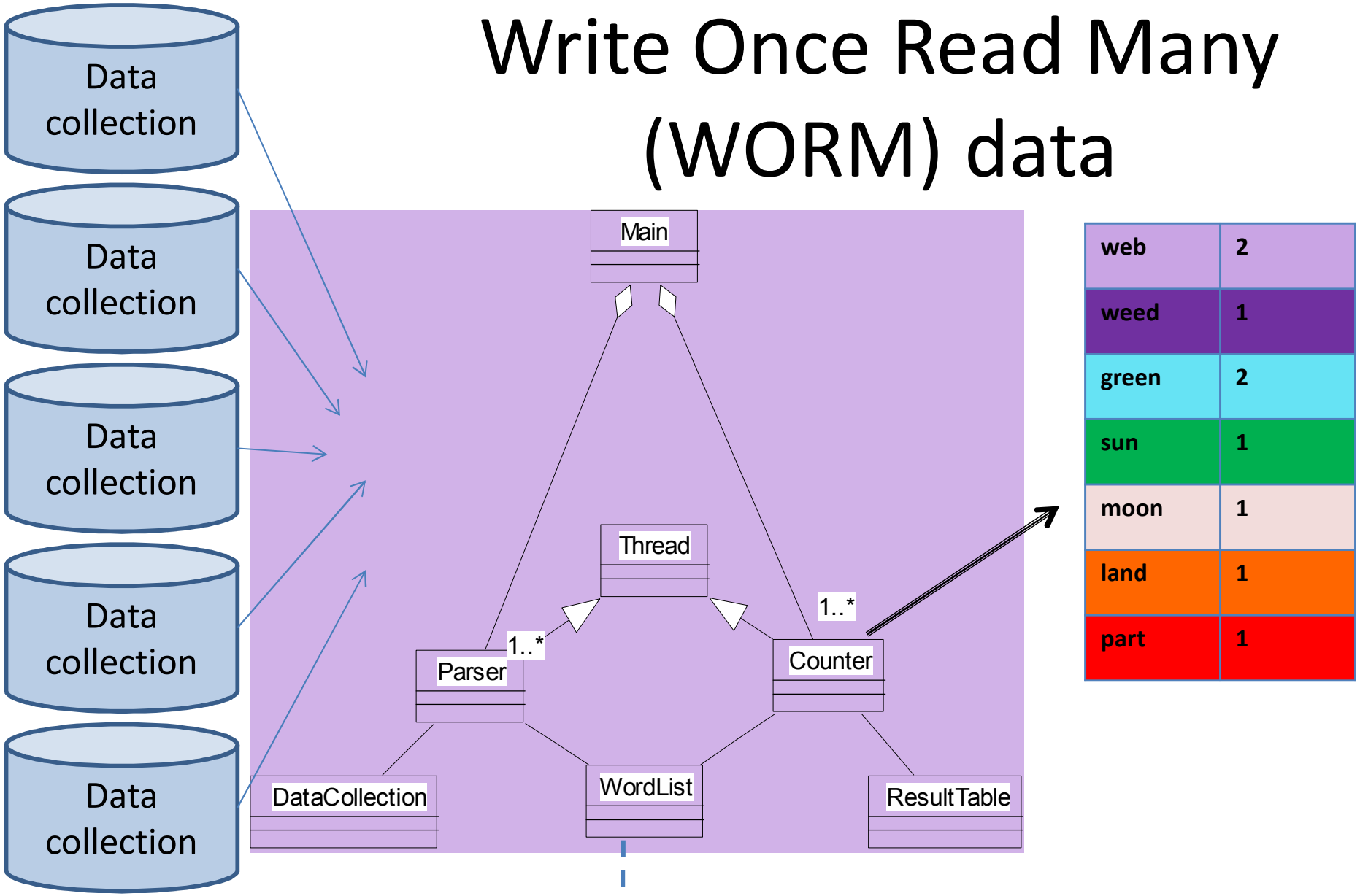


web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

Issue: managing the large scale data

KEY	web	weed	green	sun	moon	land	part	web	green	.....
VALUE										

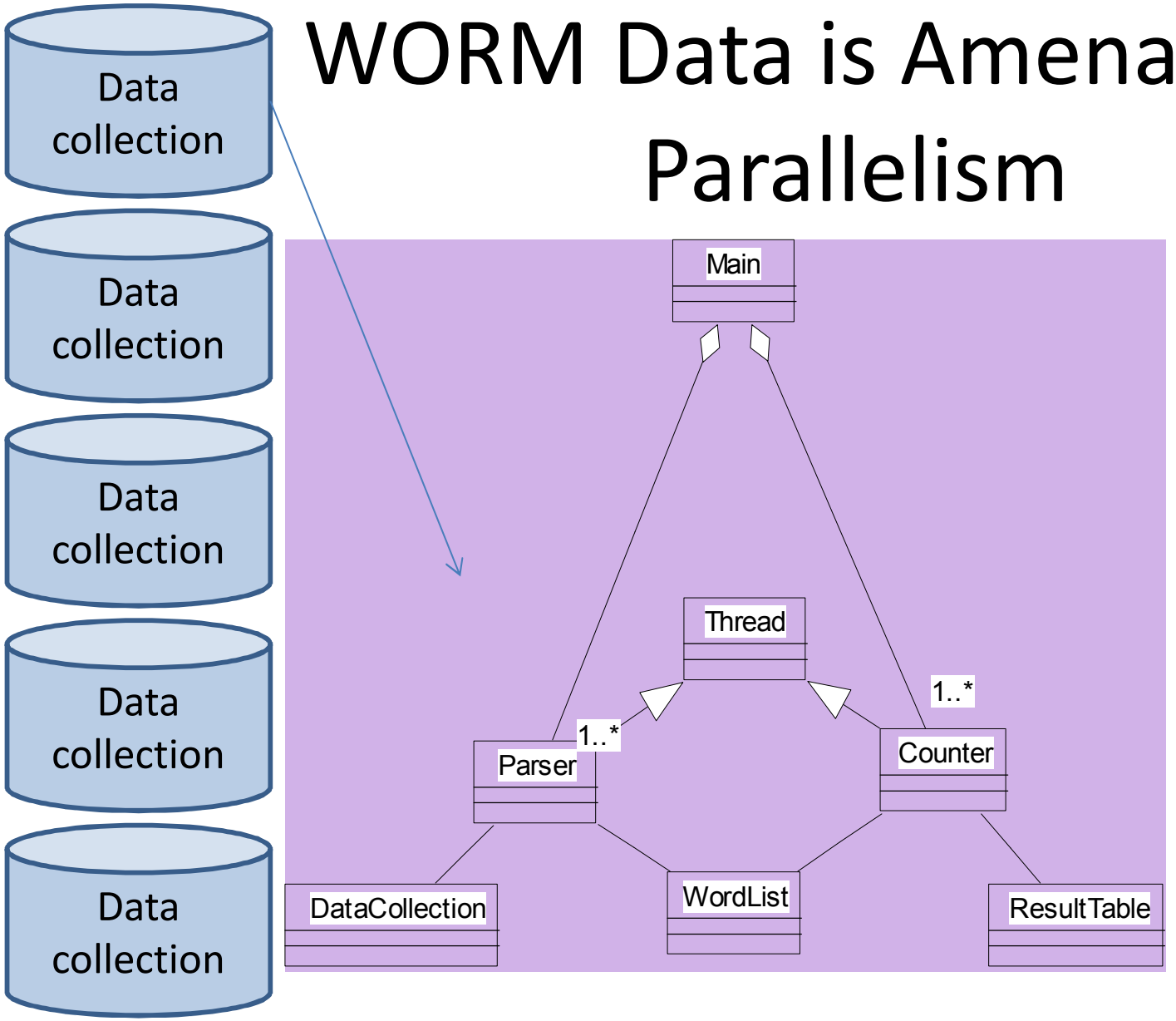
Write Once Read Many (WORM) data



web	2
weed	1
green	2
sun	1
moon	1
land	1
part	1

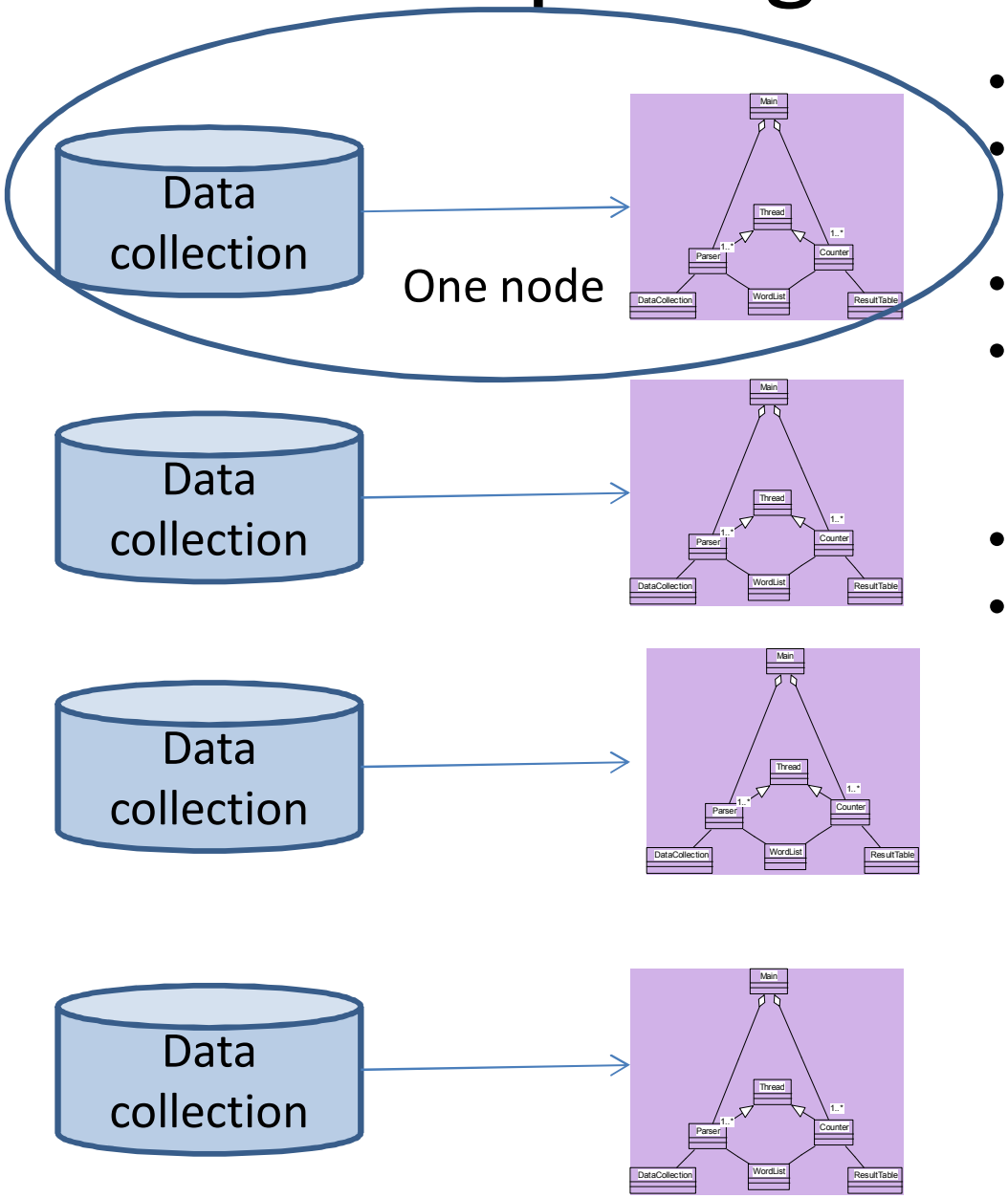
KEY	web	weed	green	sun	moon	land	part	web	green	.....
VALUE										

WORM Data is Amenable to Parallelism



- Data with WORM characteristics : yields to parallel processing
- Data without dependencies: yields to out of order processing

Divide and Conquer: Provision Computing at Data Location

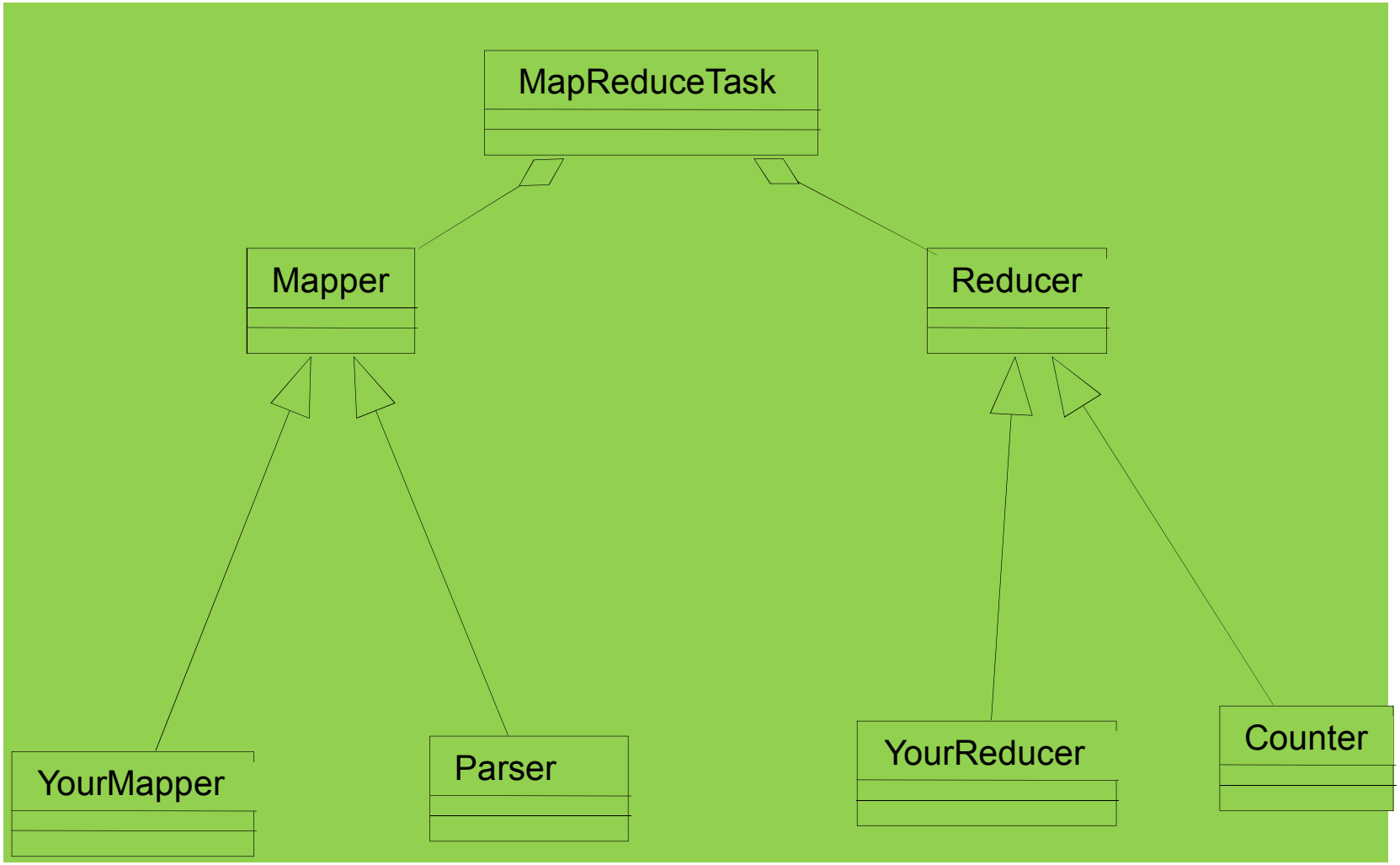


- Our parse is a mapping operation:
- MAP: input <key, value> pairs

- Our count is a reduceoperation:
- REDUCE: <key, value> pairs reduced

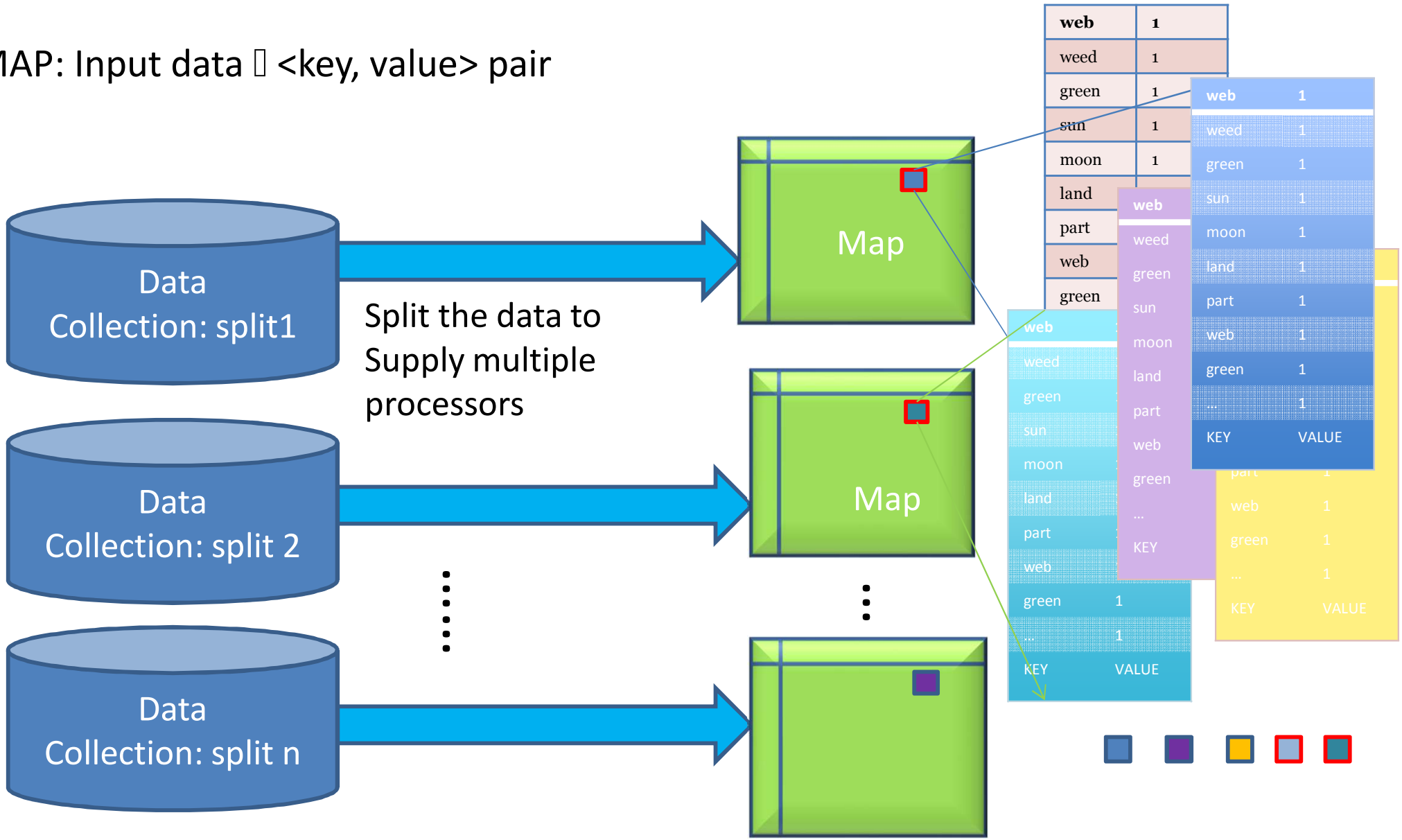
- Map/Reduce originated from Lisp
- But have different meaning here
  - Runtime adds distribution + fault tolerance + replication + monitoring + load balancing to your base application!

Mapper and Reducer



Map Operation

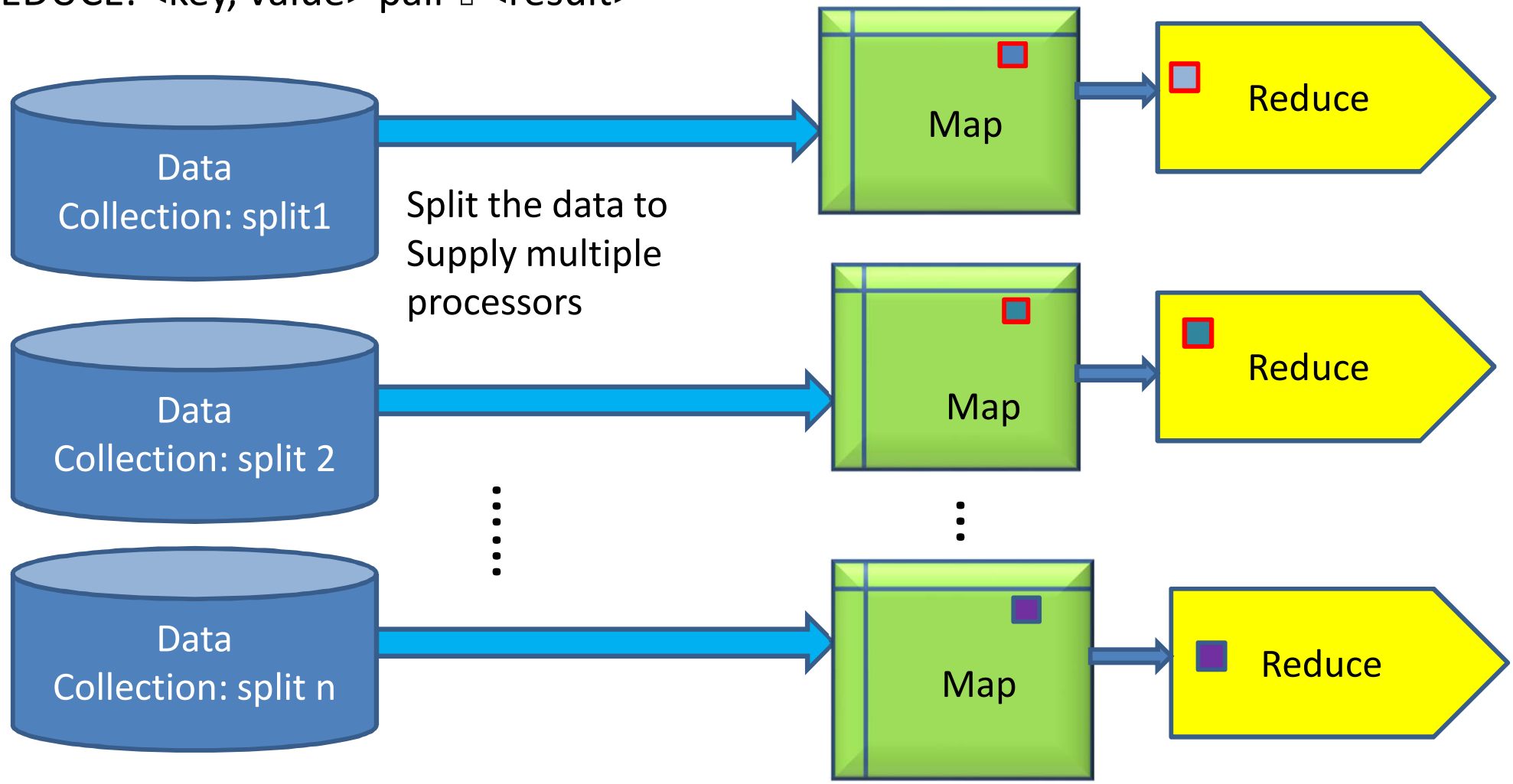
MAP: Input data <key, value> pair

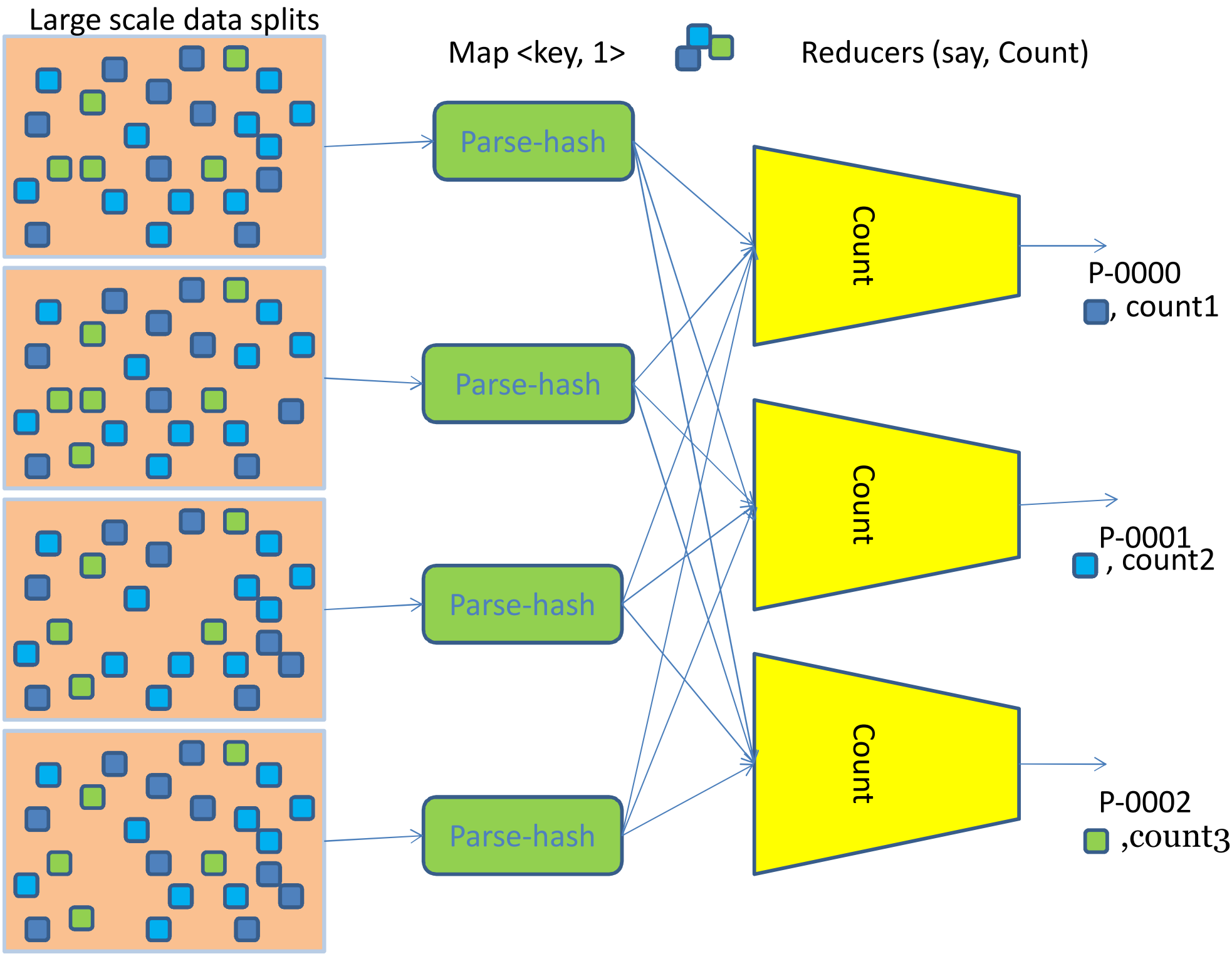


Reduce Operation

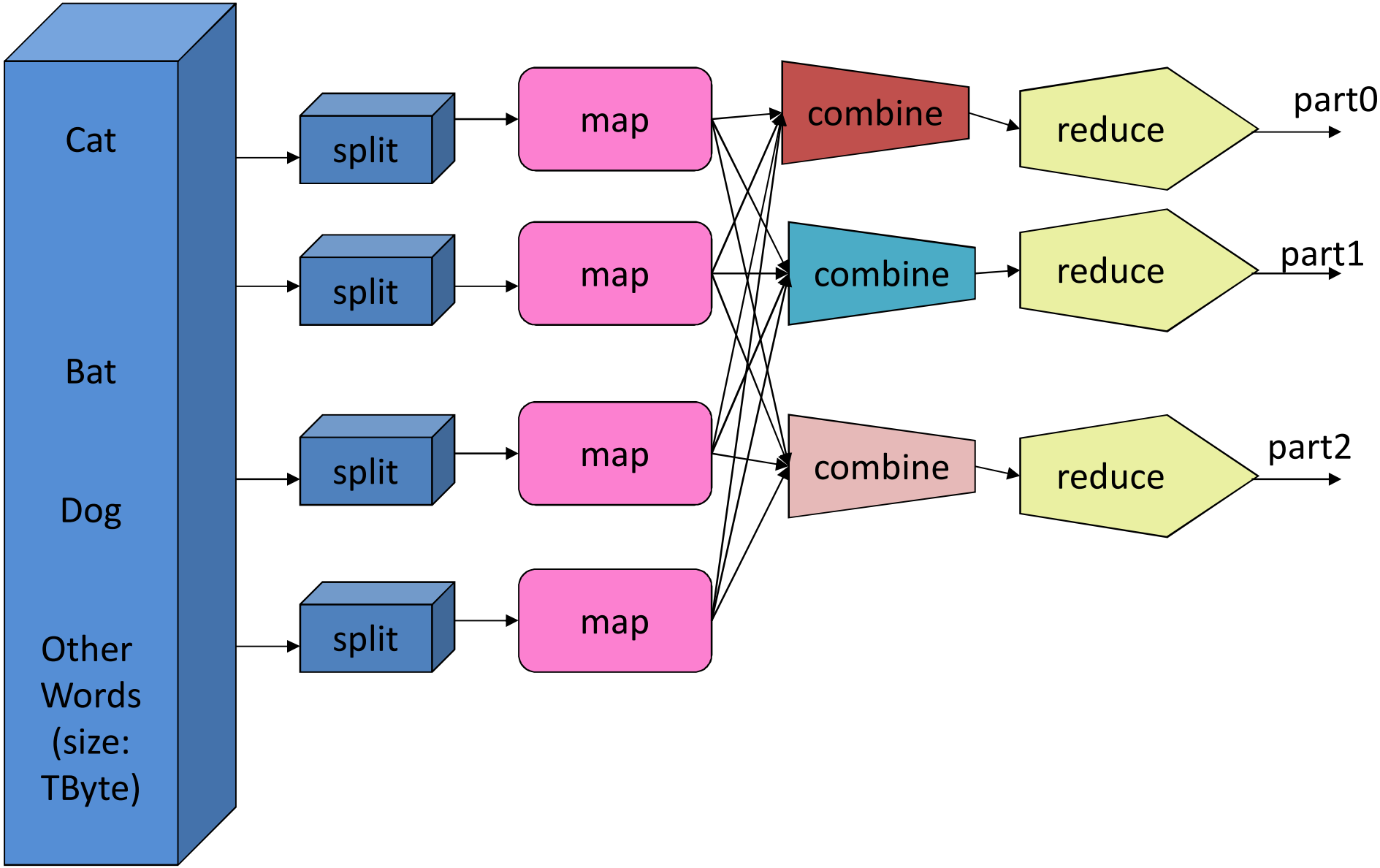
MAP: Input data <key, value> pair

REDUCE: <key, value> pair <result>




Large scale data splits
Map <key, 1>
Reducers (say, Count)
Parse-hash
Parse-hash
Parse-hash
Parse-hash
Count
Count
Count
P-0000
, count1
P-0001
, count2
P-0002
,count3

Example


Cat
Bat
Dog
Other Words (size: TByte)
split
split
split
split
map
map
map
map
combine
combine
combine
reduce
reduce
reduce
part0
part1
part2

MapReduce programming model

- Determine if the problem is parallelizable and solvable using MapReduce
- Design and implement solution as Mapper classes and Reducer classes
- Compile the source code with hadoop core
- Package the code as jar executable
- Configure the application (job) as to the number of mappers and reducers (tasks), input and output streams
- Load the data (or use it on previously available data)
- Launch the job and monitor
- Study the result

MapReduce Characteristics

- Very large scale data: peta, exa bytes
- Write once and read many data: allows for parallelism without mutexes
- Map and Reduce are the main operations: simple code
- All the map should be completed before reduce operation starts
- Map and reduce operations are typically performed by the same physical processor
- Number of map tasks and reduce tasks are configurable
- Operations are provisioned near the data
- Commodity hardware and storage
- Runtime takes care of splitting and moving data for operations

“map reducable” problems

- Google uses it (we think) for wordcount, adwords, pagerank, indexing data
- Simple algorithms such as grep, text-indexing, reverse indexing
- Bayesian classification: data mining domain
- Facebook uses it for various operations: demographics
- Financial services use it for analytics
- Astronomy: Gaussian analysis for locating extra-terrestrial objects
- Expected to play a critical role in semantic web and web3.0

Hadoop

- At Google MapReduce operation are run on a special file system called Google File System (GFS) that is highly optimized for this purpose
- GFS is not open source
- Doug Cutting and Yahoo! reverse engineered the GFS and called it Hadoop Distributed File System (HDFS)
- The software framework that supports HDFS, MapReduce and other related entities is called the project Hadoop or simply Hadoop
- This is open source and distributed by Apache

Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

Credits

- Design Patterns, Gamma, et al.; Addison-Wesley, 1995; ISBN 0-201-63361-2; CD version ISBN 0-201-63498-8
- Douglas C. Schmidt