Parallel Programming

Prof. Paul Krause, University of Surrey
Lecture 9
Principles of Parallel Algorithm Design

For more detail, see: Ananth Grama, Anshul Gupta,
George Karypis, and Vipin Kumar, *Introduction to Parallel Computing*,
Pearson
Chapter 3

Chapter Overview: Algorithms and Concurrency

- Introduction to Parallel Algorithms
  - Tasks and Decomposition
  - Processes and Mapping
  - Processes Versus Processors
- Decomposition Techniques
  - Recursive Decomposition
  - Data Decomposition
  - Exploratory Decomposition
- Characteristics of Tasks and Interactions
  - Task Generation, Granularity, and Context
  - Characteristics of Task Interactions.

Preliminaries: Decomposition, Tasks, and Dependency Graphs

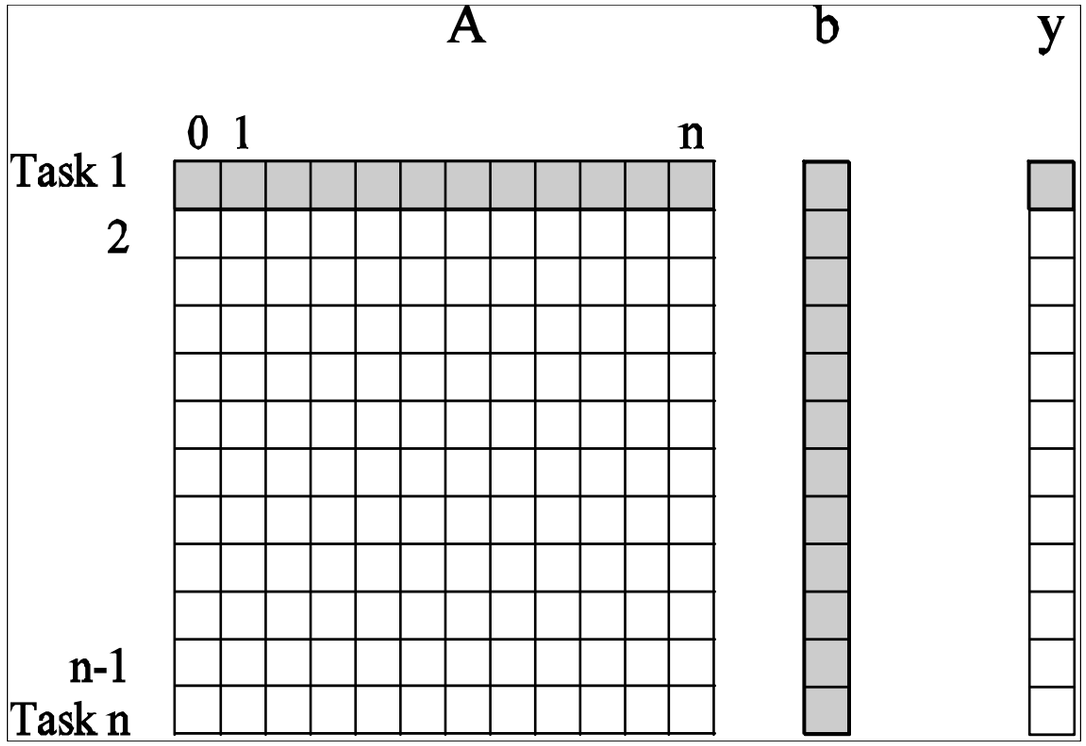
- The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently

- A given problem may be decomposed into tasks in many different ways.

- Tasks may be of same, different, or even indeterminate sizes.

- A decomposition can be illustrated in the form of a directed graph with nodes corresponding to tasks and edges indicating that the result of one task is required for processing the next. Such a graph is called a *task dependency graph*.

Example: Multiplying a Dense Matrix with a Vector



Computation of each element of output vector ***y*** is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into ***n*** tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

Observations: While tasks share data (namely, the vector ***b***), they do not have any control dependencies - i.e., no task needs to wait for the (partial) completion of any other. All tasks are of the same size in terms of number of operations. *Is this the maximum number of tasks we could decompose this problem into?*

Example: Database Query Processing

Consider the execution of the query:

MODEL = ``CIVIC'' AND YEAR = 2001 AND

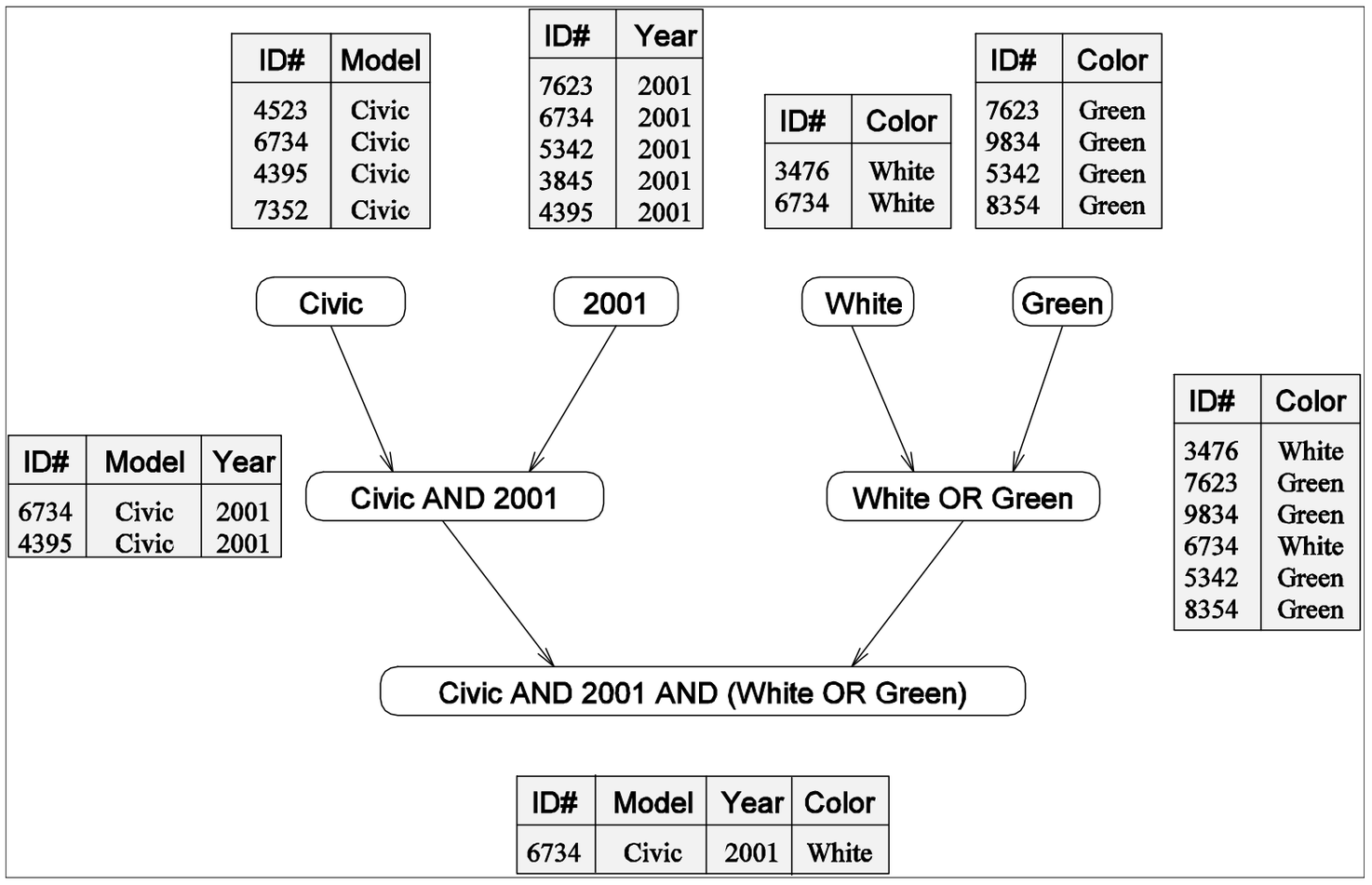
(COLOR = ``GREEN'' OR COLOR = ``WHITE)

on the following database:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Example: Database Query Processing

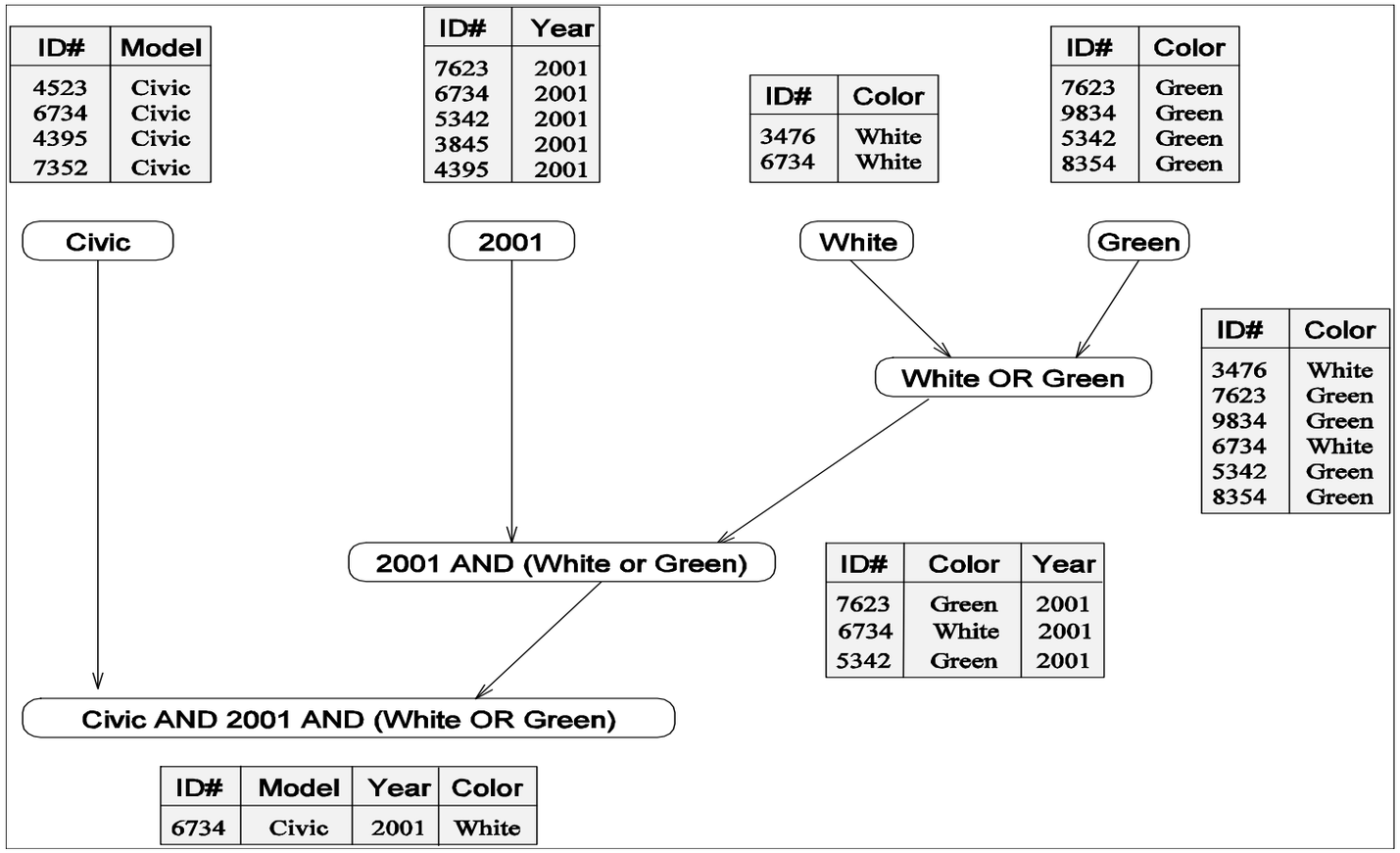
The execution of the query can be divided into subtasks in various ways. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.



Decomposing the given query into a number of tasks. Edges in this graph denote that the output of one task is needed to accomplish the next.

Example: Database Query Processing

Note that the same problem can be decomposed into subtasks in other ways as well.



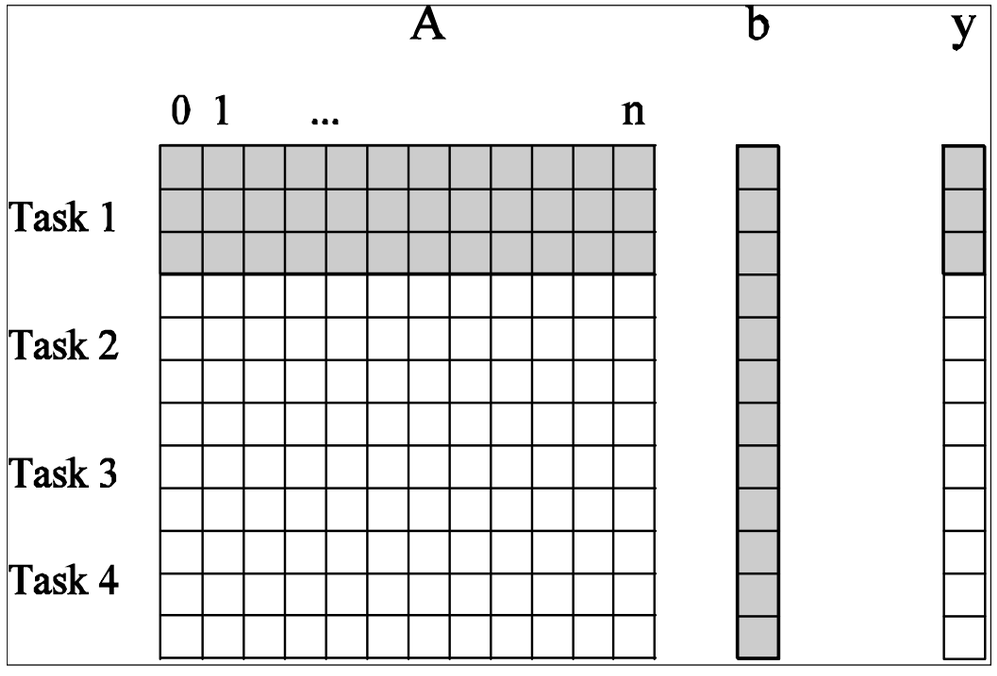
An alternate decomposition of the given problem into subtasks, along with their data dependencies.

Different task decompositions may lead to significant differences with respect to their eventual parallel performance.

Granularity of Task Decompositions

The number of tasks into which a problem is decomposed determines its granularity.

Decomposition into a large number of tasks results in fine-grained decomposition and that into a small number of tasks results in a coarse grained decomposition.



A coarse grained counterpart to the dense matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.

Degree of Concurrency

- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.

- Since the number of tasks that can be executed in parallel may change over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution. *What is the maximum degree of concurrency of the database query examples?*

- The *average degree of concurrency* is the average number of tasks that can be processed in parallel over the execution of the program. *Assuming that each task in the database example takes identical processing time, what is the average degree of concurrency in each decomposition?*

- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

Critical Path Length

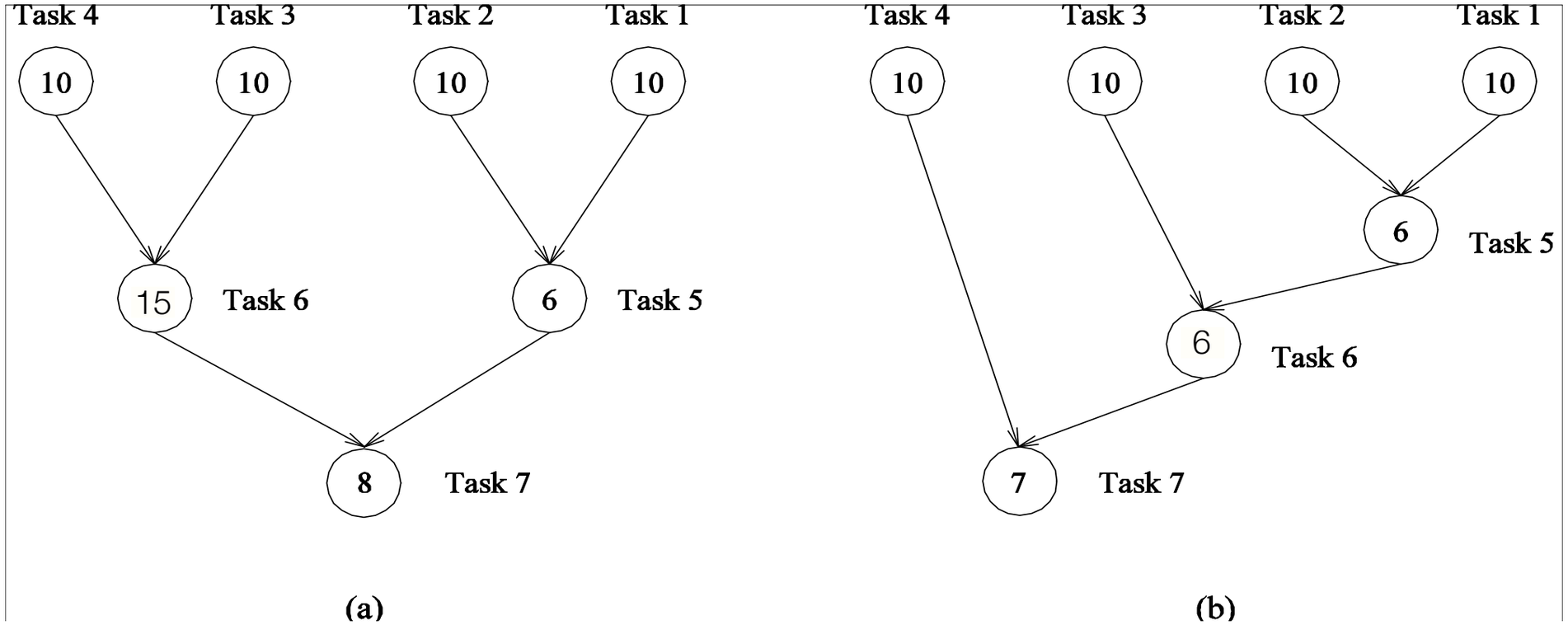
- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.

- The longest such path determines the shortest time in which the program can be executed in parallel.

- The length of the longest path in a task dependency graph is called the critical path length.

Critical Path Length

Consider the task dependency graphs of the two database query decompositions:



What are the critical path lengths for these two decompositions?
What is the average degree of concurrency?

Limits on Parallel Performance

- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity.

- There is an inherent bound on how fine the granularity of a computation can be. *For example, in the case of multiplying a dense matrix with a vector, there can be no more than* ***(n^2)*** *concurrent tasks.*

- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds.

Task Interaction Graphs

- Subtasks generally exchange data with others in a decomposition. For example, even in the trivial decomposition of the dense matrix-vector product, if the vector is not replicated across all tasks, they will have to communicate elements of the vector.

- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a *task interaction graph*.

- Note that *task interaction graphs* represent data dependencies, whereas *task dependency graphs* represent control dependencies.

Task Interaction Graphs, Granularity, and Communication

In general, if the granularity of a decomposition is finer, the associated overhead (as a ratio of useful work associated with a task) increases.

However, if the data sets are sparse, we may obtain greater efficiency by using a coarser grained task decomposition.

Processes and Mapping

- In general, the number of tasks in a decomposition exceeds the number of processing elements available.

- For this reason, a parallel algorithm must also provide a mapping of tasks to processes.

Note: We refer to the mapping as being from tasks to processes, as opposed to processors. This is because typical programming APIs, as we shall see, do not allow easy binding of tasks to physical processors. Rather, we aggregate tasks into processes and rely on the system to map these processes to physical processors. We use processes, not in the UNIX sense of a process, rather, simply as a collection of tasks and associated data.

Processes and Mapping

- Appropriate mapping of tasks to processes is critical to the parallel performance of an algorithm.

- Mappings are determined by both the task dependency and task interaction graphs.

- Task dependency graphs can be used to ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance).

- Task interaction graphs can be used to make sure that processes need minimum interaction with other processes (minimum communication).

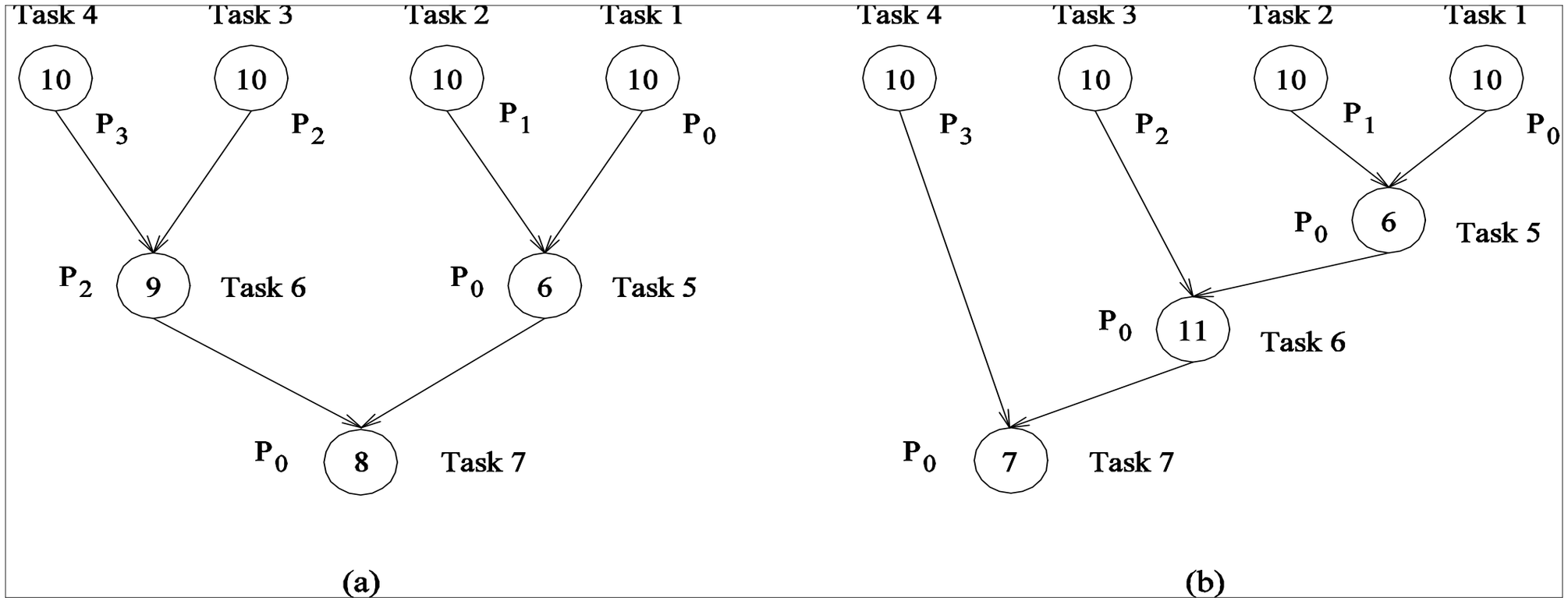
Processes and Mapping

An appropriate mapping must minimise parallel execution time by:

- Mapping independent tasks to different processes.
- Assigning tasks on the critical path to processes as soon as they become available.
- Minimising interaction between processes by mapping tasks with dense interactions to the same process.

  Note: These criteria often conflict with each other. For example, a decomposition into one task (or no decomposition at all) minimises interaction but does not result in a speedup at all! Can you think of other such conflicting cases?

Processes and Mapping: Example



Mapping tasks in the database query decomposition to processes. These mappings were arrived at by viewing the dependency graph in terms of levels (no two nodes in a level have dependencies). Tasks within a single level are then assigned to different processes.

Decomposition Techniques

So how does one decompose a task into various subtasks?

While there is no single recipe that works for all problems, we present a set of commonly used techniques that apply to broad classes of problems. These include:

- recursive decomposition
- data decomposition
- exploratory decomposition
- speculative decomposition

Recursive Decomposition

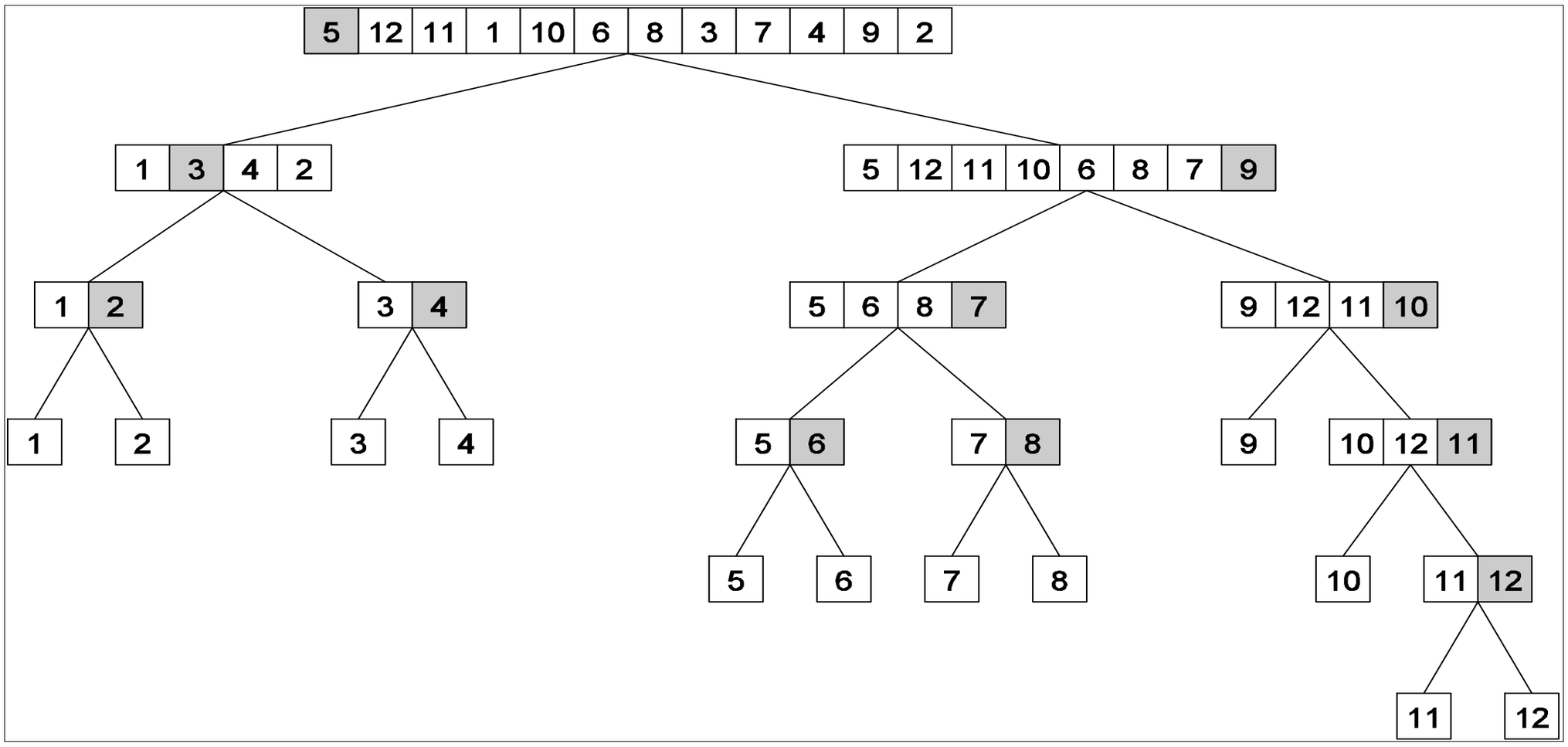
- Generally suited to problems that are solved using the divide-and-conquer strategy.

- A given problem is first decomposed into a set of sub-problems.

- These sub-problems are recursively decomposed further until a desired granularity is reached.

Recursive Decomposition: Example

A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.



In this example, once the list has been partitioned around the pivot, each sublist can be processed concurrently (i.e., each sublist represents an independent subtask). This can be repeated recursively.

Data Decomposition

- Identify the data on which computations are performed.
- Partition this data across various tasks.
- This partitioning induces a decomposition of the problem.

- Data can be partitioned in various ways - this critically impacts performance of a parallel algorithm.

Data Decomposition: Output Data Decomposition

- Often, each element of the output can be computed independently of others (but simply as a function of the input).

- A partition of the output across tasks decomposes the problem naturally.

Exploratory Decomposition

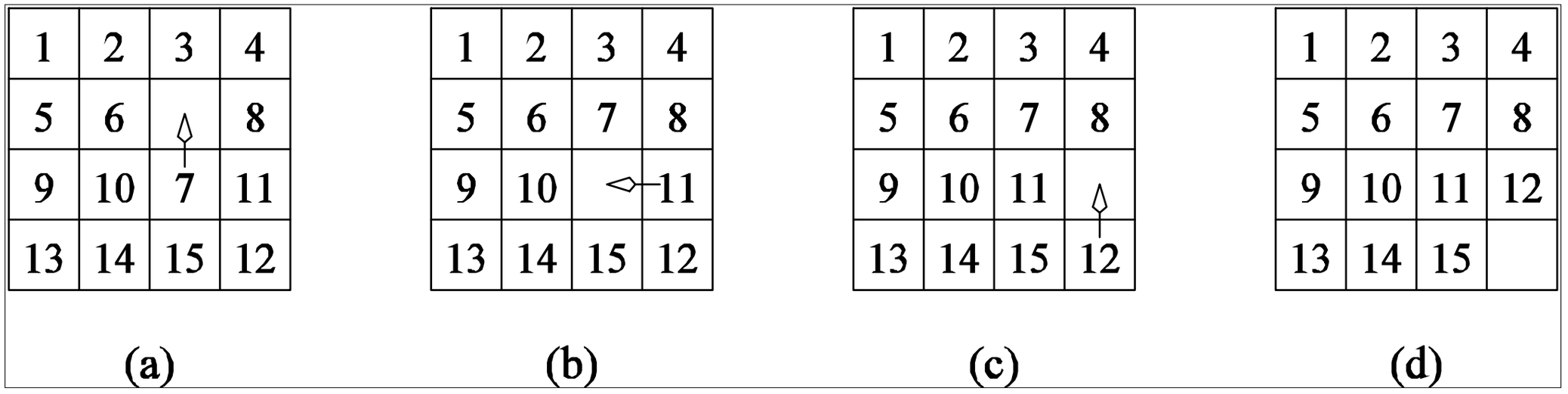
- In many cases, the decomposition of the problem goes hand-in-hand with its execution.

- These problems typically involve the exploration (search) of a state space of solutions.

- Problems in this class include a variety of discrete optimization problems, theorem proving, game playing, etc.

Exploratory Decomposition: Example

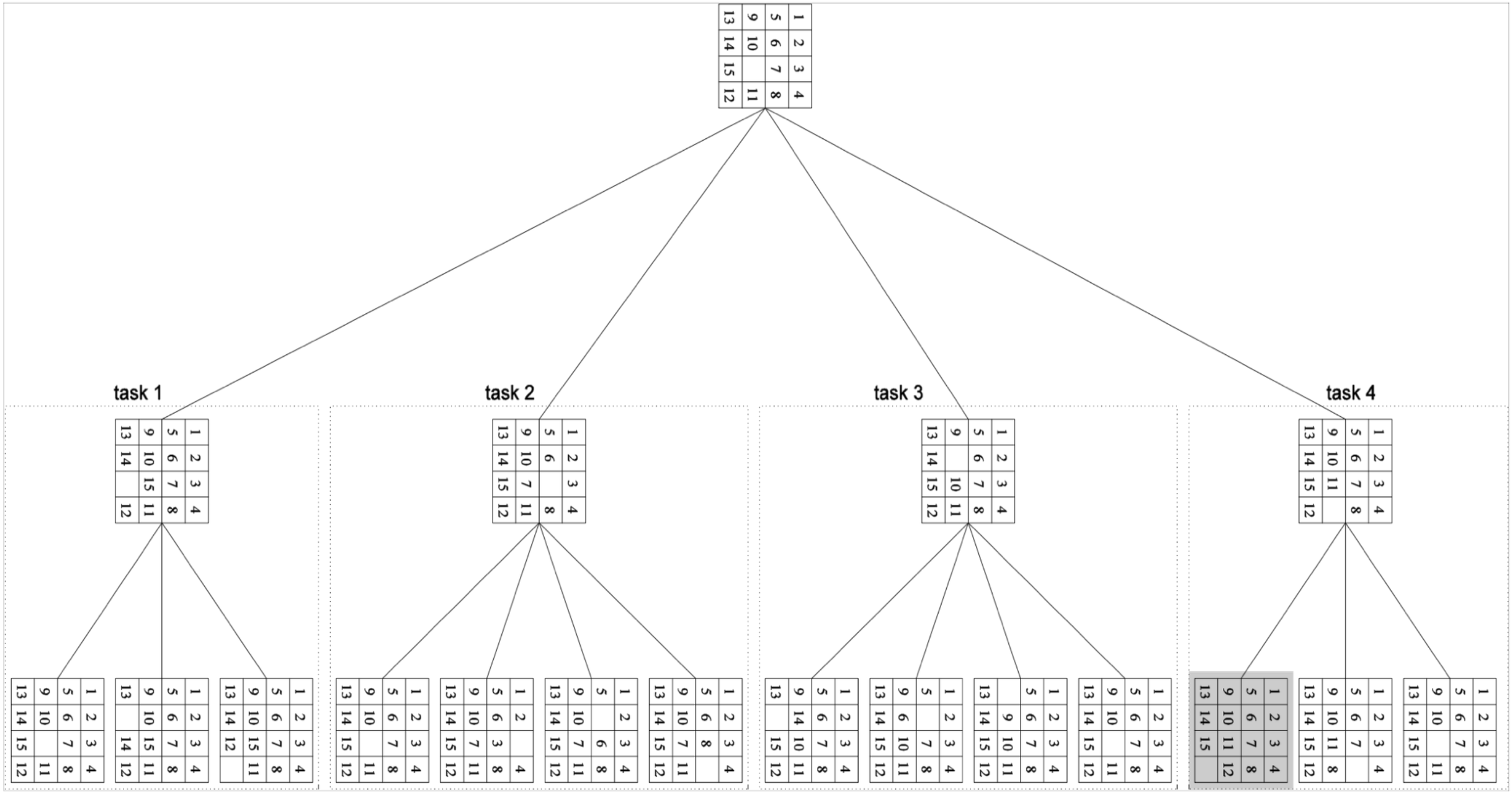
A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).



Of-course, the problem of computing the solution, in general, is much more difficult than in this simple example.

Exploratory Decomposition: Example

The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



Summary

- We have explored some general approaches to parallel algorithm design
- Now we will explore some specific examples: reduce, scan and histogram