Parallel and Distributed Computing
CS3006 (BCS-6C/6D)
Lecture 07

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

14 February, 2023

Previous Lecture

- Evaluating static interconnections
    - Cost, diameter, bisection width, arc connectivity
- Parallel Algorithm Design
    - Identification (of tasks)
    - Mapping (tasks to processes)
    - Data Partitioning
    - Defining access protocol, Synchronizing
- Tasks & Decomposition
    - Task-Dependency Graphs
        - Granularity of decomposition (depends on size of tasks)
        - Maximum and average degree of concurrency
    - Task-Interaction Graphs
        - Example of sparse-matrix multiplication

Principles of Parallel Algorithm Design

Task Interact Graph (Sparse-matrix multiplication**)**

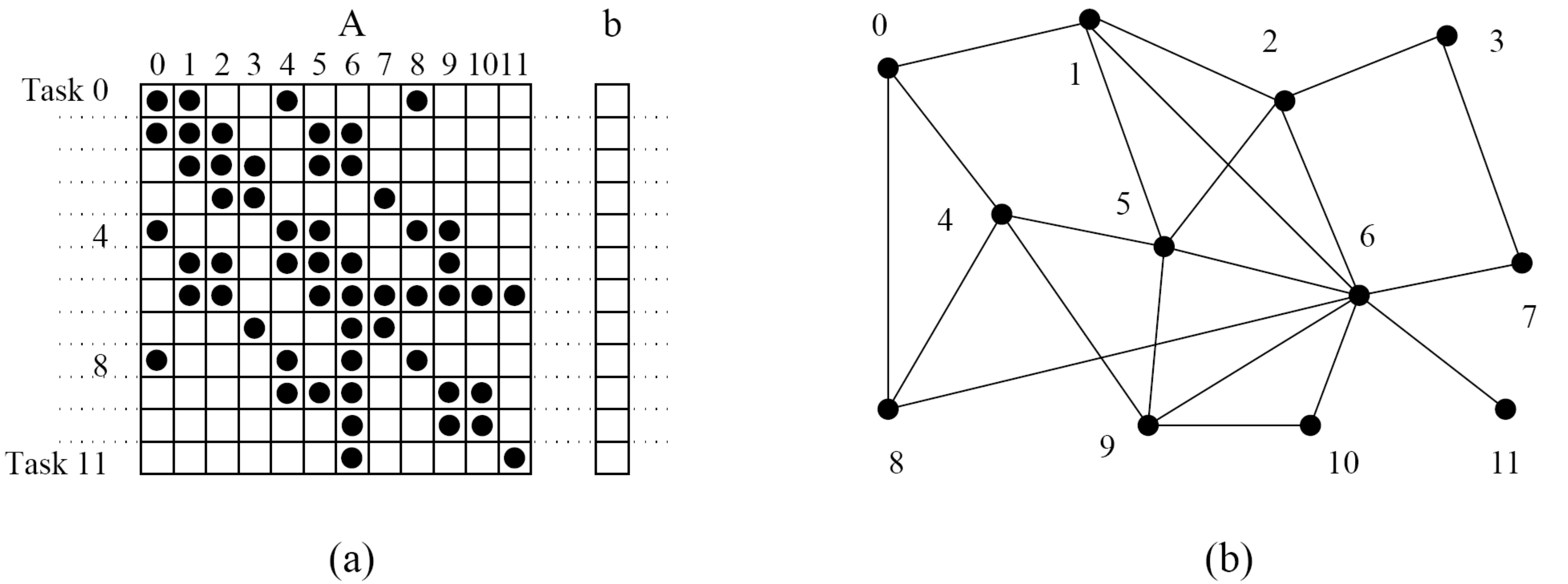


Figure 3.6 A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \le j \le 11, A[i,j] \neq 0} A[i, j].b[j]$.

Processes and Mapping

- The *logical processing* or *computing agent* that performs tasks is called a **process**.
- The *mechanism* by which tasks are *assigned to processes* for execution is called ***mapping.***
- *Multiple tasks* can be mapped on a *single process*
- *Independent tasks* should be mapped onto *different processes*
- *Tasks* with high *mutual-interactions* should be mapped onto a *single process*

Processes and Mapping

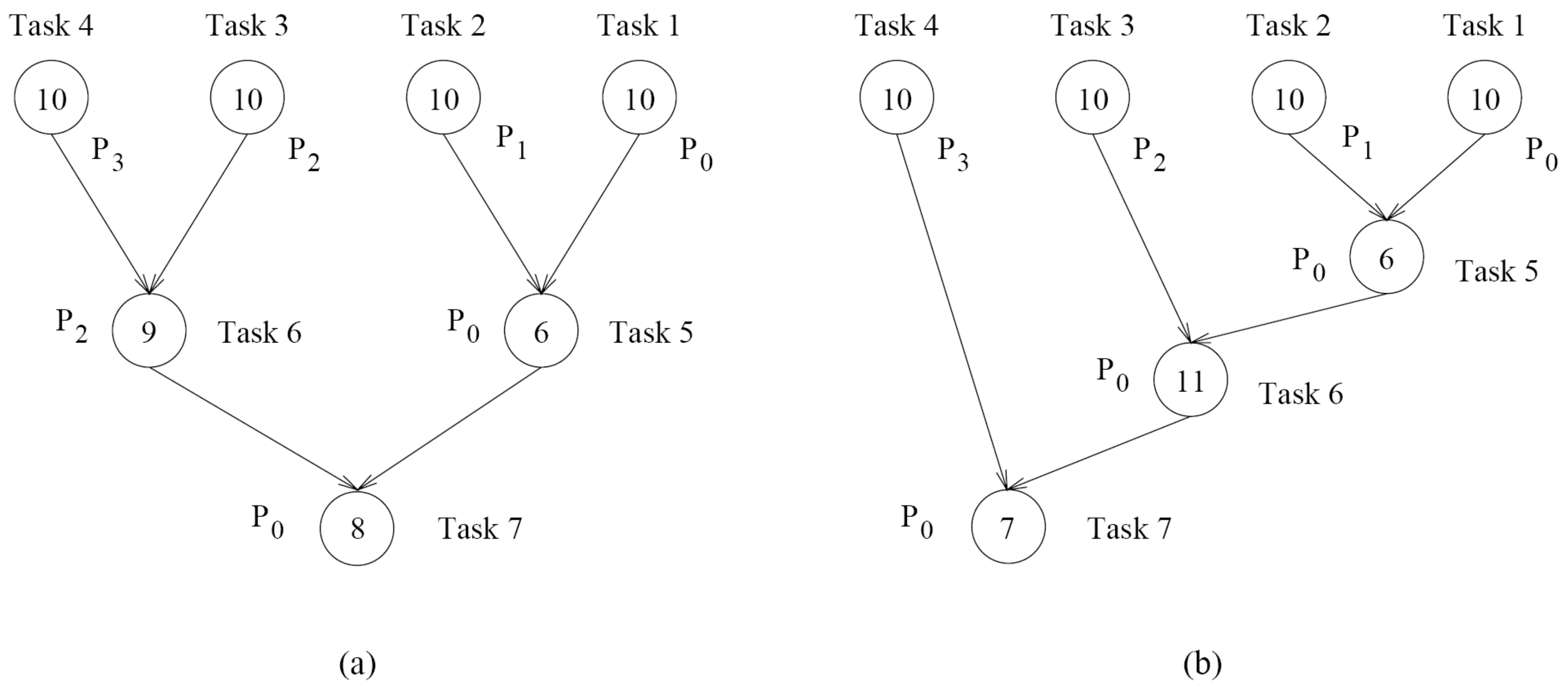
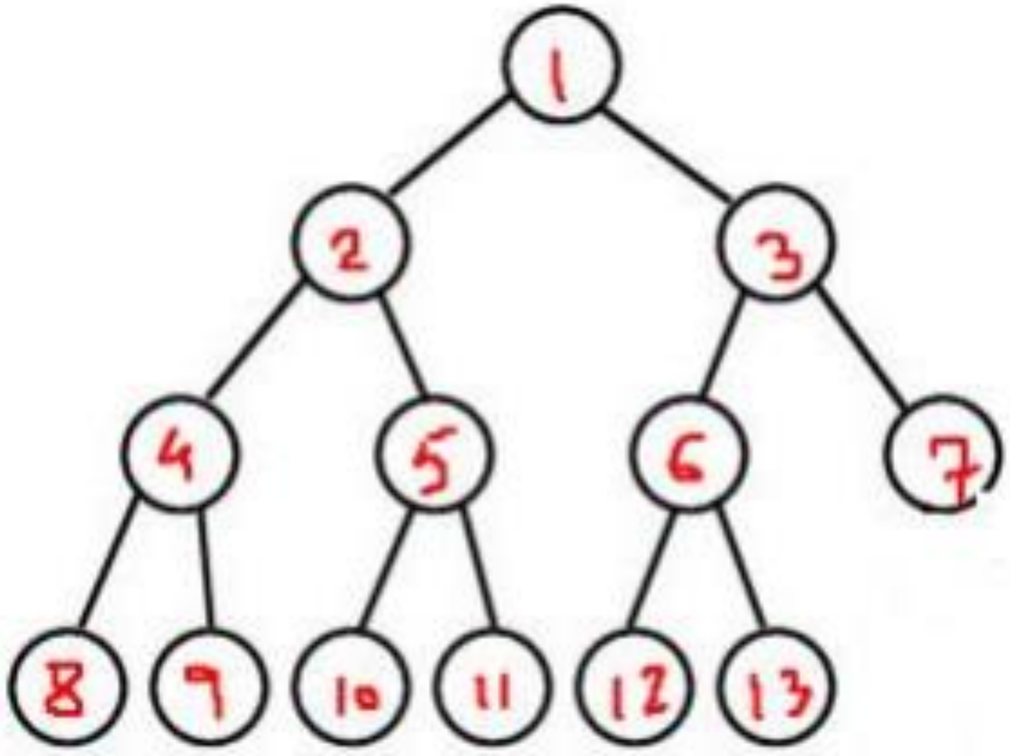


Figure 3.7 Mappings of the task graphs of Figure 3.5 onto four processes.

Processes vs. Processors

- **Processes** are logical computing agents that perform tasks
- **Processors** are the hardware units that physically perform computations
- Depending on the problem, *multiple processes* can be mapped onto a *single processor*
- But, in most of the cases, there is a *one-to-one correspondence between processors and processes*
- So, we assume that there are *as many processes* as the number of *physical CPUs* on the parallel computer

Exercise:



- For the task graph given above, determine:
- Maximum degree of concurrency
- Critical path
- Maximum possible speedup assuming large number of process are available
- Minimum number of processes needed to obtain the maximum possible speedup
- Maximum speedup if number of processes are limited to 4

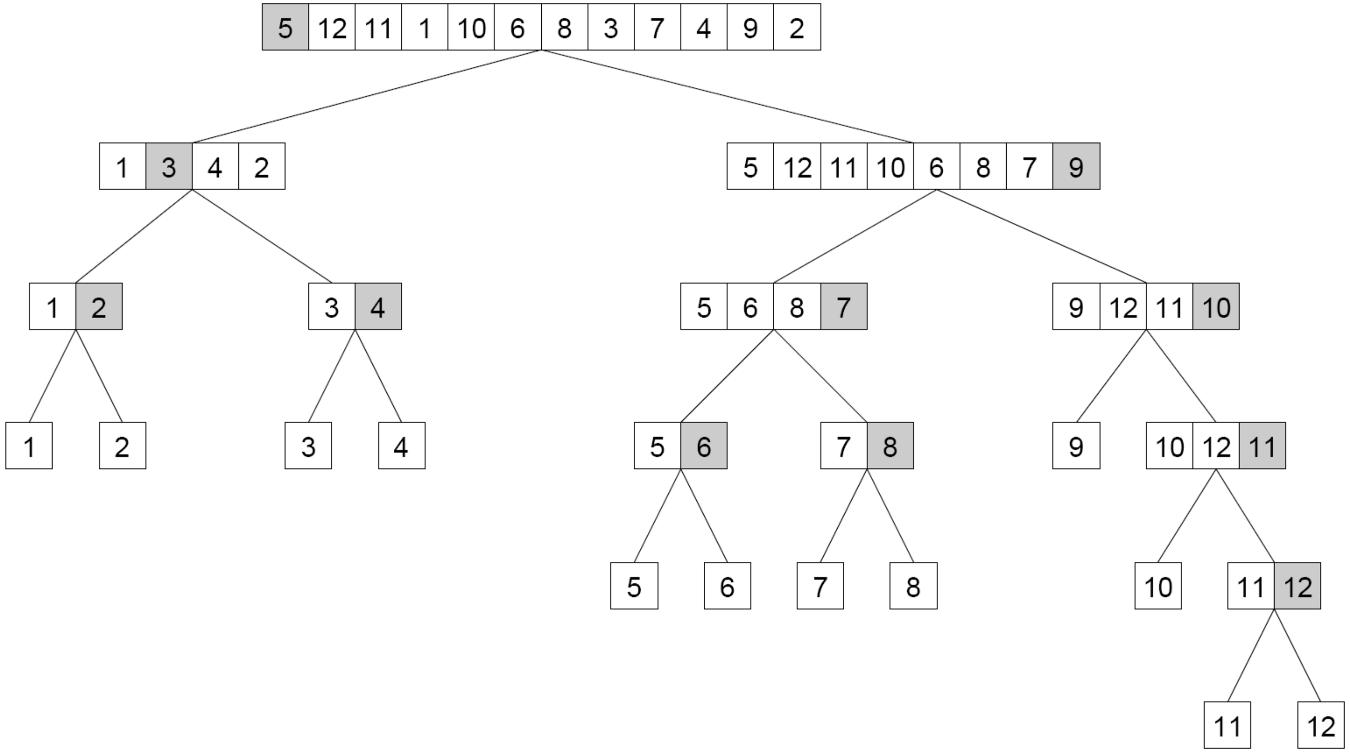
Decomposition Techniques

- The process of decomposing larger problems into smaller tasks for concurrent executions is known as *decomposition*.
- The techniques that facilitate this decomposition are known as *decomposition techniques*.
- Common techniques:
  - Recursive
  - Data-decomposition
  - Exploratory decomposition
  - Speculative decomposition
  - Hybrid
- *Recursive* and *data decompositions* are relatively *general purpose*
- *Exploratory* and *speculative* are *special purpose* in nature

Recursive Task Decomposition

- Recursive decomposition is a method for inducing concurrency in the problems that can be solved using a *divide and conquer strategy*
- It *divides* each problem into a *set of independent sub-problems*
- Each one of these *subproblems* is solved by *recursively applying* a similar *division* into *smaller subproblems* followed by a *combination of their results*
- A *natural concurrency* exists as *different subproblems* can be *solved concurrently*.

Recursive Decomposition



At each level and for each vector

1. Select a pivot
2. Partition set around pivot
3. Recursively sort each subvector

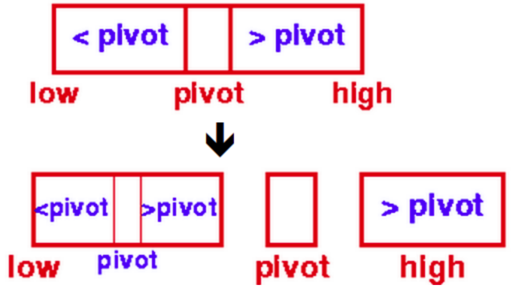


Figure 3.8 The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.

Recursive Decomposition

Modifying simple problem to support recursive decomposition

```
procedure SERIAL_MIN (A, n)
begin
min = A[0];
for i := 1 to n − 1 do
    if (A[i] < min) min := A[i];
endfor;
return min;
end SERIAL_MIN
```

Algorithm 3.1 A serial program for finding the minimum in an array of numbers A of length n.

```
procedure RECURSIVE_MIN (A, n)
begin
if (n = 1) then
    min := A[0];
else
    lmin := RECURSIVE_MIN (A, n/2);
    rmin := RECURSIVE_MIN (&(A[n/2]), n − n/2);
    if (lmin < rmin) then
        min := lmin;
    else
        min := rmin;
    endelse;
endelse;
return min;
end RECURSIVE_MIN
```

Algorithm 3.2 A recursive program for finding the minimum in an array of numbers A of length n.

Recursive Decomposition

Modifying simple problem to support recursive decomposition

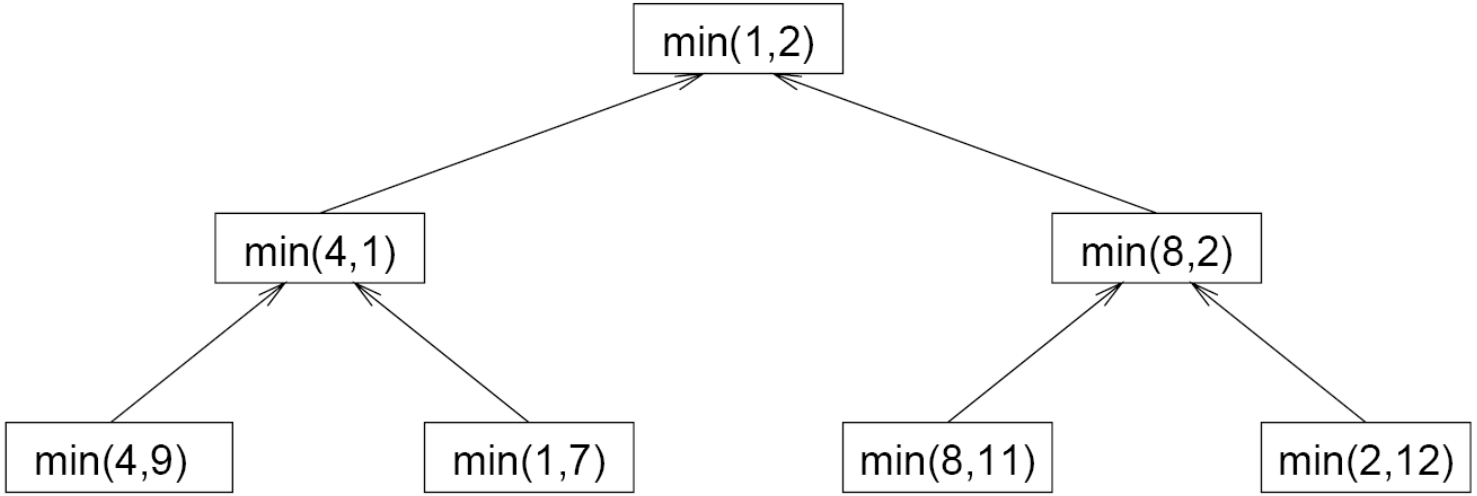


Figure 3.9 The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.

Data Decomposition

- Powerful and commonly used method
- Two step procedure:
  - *Partition data* on which computation is to be performed
  - This data partitioning is used to *induce* a *partitioning of the computations* into tasks.
- Partitioning output data
  - Used where each element of the output *can be computed independently* of others *as a function of the input*.
  - Partitioning of the output data *automatically induces a decomposition* of the problems into tasks
  - Each task is assigned *the work of computing a portion of the output*

Data Decomposition (Partitioning Output Data)

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Figure 3.10 (a) Partitioning of input and output matrices into 2×2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

Data Decomposition (Partitioning Output Data)

Decomposition I	**Decomposition II**
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

Figure 3.11 Two examples of decomposition of matrix multiplication into eight tasks.

Data Decomposition
(Partitioning Output Data)

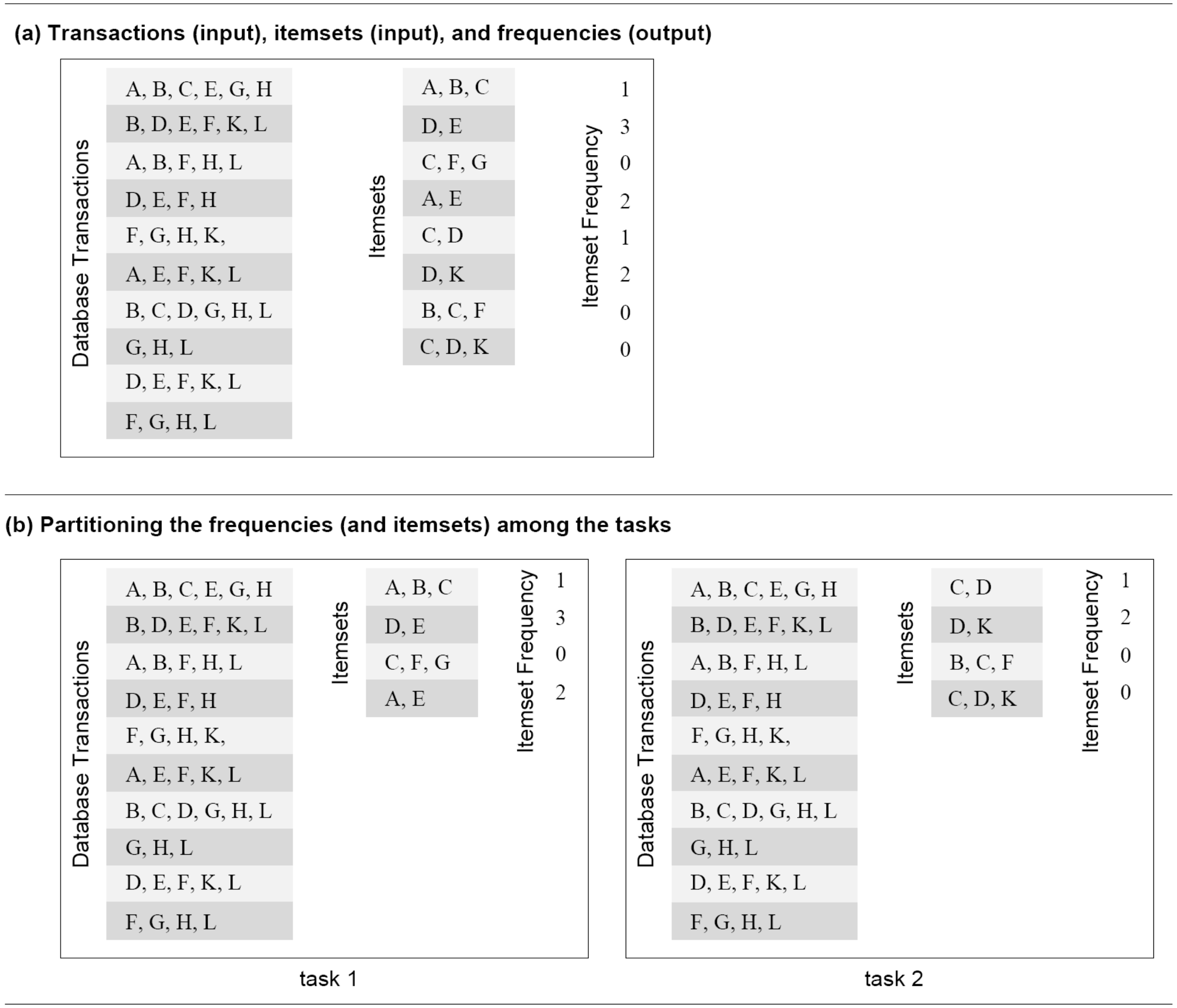


Figure 3.12 Computing itemset frequencies in a transaction database.

Decomposition Techniques

Data Decomposition

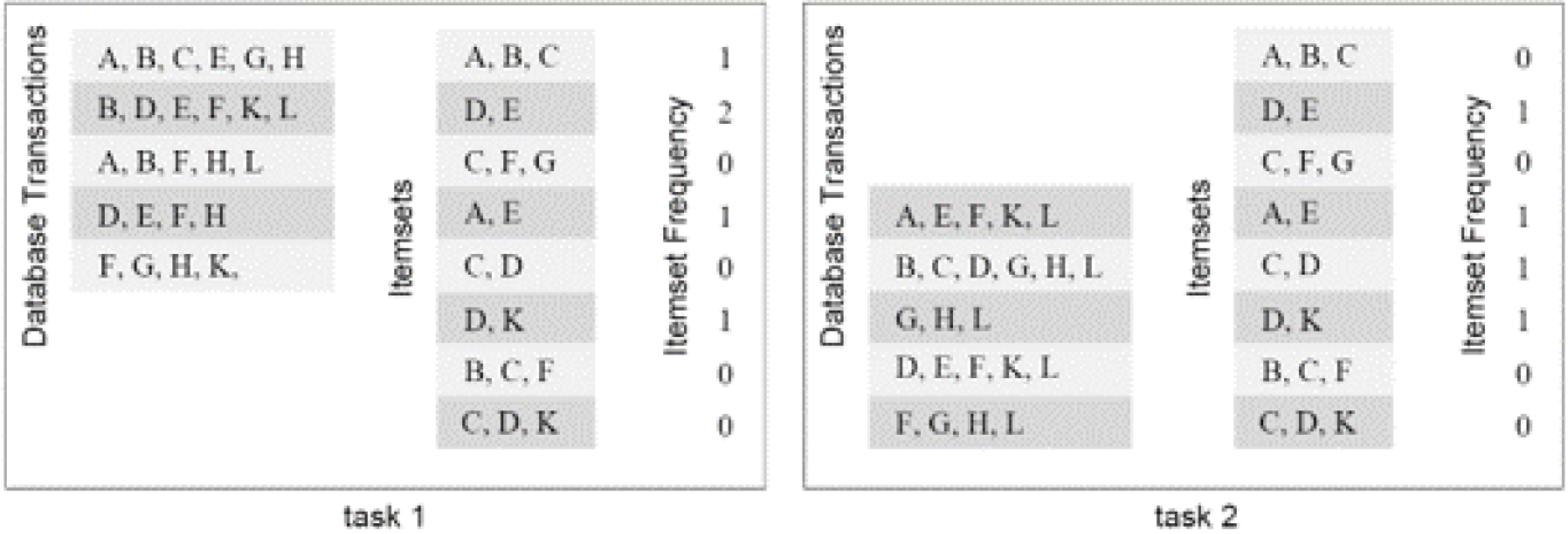
Partitioning *input* data

- In many algorithms, it is not possible or desirable to *partition the output data*.
    - The output may be a *single unknown value*.
    - Such as in case of *finding sum*, *minimum*, *maximum* or *frequencies of a number*.
- It is sometimes possible to *partition the input data*, and then use this partitioning to *induce concurrency*
- A *task is created for each partition of the input data* and this task performs *as much computation as possible using this local data*
- Then *local solutions are combined* to generate a *global solution*

Decomposition Techniques

Partitioning input data

- (a) Partitioning the transactions among the tasks



Decomposition Techniques

Data Decomposition

Partitioning *both input and output* data

- Consider the problems where output data-partitioning is possible
- Here, partitioning the input also, can offer additional concurrency
- The next example shows *4-way decomposition of the previous example* based on both input-output partitioning.

Decomposition Techniques

Partitioning *both input and output* data

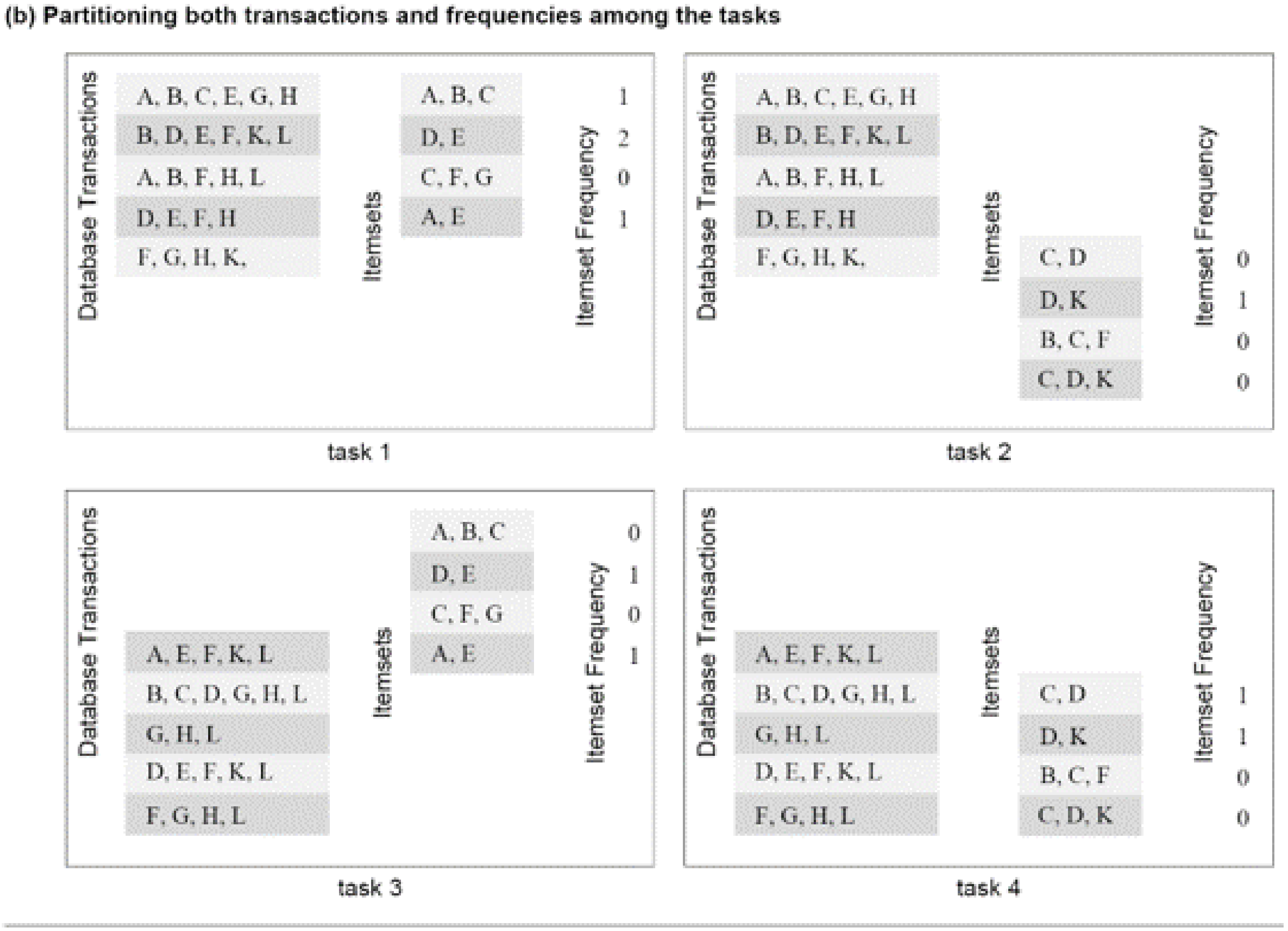


Figure 3.13 Some decompositions for computing itemset frequencies in a transaction database.

Decomposition Techniques

Partitioning *both intermediate data*

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

Task 01: $D_{1,1,1} = A_{1,1} B_{1,1}$
Task 02: $D_{2,1,1} = A_{1,2} B_{2,1}$
Task 03: $D_{1,1,2} = A_{1,1} B_{1,2}$
Task 04: $D_{2,1,2} = A_{1,2} B_{2,2}$
Task 05: $D_{1,2,1} = A_{2,1} B_{1,1}$
Task 06: $D_{2,2,1} = A_{2,2} B_{2,1}$
Task 07: $D_{1,2,2} = A_{2,1} B_{1,2}$
Task 08: $D_{2,2,2} = A_{2,2} B_{2,2}$
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$
Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$
Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Figure 3.15 A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

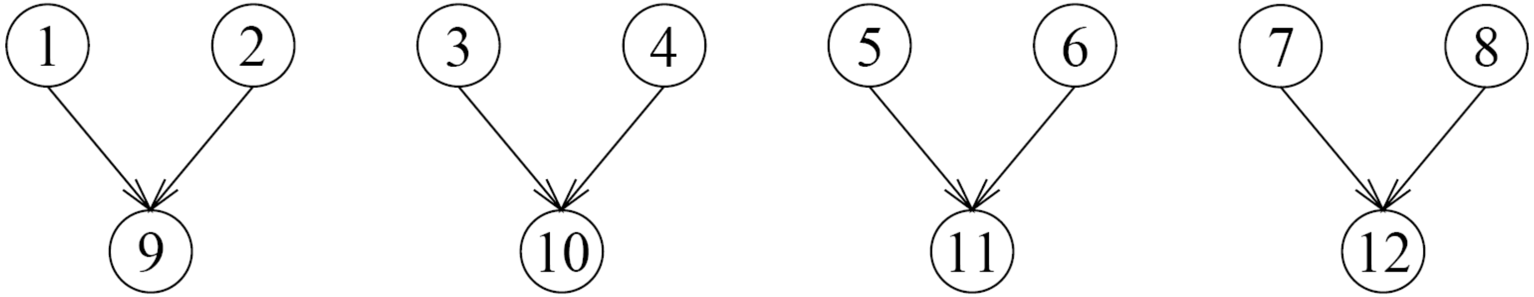


Figure 3.16 The task-dependency graph of the decomposition shown in Figure 3.15.

Decomposition Techniques

Partitioning *both intermediate data*

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$
Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$
Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$
Task 04: $D_{2,1,2} = A_{1,2}B_{2,2}$
Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$
Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$
Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$
Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$
Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$
Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Figure 3.15 A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

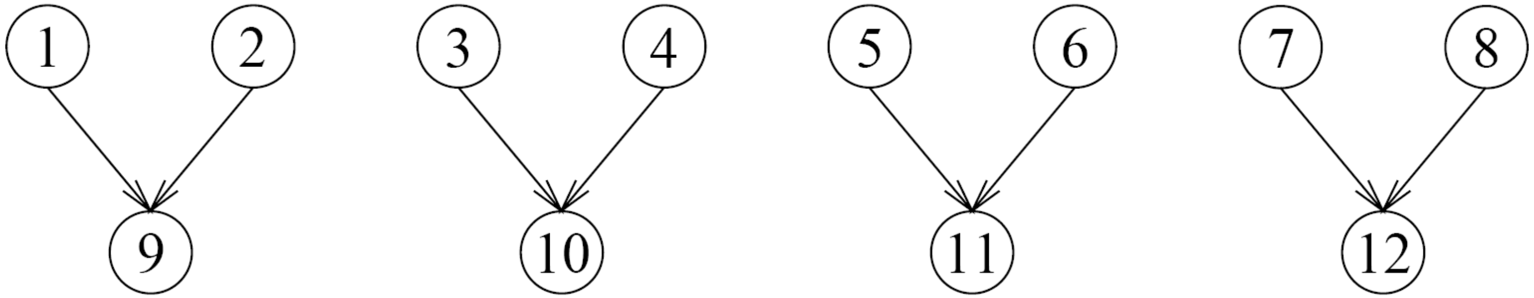


Figure 3.16 The task-dependency graph of the decomposition shown in Figure 3.15.

Decomposition Techniques

Owner Compute Rule

- Task decomposition based on data-partitioning is widely known as *owner compute rule*.
- *Two types of partitioning* hence, two definitions:

1. If we assign *partitions of the input data to tasks*:
    - The rule means that a task performs all the computations that can be done using this data
2. If we assign *partition of the output data to the tasks*:
    - The rule means that a task computes all the data in the partition assigned to it (portion of the output).

Decomposition Techniques

3. Exploratory Decomposition

- Specially used to decompose the problems having underlying computation *like search-space exploration*.
- Steps:
    1. Partition the *search space into smaller parts*
    2. Search each one of these parts concurrently, *until the desired solutions* are found.

Decomposition Techniques

3. Exploratory Decomposition

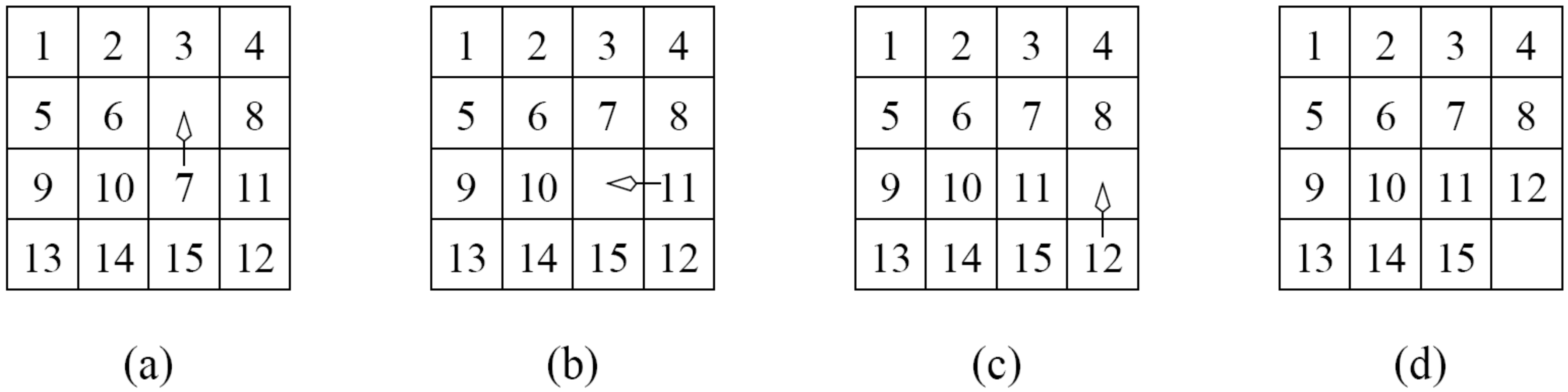


Figure 3.17 A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.

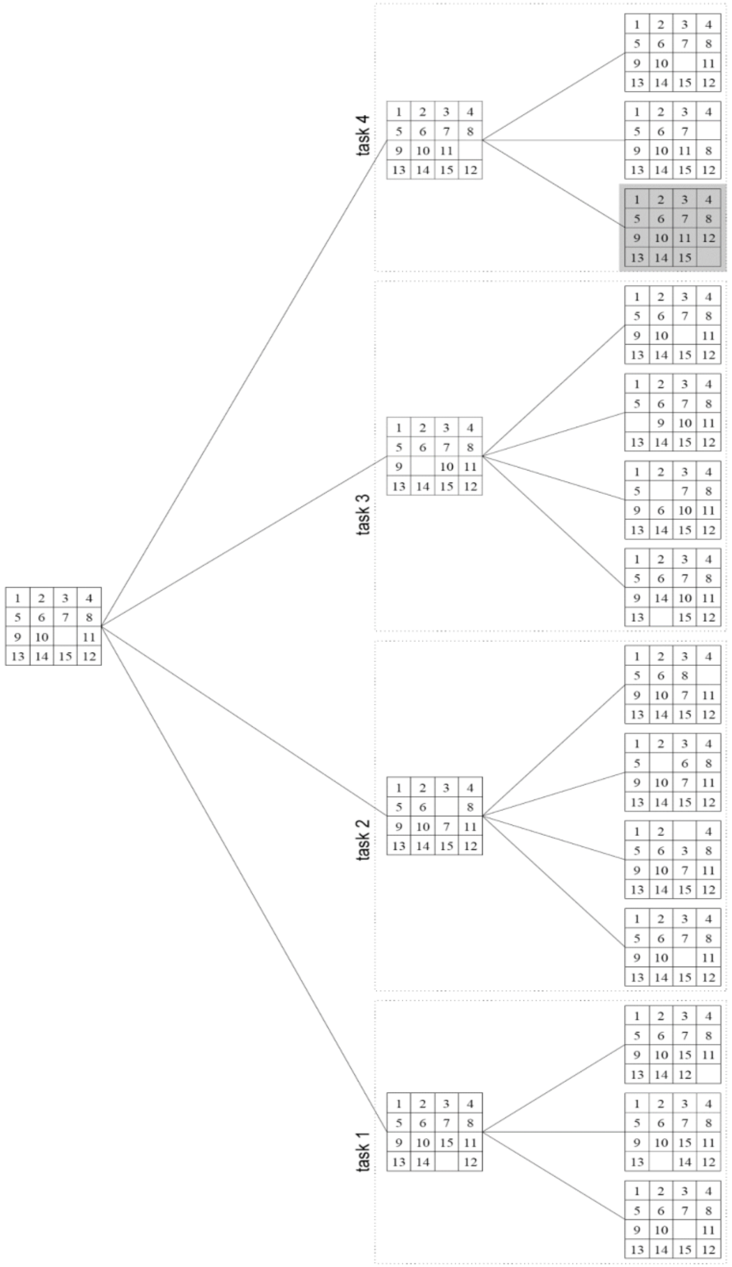


Figure 3.18 The states generated by an instance of the 15-puzzle problem.

Sources

- Slides of Dr. Rana Asif Rahman & Dr. Haroon Mahmood, FAST
- (Chapter 2) Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). Introduction to parallel computing (Vol. 110). Redwood City, CA: Benjamin/Cummings.
- Quinn, M. J. Parallel Programming in C with MPI and OpenMP,(2003).