

# Module 3

## Decomposition Techniques

# Preliminaries

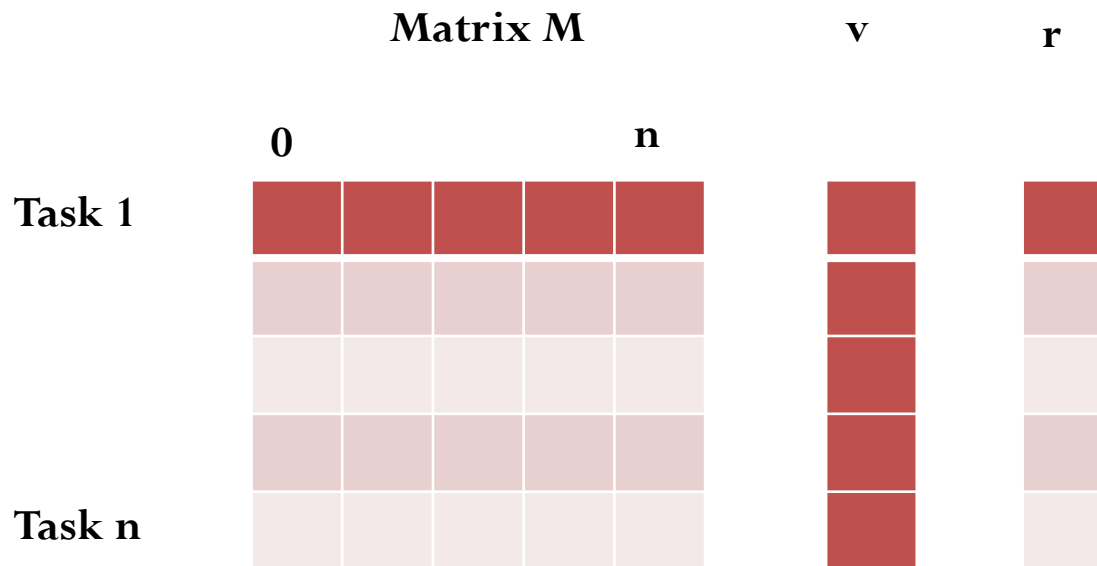
- Dividing a computation into smaller tasks for parallel execution is called as decomposition
- The sub division of main computation by means of decomposition as decided by the programmer is represented as tasks
- Concurrent execution of multiple tasks is the preferred solution for reducing the overall execution time
- A task could be of any arbitrary size
- The decomposed problem into multiple task need not to be of the same size. It could vary as per programmer's decision.

# Task Dependency Graph

- It is the abstraction used for indicating task dependencies and the relative execution order.
- It is a directed acyclic graph where the nodes representing tasks and edges representing dependencies between them.
- The task representing a node could be executed only if all the tasks connected by incoming edges have completed.
- The task dependency graphs could also be disconnected and the edge set of it could be empty in the case of matrix multiplication where each task corresponds to entries subset of the final product vector matrix.

# Example 1: Multiplication of $n \times n$ matrix

- Multiplication of Matrix  $M$  with vector  $v$  yields the resultant output vector  $r$ . The computation of each  $M[i]$  is considered as a task.



# Example 2: Database Query Processing

- Consider the relational database of flowers sales. Each row in the table represents a record which contains data related to a particular flower as its ID, Category, Year of Sales, Color and Sold\_Quantity.
- Eg: Model= “ Civic” AND Year = “2002” AND ( Color = “White” or Color = “Green”)
- Vehicles database

ID	Model	Year	Color	Dealer	Price
1	Civic	2002	Green	NY	\$15,000
2	Maxima	2002	Blue	MN	\$18,000
3	Camry	2003	Blue	OR	\$15,000
4	Prius	2001	White	OR	\$18,000
5	Maxima	2001	White	OR	\$18,000
6	Altima	2002	Green	NY	\$15,000

# Task dependency graph for visualizing the query

ID	Model
2	Maxima
5	Maxima

ID	Year
4	2001
5	2001

ID	Color
4	White
5	White

ID	Color
1	Green
6	Green

Maxima

2001

White

Green

ID	Model	Year
5	Maxima	2001

Maxima AND  
2001

White OR  
Green

Maxima AND 2001 AND (White OR  
Green)

ID	Color
1	Green
4	White
5	White
6	Green

ID	Model	Year	Color
5	Maxima	2001	White

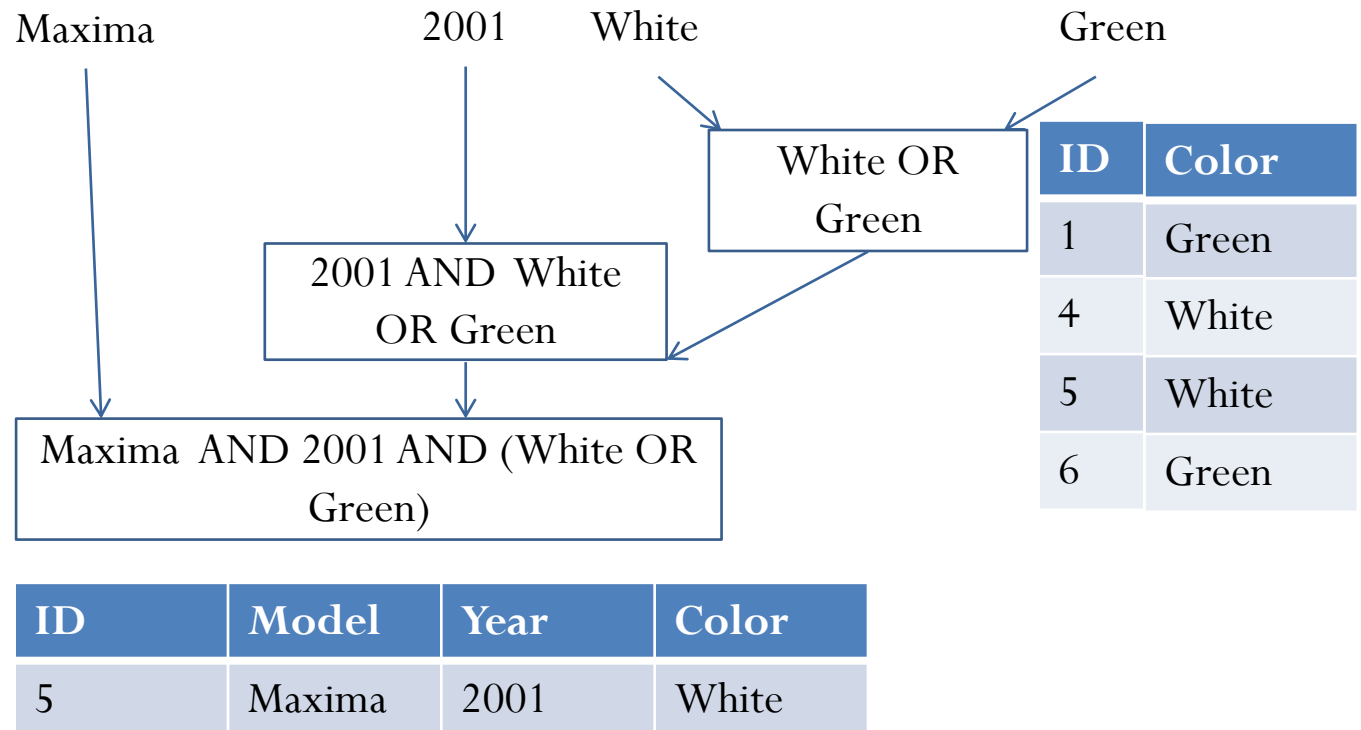
# Task Dependency Graph 2

ID	Model
2	Maxima
5	Maxima

ID	Year
4	2001
5	2001

ID	Color
4	White
5	White

ID	Color
1	Green
6	Green



# Decomposition Terminologies

- **Granularity:** The number and size of tasks based on which a problem is decomposed
- **Fine-Grained:** Decomposition of a problem into huge number of small tasks . Eg: In matrix multiplication of  $n \times n$ , where  $n = 100$ , with Number of tasks = 100.
- **Coarse- Grained:** Decomposition of a problem into small number of huge tasks. Eg: In matrix multiplication of  $n \times n$ , where  $n = 100$ , with Number of tasks = 4.



# Decomposition Terminologies

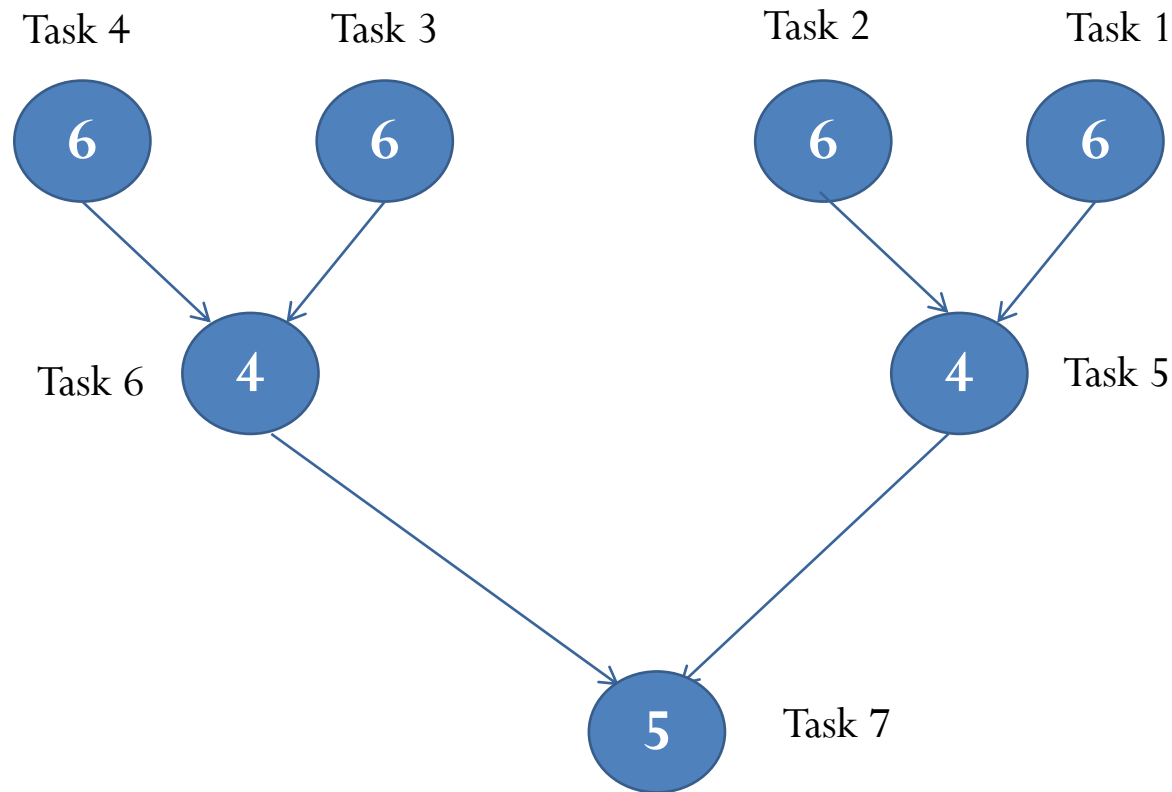
## (Continued...)

- **Degree of Concurrency:** The maximum count of tasks which could be executed in parallel at any given time simultaneously.
- In majority of cases, the maximum concurrency degree is less than the total number of tasks due to dependencies among them
- The maximum concurrency degree for the Task Graphs 1 and 2 is 4 as we are computing the vehicle information based on 4 database entries in parallel.
- In general, the maximum concurrency degree is always equal to the number of leaf nodes in the tree structure where the task dependency graph indicates the tree.

# Decomposition Terminologies (Continued...)

- **Average Degree of Concurrency:** The average count of tasks which can run concurrently throughout the entire program execution duration
- The maximum and average concurrency degrees usually increases if the granularity of tasks are small.
- Eg: In matrix multiplication of  $n \times n$ , where  $n = 100$ , with Number of tasks = 100 has small granularity with high degree of concurrency whereas the same matrix size with Number of tasks = 4 has a larger granularity with smaller concurrency degree

# Abstraction of Task Graph 1



The number inside each node indicates the amount of work to be completed by each node

# Calculation of Average Degree of Concurrency

- Phase 1: The tasks 1, 2, 3 and 4 are executed in parallel

$$\begin{aligned}\text{Degree of Concurrency} &= \text{Sum of the weight values} \\ &= 6+6+6+6 \Rightarrow 24\end{aligned}$$

- Phase 2: The tasks 5 and 6 are executed in parallel

$$\text{Degree of Concurrency} = 4+4 \Rightarrow 8$$

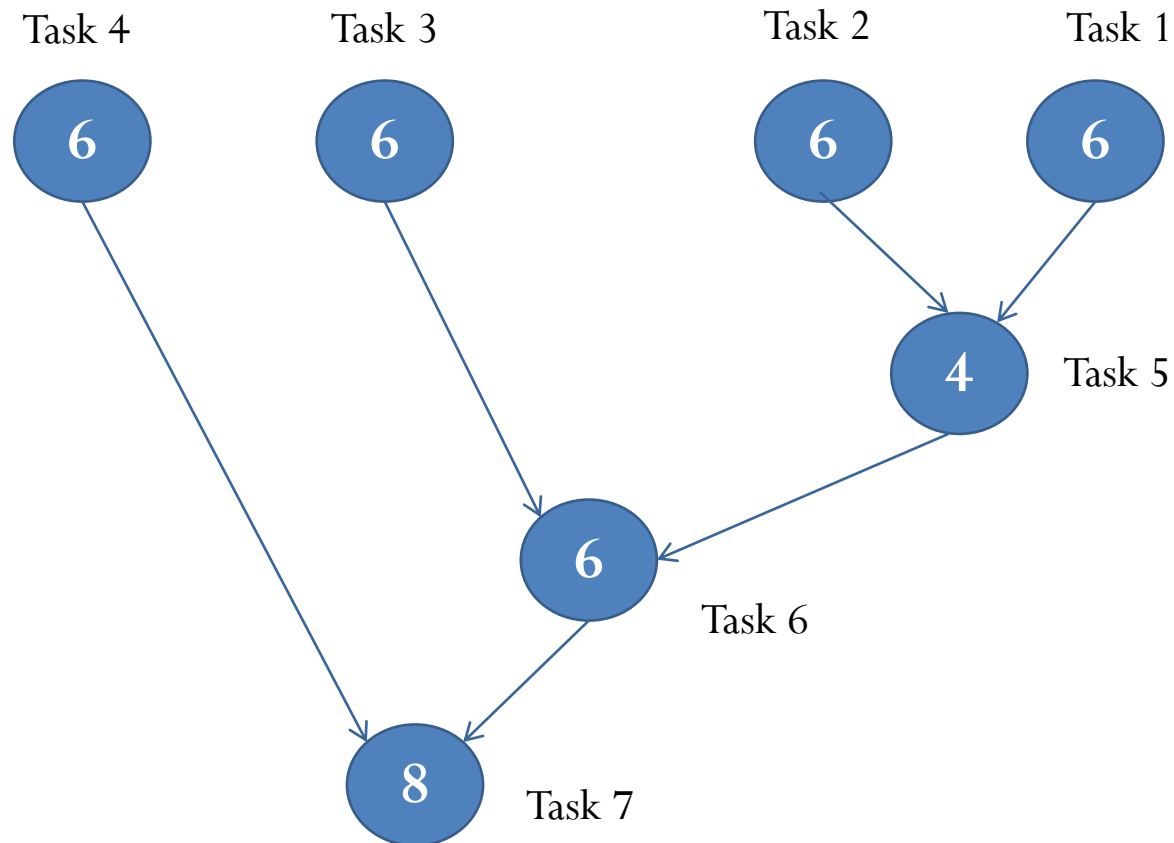
- Phase 3: The task 7 is alone getting executed

$$\text{Degree of Concurrency} = 5$$

- Maximum degree of Concurrency =  $\max(24, 8, 5) \Rightarrow 24$

- Average degree of Concurrency =  
 $(24+8+5)/(6+4+5) \Rightarrow 2.4$

# Abstraction of Task Graph 2



# Calculation of Average Degree of Concurrency

- Phase 1: The tasks 1, 2, 3 and 4 are executed in parallel

$$\begin{aligned}\text{Degree of Concurrency} &= \text{Sum of the weight values} \\ &= 6+6+6+6 \Rightarrow 24\end{aligned}$$

- Phase 2: The tasks alone executed in parallel

$$\text{Degree of Concurrency} = 4$$

- Phase 3: The task 6 is alone getting executed

$$\text{Degree of Concurrency} = 6$$

- Phase 4: The task 7 alone getting executed

- Degree of Concurrency = 8

- Maximum degree of Concurrency =  $\max(24, 4, 6, 8) \Rightarrow 24$

- Average degree of Concurrency =  
 $(24+4+6+8)/(6+4+6+8) \Rightarrow 1.75$

# Critical path calculation

- The feature of task dependency graph which decides the average degree of concurrency for a input granularity is called as its critical path
- The sum of weights of nodes along the path is called as critical path length
- A shorter critical path influences high degree of concurrency
- The critical path length of task dependency graph 1 is: 15 and for task dependency graph 2 : 24
- The factor which limits to ability to yield unbounded speed up is interaction among tasks which runs on various physical processors

# Task- Interaction Graph

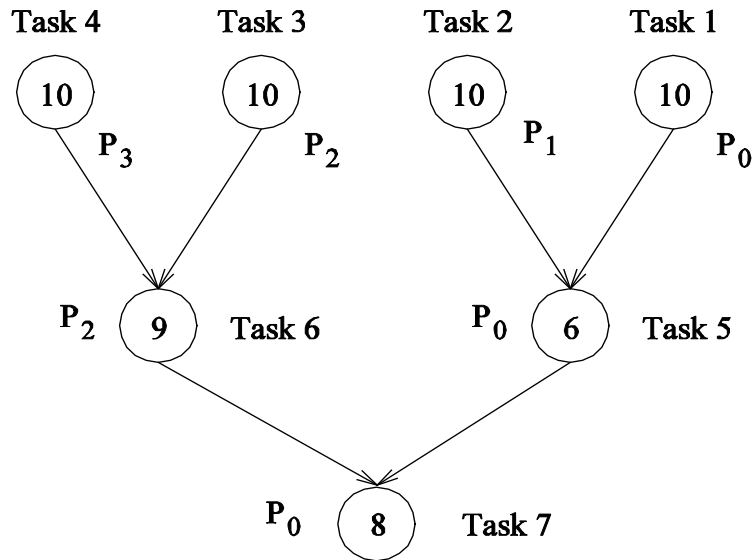
- The interaction pattern among the various tasks could be captured through the task interaction graph.
- The nodes represents the task whereas the edges represents interactions between them
- The edges are undirected but means to indicate the flow of data
- For the vehicles database example, the task interaction graph is the same as that of task dependency graph



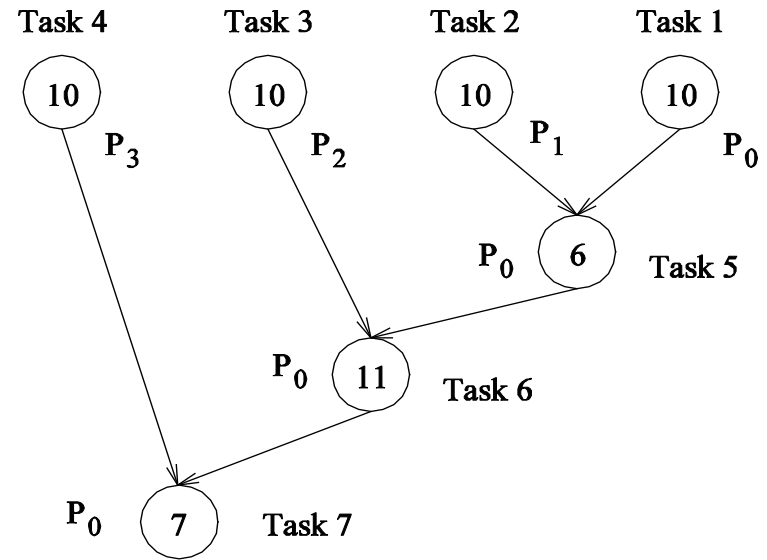
# Mapping processes to processors

- The term process represents the computing agent or processor which carries out the tasks
- The tasks are assigned to various processes to carry out the execution procedure is called as mapping
- The task dependency and task interaction graphs formed from a decomposition process plays a major role in the appropriate selection of a good mapping algorithm
- A good mapping should result in maximum utilization of concurrency by mapping independent tasks into different processes which seeks to minimize the total completion time
- It should ensure that the processes should be available to execute the task on the critical path and minimize task interactions

# Processes and Mapping: Example



(a)



(b)

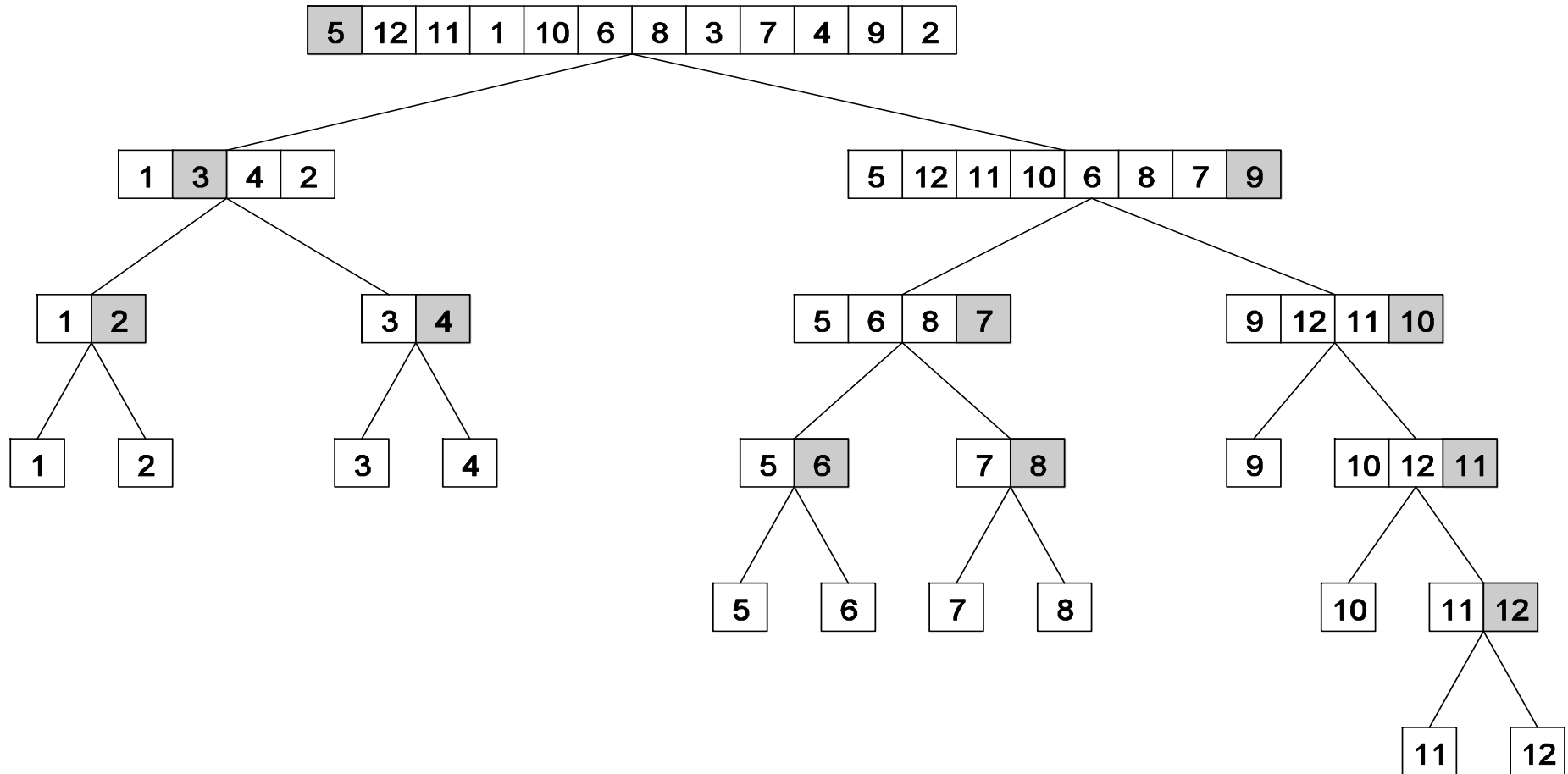
# Decomposition Techniques

- Recursive decomposition
- Data decomposition
- Exploratory decomposition
- Speculative decomposition

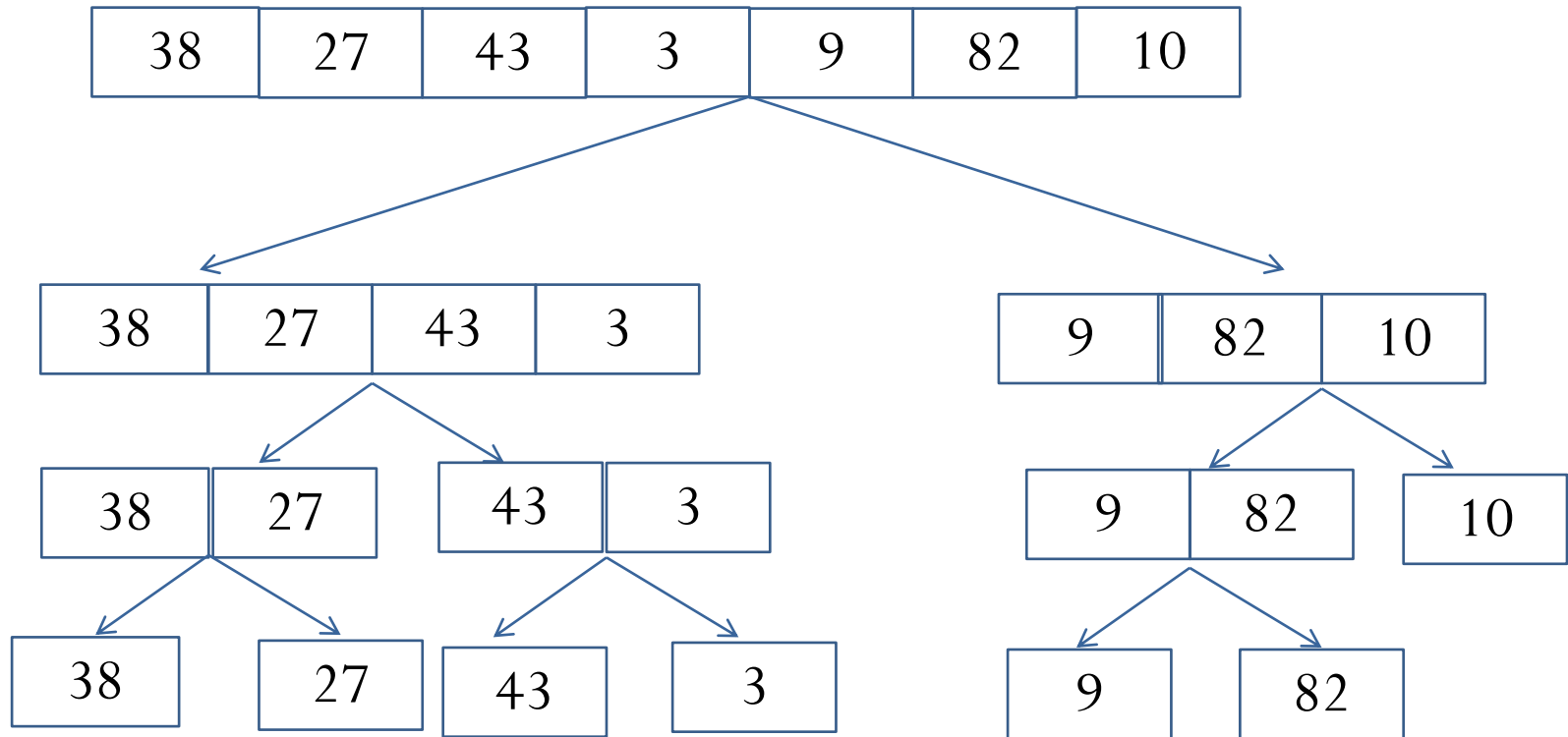
# Recursive Decomposition

- It is a method of problem solving using divide and conquer strategy for achieving concurrency
- The main problem is divided into a group of sub problems
- Each sub problem is then split by applying the same division strategy.
- This strategy achieves concurrency by nature
- Example: Quick sort, Merge sort

# Quick Sort Example: Pivot element



# Merge sort example



# Merge sort algorithm

- The merge sort function repeatedly divides the entire array into two parts until the sub array size of 1 is reached
- Once the size  $p==r$  is reached, the merge function plays the role to combine the sorted arrays.

MergeSort(A, p, r)

    If ( $p > r$ )

        return

$q = (p+r)/2$

    mergeSort(A, p, q)

    mergeSort(A, q+1, r)

    merge(A, p, q, r)

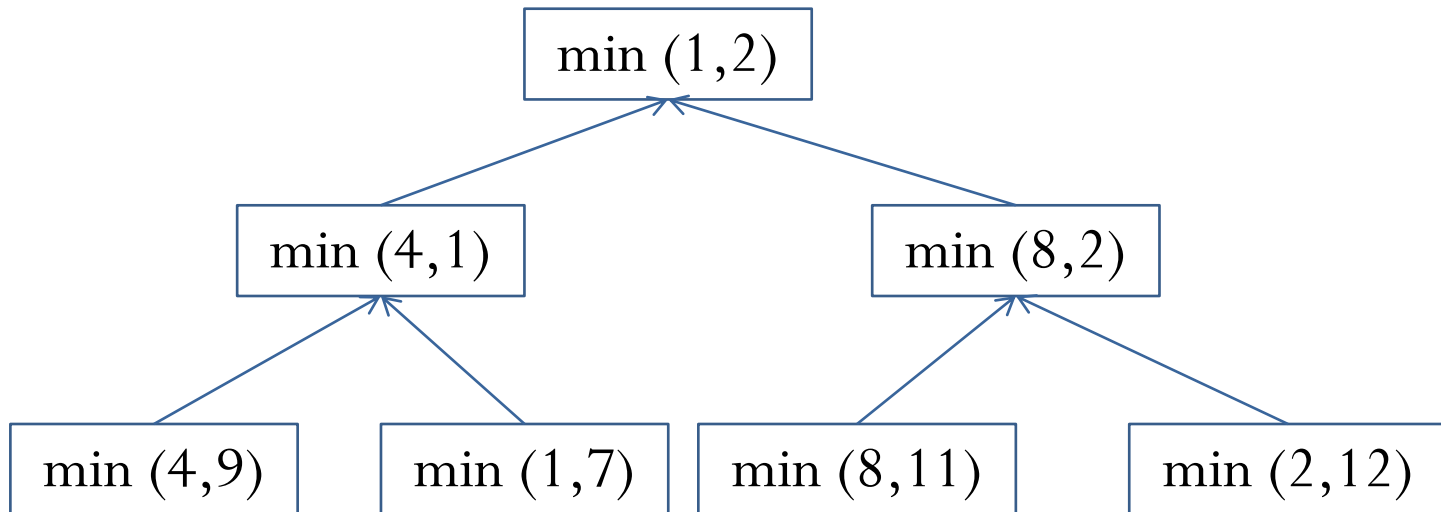
# Finding Minimum of an array

- 1. **procedure** SERIAL\_MIN ( $A, n$ )
- 2. **begin**
- 3.  $min = A[0];$
- 4. **for**  $i := 1$  **to**  $n - 1$  **do**
- 5.       **if** ( $A[i] < min$ )  $min := A[i];$
- 6. **endfor**;
- 7. **return**  $min;$
- 8. **end** SERIAL\_MIN



# Task-dependency graph for finding minimum numbers

- 4,9,1,7,8,11,2,12



# Recursive algorithm

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

# Data Decomposition

- It is a commonly used decomposition technique to operate on the large data structure.
- There are two steps in this method. The first step is related to partitioning of data related to computations and the second step is related to induce partitioning of computations into various tasks.
- Partition: Input, Output, Input+Output, Intermediate

# Partitioning of output data

- In many cases, each output element can be computed independently of each other as a input function
- Partitioning of output data decomposes the problem into separate tasks where each task is assigned to the work of computing the output portion
- Eg: Matrix multiplication
- Multiplying 2  $n \times n$  matrices A, B yields the result C
- Matrix is decomposed into 4 tasks based on the partitioning

# Partitioning of output data (Continued...)

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

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

A partitioning of output data does not result in a unique decomposition into tasks

Decomposition I	Decomposition II
Task 1: $\mathbf{C}_{1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1}$	Task 1: $\mathbf{C}_{1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1}$
Task 2: $\mathbf{C}_{1,1} = \mathbf{C}_{1,1} + \mathbf{A}_{1,2} \mathbf{B}_{2,1}$	Task 2: $\mathbf{C}_{1,1} = \mathbf{C}_{1,1} + \mathbf{A}_{1,2} \mathbf{B}_{2,1}$
Task 3: $\mathbf{C}_{1,2} = \mathbf{A}_{1,1} \mathbf{B}_{1,2}$	Task 3: $\mathbf{C}_{1,2} = \mathbf{A}_{1,2} \mathbf{B}_{2,2}$
Task 4: $\mathbf{C}_{1,2} = \mathbf{C}_{1,2} + \mathbf{A}_{1,2} \mathbf{B}_{2,2}$	Task 4: $\mathbf{C}_{1,2} = \mathbf{C}_{1,2} + \mathbf{A}_{1,1} \mathbf{B}_{1,2}$
Task 5: $\mathbf{C}_{2,1} = \mathbf{A}_{2,1} \mathbf{B}_{1,1}$	Task 5: $\mathbf{C}_{2,1} = \mathbf{A}_{2,2} \mathbf{B}_{2,1}$
Task 6: $\mathbf{C}_{2,1} = \mathbf{C}_{2,1} + \mathbf{A}_{2,2} \mathbf{B}_{2,1}$	Task 6: $\mathbf{C}_{2,1} = \mathbf{C}_{2,1} + \mathbf{A}_{2,1} \mathbf{B}_{1,1}$
Task 7: $\mathbf{C}_{2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2}$	Task 7: $\mathbf{C}_{2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2}$
Task 8: $\mathbf{C}_{2,2} = \mathbf{C}_{2,2} + \mathbf{A}_{2,2} \mathbf{B}_{2,2}$	Task 8: $\mathbf{C}_{2,2} = \mathbf{C}_{2,2} + \mathbf{A}_{2,2} \mathbf{B}_{2,2}$

# Partitioning

- Partitioning of output data could be done if each output is computed as a function of input
- In most cases, it is not possible to partition the output data
- For finding the sum of sequence of  $N$  numbers using  $n$  processors, the input could be partitioned into  $p$  subsets each of equal sizes
- Each task then computes the sum of each numbers in their own subset
- The final partial results are added to yield the final result

# Output Data Decomposition-Example

- Consider the problem of counting the instances of given itemsets in a database of transactions. In this case, the output (itemset frequencies) can be partitioned across tasks.



**(a) Transactions (input), itemsets (input), and frequencies (output)**

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	1
	B, D, E, F, K, L		D, E		3
	A, B, F, H, L		C, F, G		0
	D, E, F, H		A, E		2
	F, G, H, K,		C, D		1
	A, E, F, K, L		D, K		2
	B, C, D, G, H, L		B, C, F		0
	G, H, L		C, D, K		0
	D, E, F, K, L				
	F, G, H, L				

**(b) Partitioning the frequencies (and itemsets) among the tasks**

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	1
	B, D, E, F, K, L		D, E		3
	A, B, F, H, L		C, F, G		0
	D, E, F, H		A, E		2
	F, G, H, K,				
	A, E, F, K, L				
	B, C, D, G, H, L				
	G, H, L				
	D, E, F, K, L				
	F, G, H, L				

**task 1**

Database Transactions	A, B, C, E, G, H	Itemsets	C, D	Itemset Frequency	1
	B, D, E, F, K, L		D, K		2
	A, B, F, H, L		B, C, F		0
	D, E, F, H		C, D, K		0
	F, G, H, K,				
	A, E, F, K, L				
	B, C, D, G, H, L				
	G, H, L				
	D, E, F, K, L				
	F, G, H, L				

**task 2**

From the previous example, the following observations can be made:

- If the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.
- If the database is partitioned across processes as well (for reasons of memory utilization), each task first computes partial counts. These counts are then aggregated at the appropriate task.

# Input Data Decomposition Example

- In the database counting example, the input (i.e., the transaction set) can be partitioned. This induces a task decomposition in which each task generates partial counts for all itemsets. These are combined subsequently for aggregate counts.

Partitioning the transactions among the tasks

Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	1
	B, D, E, F, K, L		D, E		2
	A, B, F, H, L		C, F, G		0
	D, E, F, H		A, E		1
	F, G, H, K,		C, D		0
			D, K		1
			B, C, F		0
			C, D, K		0

task 1

Database Transactions		Itemsets	A, B, C	Itemset Frequency	0
			D, E		1
			C, F, G		0
	A, E, F, K, L		A, E		1
	B, C, D, G, H, L		C, D		1
	G, H, L		D, K		1
	D, E, F, K, L		B, C, F		0
	F, G, H, L		C, D, K		0

task 2

# Partitioning Input *and* Output Data

- Often input and output data decomposition can be combined for a higher degree of concurrency. For the itemset counting example, the transaction set (input) and itemset counts (output) can both be decomposed as follows:

Partitioning both transactions and frequencies among the tasks

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	2
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	1
F, G, H, K,		

task 1

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H		
B, D, E, F, K, L		
A, B, F, H, L		
D, E, F, H	C, D	0
F, G, H, K,	D, K	1
	B, C, F	0
	C, D, K	0

task 2

Database Transactions	Itemsets	Itemset Frequency
	A, B, C	0
	D, E	1
	C, F, G	0
	A, E	1
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

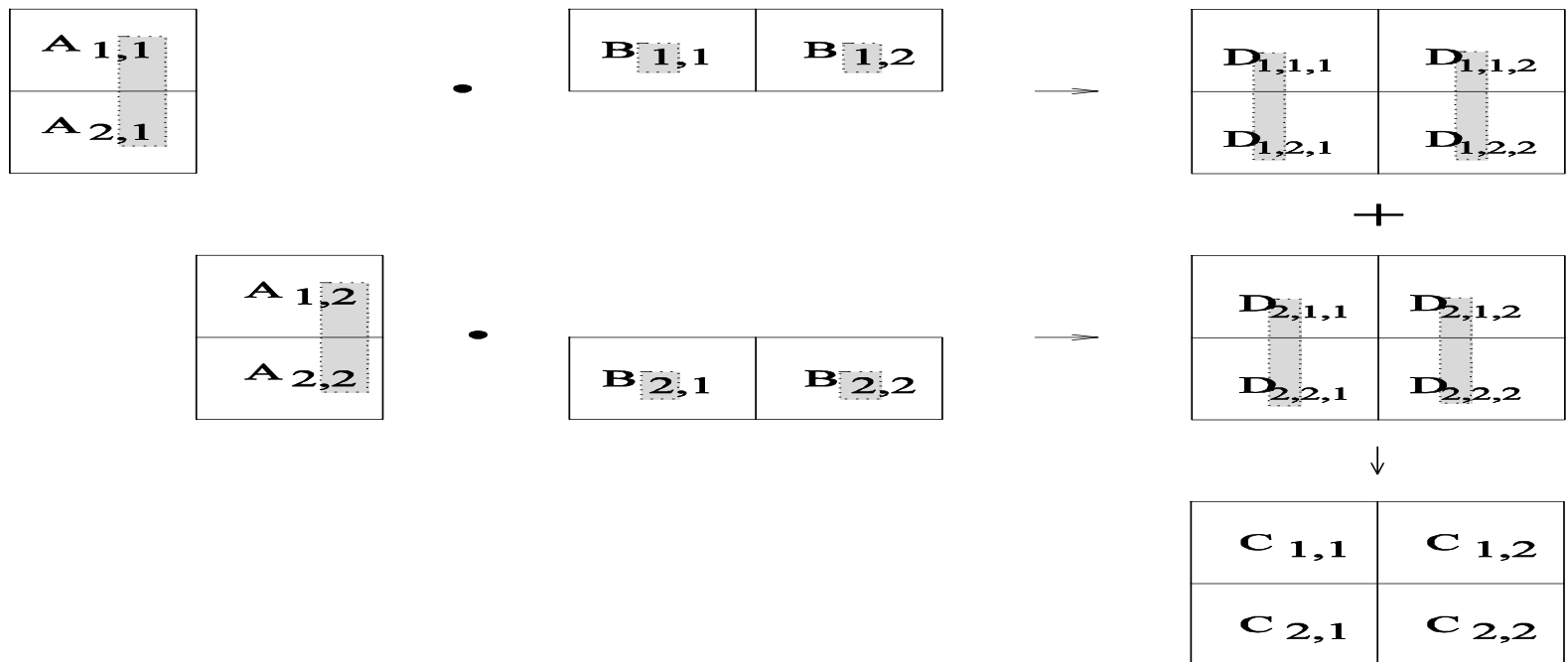
task 3

Database Transactions	Itemsets	Itemset Frequency
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L	C, D	1
D, E, F, K, L	D, K	1
F, G, H, L	B, C, F	0
	C, D, K	0

task 4

# Intermediate Data Partitioning

- Computation can often be viewed as a sequence of transformation from the input to the output data.
- In these cases, it is often beneficial to use one of the intermediate stages as a basis for decomposition.



# Intermediate Data Partitioning: Example

A decomposition of intermediate data structure leads to the following decomposition into 8 + 4 tasks:

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

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

Task 01:  $\mathbf{D}_{1,1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1}$

Task 03:  $\mathbf{D}_{1,1,2} = \mathbf{A}_{1,1} \mathbf{B}_{1,2}$

Task 05:  $\mathbf{D}_{1,2,1} = \mathbf{A}_{2,1} \mathbf{B}_{1,1}$

Task 07:  $\mathbf{D}_{1,2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2}$

Task 09:  $\mathbf{C}_{1,1} = \mathbf{D}_{1,1,1} + \mathbf{D}_{2,1,1}$

Task 11:  $\mathbf{C}_{2,1} = \mathbf{D}_{1,2,1} + \mathbf{D}_{2,2,1}$

Task 02:  $\mathbf{D}_{2,1,1} = \mathbf{A}_{1,2} \mathbf{B}_{2,1}$

Task 04:  $\mathbf{D}_{2,1,2} = \mathbf{A}_{1,2} \mathbf{B}_{2,2}$

Task 06:  $\mathbf{D}_{2,2,1} = \mathbf{A}_{2,2} \mathbf{B}_{2,1}$

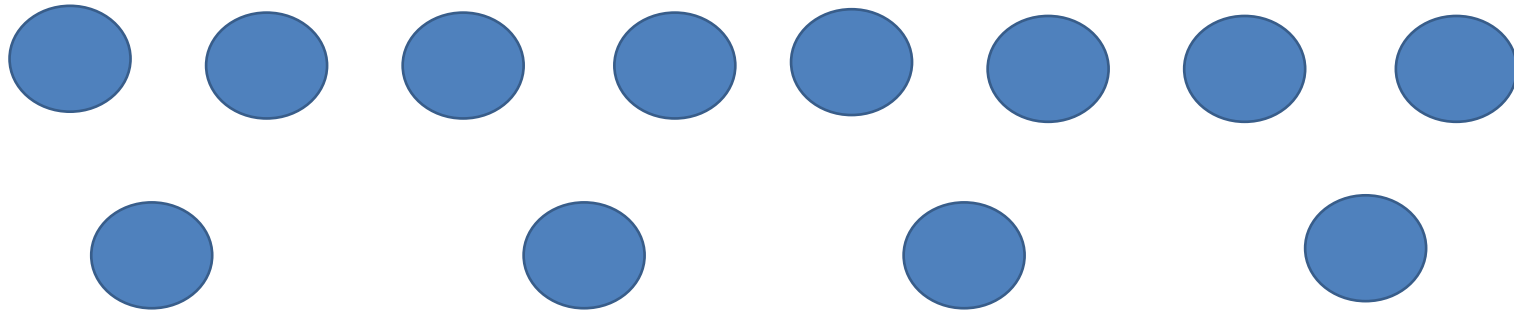
Task 08:  $\mathbf{D}_{2,2,2} = \mathbf{A}_{2,2} \mathbf{B}_{2,2}$

Task 10:  $\mathbf{C}_{1,2} = \mathbf{D}_{1,1,2} + \mathbf{D}_{2,1,2}$

Task 12:  $\mathbf{C}_{2,2} = \mathbf{D}_{1,2,2} + \mathbf{D}_{2,2,2}$



# Task dependency graph ??



# The Owner Computes Rule


- The ***Owner Computes*** Rule generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of **input data decomposition**, the owner computes rule implies that **all computations that use the input data are performed by the process**.
- In the case of **output data decomposition**, the owner computes rule implies that **the output is computed by the process to which the output data is assigned**

# Exploratory decomposition


- This method is used for the problems whose underlying calculations correspond to a search of a space for yielding solutions.
- Eg: 15-Puzzle problem
- The 15 puzzle problem contains 15 tiles each numbered from 1 to 15 and one blank tile placed in 4X4 grid.
- Based on the grid configuration, the possible moves could be up, down, left and right
- This problem could be solved using tree search techniques

# 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).

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

(a)

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

(b)

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

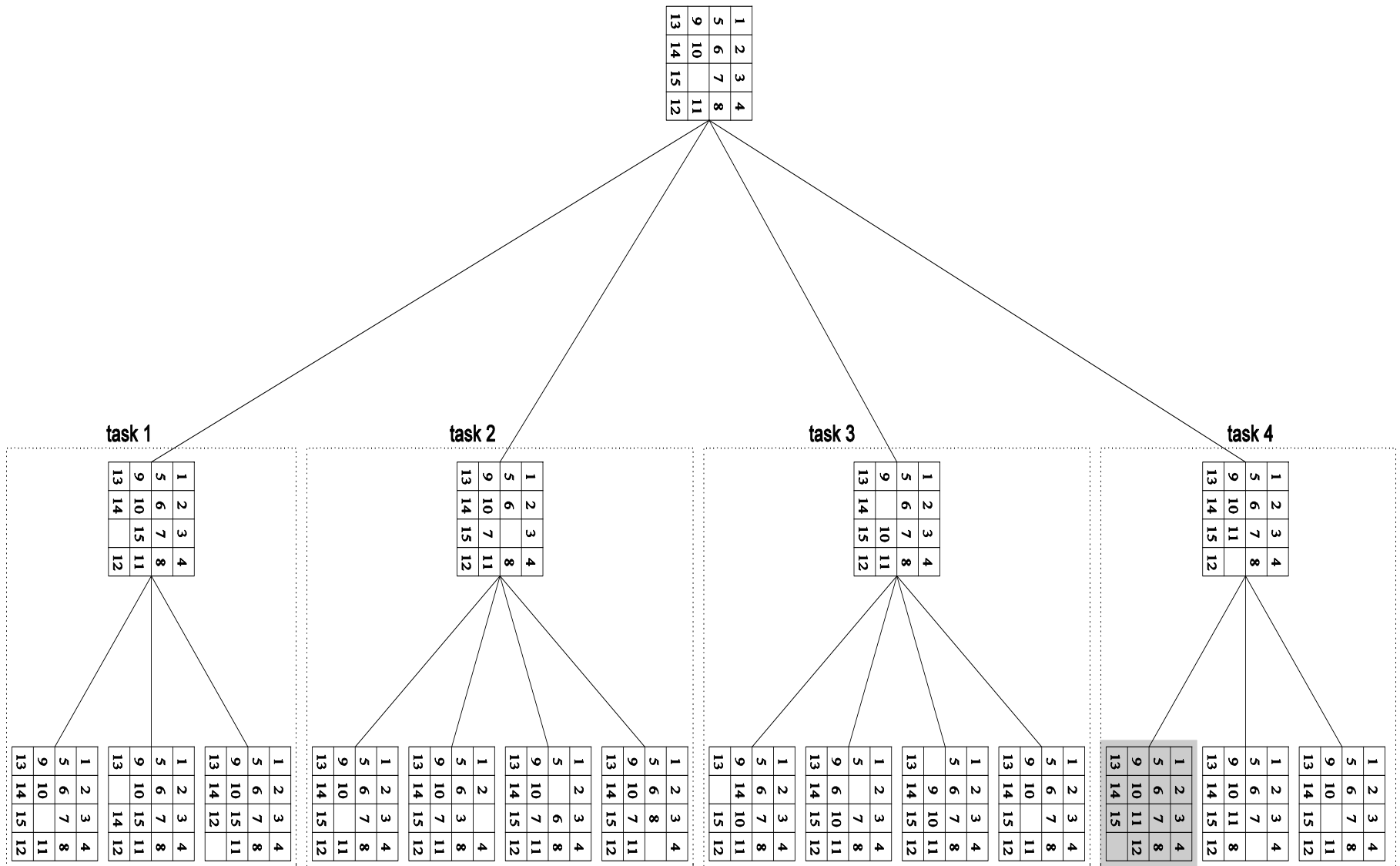
(c)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(d)

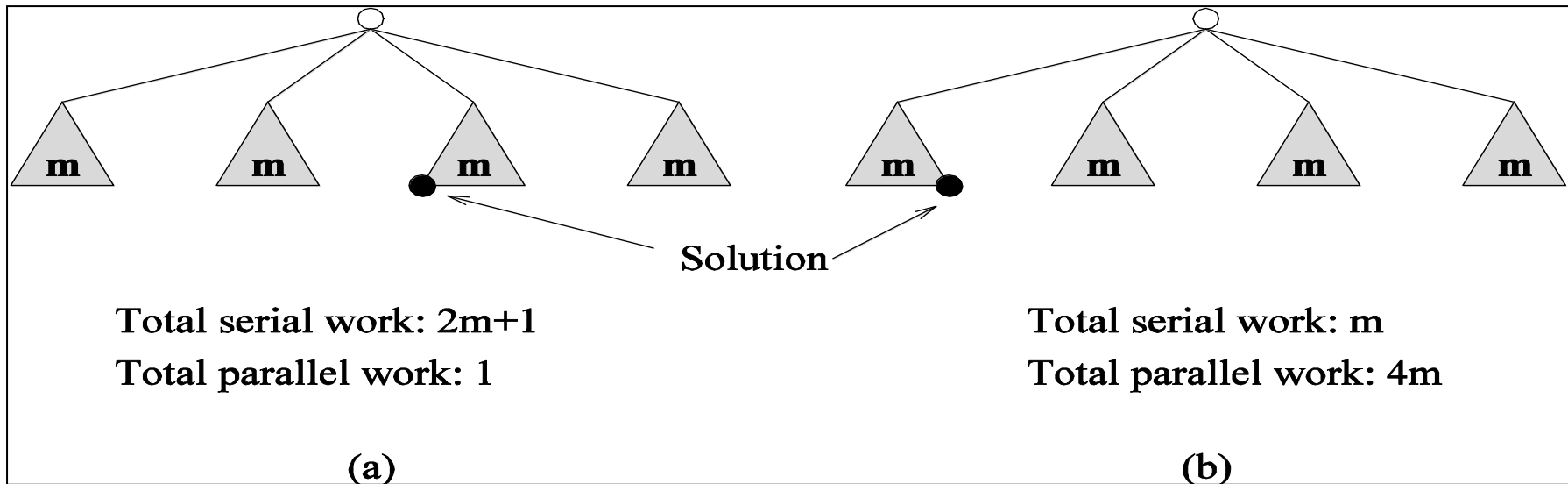
Of-course, the problem of computing the solution, in general, is much more difficult than in this simple example.

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



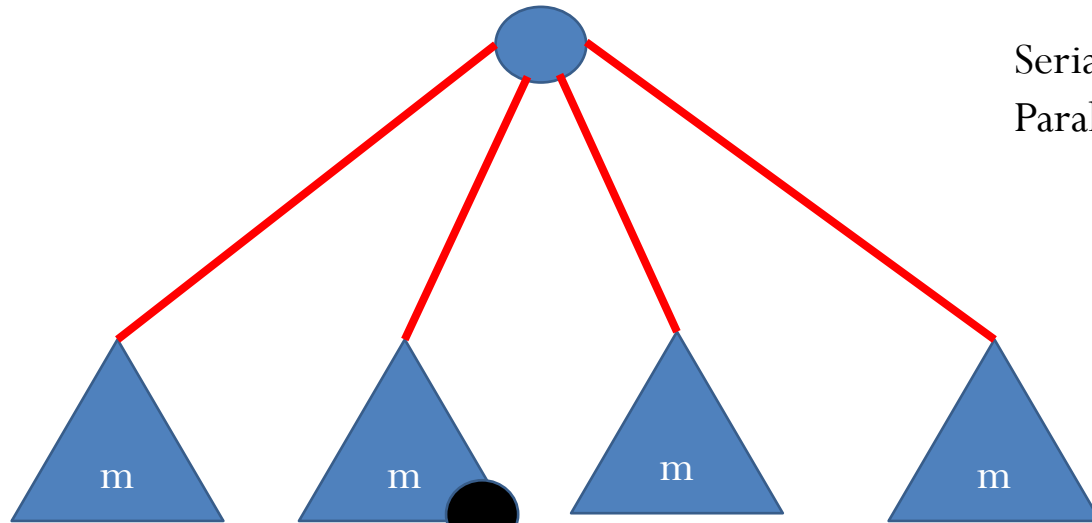
# Continued...

- In exploratory decomposition, unfinished tasks could be terminated when the overall solution is achieved
- The work done by the parallel process could be either smaller or greater than performed in a serial fashion
- The speed up of parallel algorithm is based on in which partitioned task results in achieving a final solution



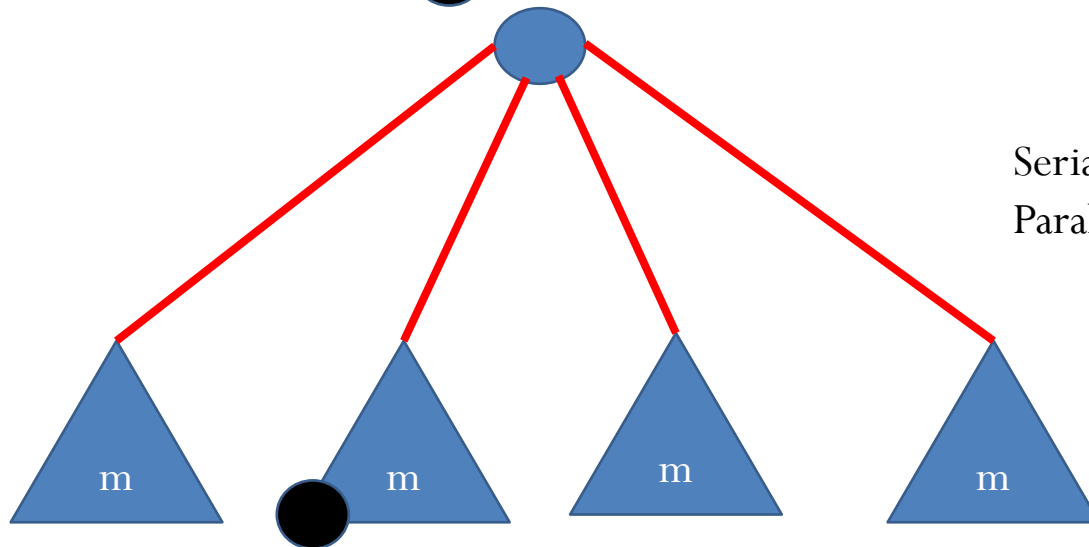
- For example, consider a search space that has been partitioned into **four** concurrent tasks.
- If the **solution lies right at the beginning** of the search space corresponding to task 3 then **it will be found** almost immediately by the **parallel** formulation.
- The **serial** algorithm would have found the solution **only after performing work** equivalent to searching the entire space corresponding to tasks 1 and 2.
- On the other hand, **if the solution lies towards the end of the search space corresponding to task 1**, then the **parallel** formulation will perform **almost four times** the work of the **serial** algorithm and will yield **no speedup**.

# Example-1



Serial Work:????

Parallel Work:????

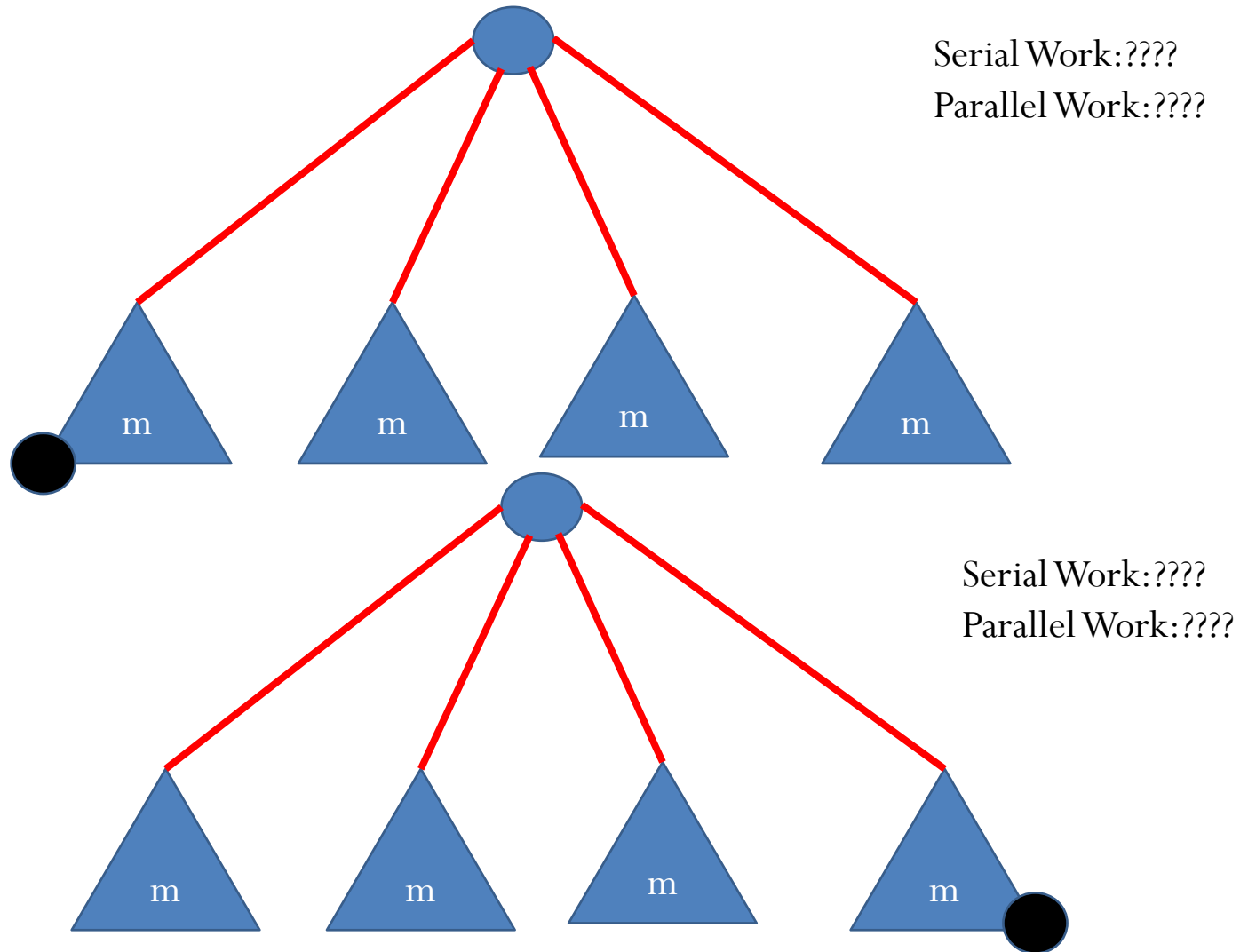


Serial Work:????

Parallel Work:????



## Example-2



# Speculative decomposition

- In some applications, dependencies between tasks are not known a-priori.
- For such applications, it is impossible to identify independent tasks.
- There are generally two approaches to dealing with such applications:
  - **conservative approaches**: which identify independent tasks only when they are *guaranteed to not have dependencies*, and,
  - **optimistic approaches**: which schedule tasks even when they may *potentially be erroneous*.
- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.

# Speculative decomposition

- It is a method for representing a program taking many possible computationally significant branches based on the output of its preceding ones.
- If the output of Task 1 is required for the computation of Task 2, rest of the tasks can concurrently begin the computations.
- The run time of parallel computation is smaller than the serial run time based on the time required for evaluating the computations as the next stage task may depend on that.
- Parallel computation with concept of switching leads to unnecessary intermediate computations
- Hence, different formulation of speculative method is used

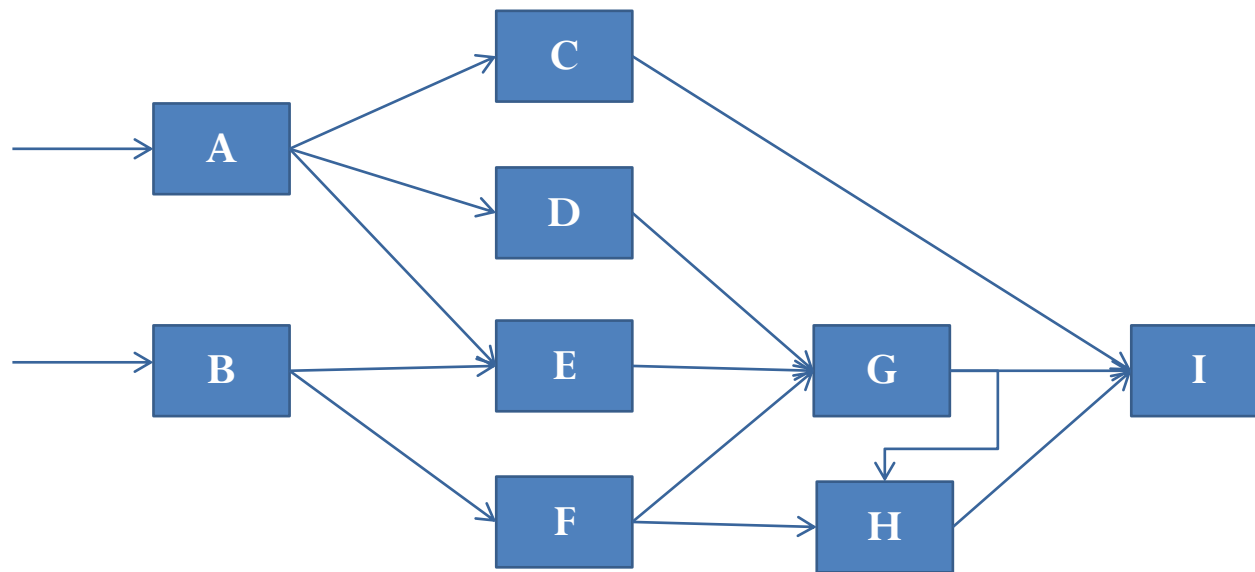
- Events are extracted precisely in time order, processed, and if required, resulting events are inserted back into the event list.
- Consider your day today as a **discrete event system** - you get up, get ready, drive to work, work, eat lunch, work some more, drive back, eat dinner, and sleep.
- Each of these events may be processed independently, however, in driving to work, you might meet with an unfortunate accident and not get to work at all.
- Therefore, an **optimistic scheduling** of other events will have to **be rolled back**

# Continued...

- Speculative decomposition is useful in the case of discrete event simulations
- Consider a simulation system represented as a network
- The network nodes indicate components each has buffer of jobs with initial one in idle state
- From the input queue, jobs will be picked up for execution within a finite time and puts it in the input buffer of connected components by outgoing edges
- A component need to wait till the input buffer of any of its neighbour is full, and the number of job types is finite

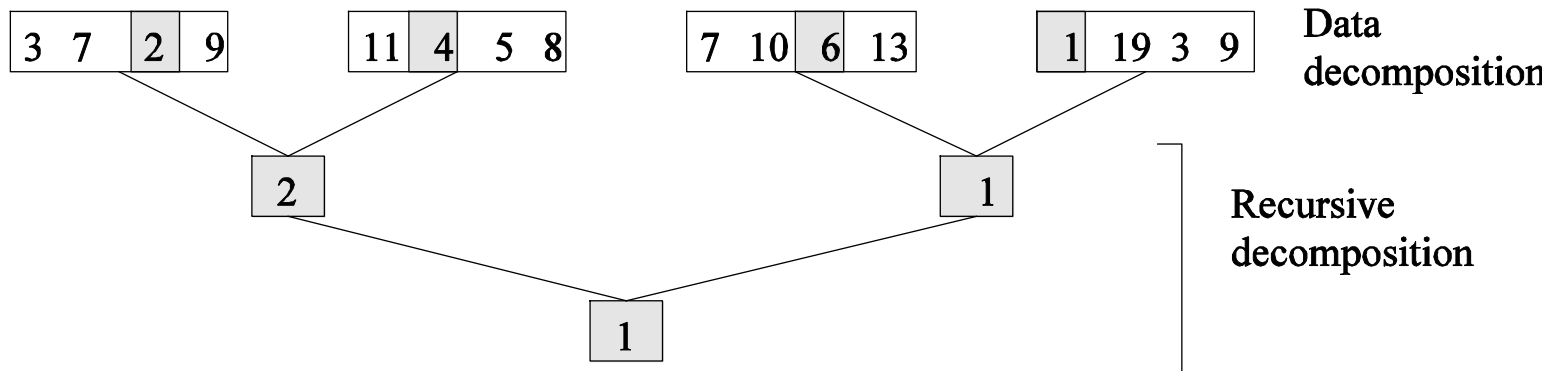
# Continued...

- The role of input job is to process the output component in a finite time
- The network functioning is to be done based on the given sequences of input jobs and compute the total completion time



# Hybrid decompositions

- These methods are not exclusive but could be combined together
- If a computation is structured in multiple stages, it would be necessary to apply various decomposition methods in the various stages
- Even for simple problems like finding a minimum of a list of numbers, a mix of data and recursive decomposition works well.



# Characteristics of Tasks and Interactions

- The tasks which constitutes parallel algorithms could be generated either statically or dynamically.
- **Static task** generation represents all tasks are **known before** execution. Example: Matrix multiplication, Finding Minimum
- **Dynamic task** generation indicates the actual tasks and dependencies are **not known priori**. Example: Quick sort
- Exploratory decomposition could formulate tasks either statically or dynamically, Example: 15-puzzle problem
- **Task size** is the relative time required for completing a task



# Continued...

- The tasks are said to be **uniform** if all requires roughly the same time to complete Example: **Matrix multiplication**
- The tasks are said to be **non-uniform** if the task time varies significantly. Example: **Quick sort**
- **Knowledge of task** sizes indicates the choice of mapping scheme based on knowledge of task size. If the tasks size is known, it could be **utilized for mapping** tasks to processes.
- **Size of data associated with tasks** is an important criteria for **mapping the associated data** with task. The size and location of the data decides data-movement overheads

# Characteristics of Inter-task interactions

- Static Vs Dynamic:
  - Classification based on interaction pattern
  - Programming could be done using message passing paradigm
  - Dynamic interactions are harder to program
- Regular Vs Irregular
  - Classification based on spatial structure
  - A pattern is **regular** if it needs some **structure** to be exploited for efficient implementation
  - A pattern is **irregular** if **no regular** pattern exists

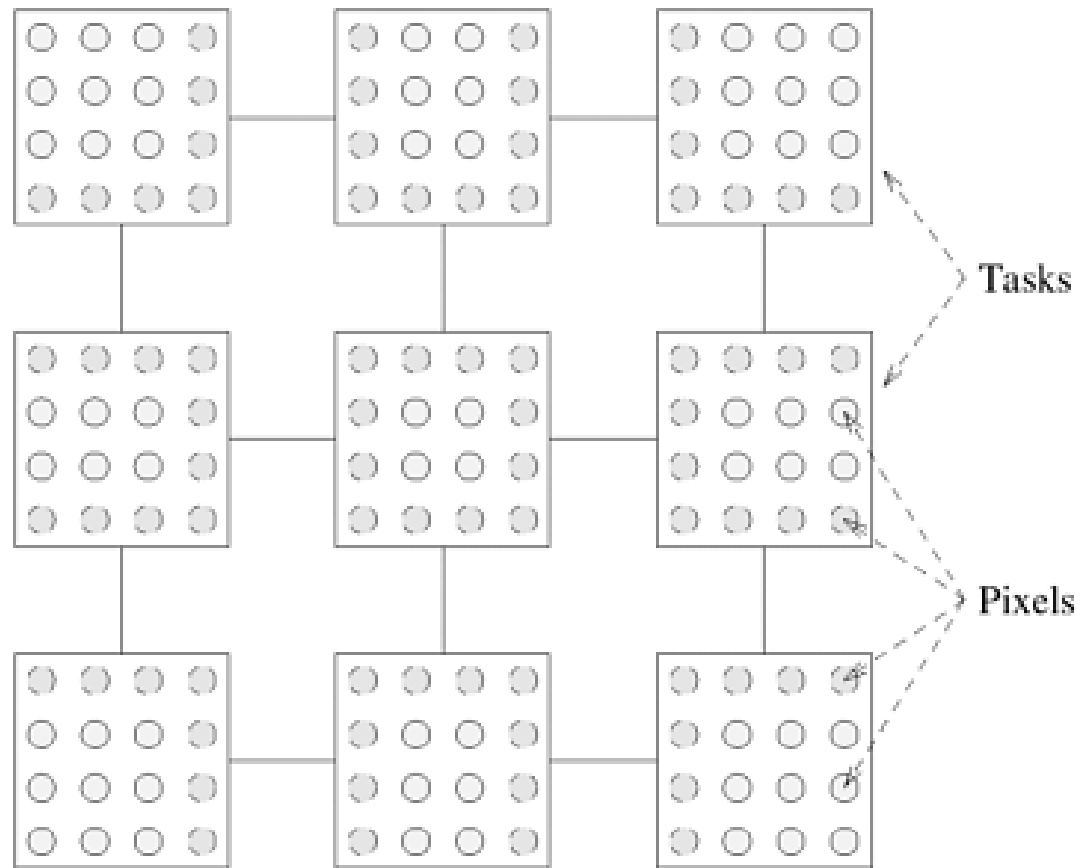
# Continued...

- Read-only Vs Read-write:
  - In read-only interactions, the tasks require read –access for the data to be shared among various concurrent tasks
  - Example: In matrix multiplication, the tasks need to read the shared input matrices A and B.
  - In read-write interactions, multiple tasks will be waiting to read and write on the same shared input data.
- One Way Vs Two way:
  - The two way interactions involves the producer and consumer tasks and the communication is two way.
  - If only a pair of communicating tasks both initiates and completes the interaction **without disturbing** other one, then it is called as one way.

# Example: Image Dithering

- In image dithering, the color of each pixel in the image is determined as the weighted average of its original value and the values of its neighboring pixels.
- We can easily decompose this computation, by breaking the image into square regions and using a different task to dither each one of these regions.
- Note that each task needs to access the pixel values of the region assigned to it as well as the values of the image surrounding its region.
- Thus, if we regard the tasks as nodes of a graph with an edge linking a pair of interacting tasks, the resulting pattern is a two-dimensional mesh

# Example: Image Dithering



# Parallel algorithm models

- The data parallel model
- Task graph model
- The work pool model
- Pipelining or producer-consumer model

# Data-parallel model

- The tasks are mapped either statically or semi-statically into processes and each task performs similar operations on different data
- The result of identical operations applied on concurrently various data items is represented as data parallelism
- The data operates on various phases will be different
- It could be implemented in either in shared model or message passing model
- Interaction overheads in this model could be reduced by choosing a **locality preserving decomposition** through overlapping of computation and interaction

# Task graph model

- The inter dependencies and relationships between various tasks are utilized for promoting locality or for reducing the interaction costs
- This model would be appropriate if the **amount of data associated with tasks is large** when compared with the amount of related computation.
- In general, the task mapping is done statically for optimizing cost of data movement
- In few scenario's, decentralized dynamic mapping is adopted
- This type of parallelism expressed by independent tasks in a task dependency graph is represented as task level parallelism



# The work pool model

- This model is characterized by the dynamic mapping of tasks into processes to carry out load balancing
- There is no desired pre-mapping
- The mapping could follow either centralized or decentralized fashion
- The task pointers are stored in list, priority queue, hash table, tree or any physically distributed data structure.
- The processes may generate work and add it to the global work pool
- Eg: Parallelization of loops through chunk scheduling

# Master-Slave Model

- One or more process generate work and allocate it to worker processes.
- If the size of tasks could be estimated priori, tasks could be allocated or a random mapping could be done for load balancing.
- At different times, the workers will be assigned with smaller pieces of work.
- Synchronization between workers is done through the manager
- This could be viewed as hierarchical or multi-level manager worker model in which the top level is responsible for providing input chunks to the second level where it is further sub divided.

# Pipelining or producer-consumer model

- A stream of data is passed through various processors with tasks assigned to each of it.
- The concurrent execution of different programs on a data stream is called stream parallelism
- Load balancing is a function of task granularity
- If the granularity is large, the time taken is more to fill the pipeline
- If the granularity is small, the interaction overheads increases as the tasks need to interact even for small pieces of computation.