Module-3

Parallel Algorithm Design

Overview:

- *Tasks and Decomposition (concurrent work)*
- *Decomposition Techniques*
- *Processes and Mapping (work to parallel process mapping)*
- *Processes Versus Processors*
- *Mapping Techniques for Load Balancing*
- *Distribution : input, output, intermediate data*
- *Managing access to data*
- *Synchronizing the processors*
- *Characteristics of Tasks and Interactions*
- *Methods for Minimizing Interaction Overheads*
- *Parallel Algorithm Design Models*

Correct + Fast+ efficient → Performance.

Theory (complexity) vs. Practical (actual run-time)

Decomposition, Tasks, Dependency Graphs

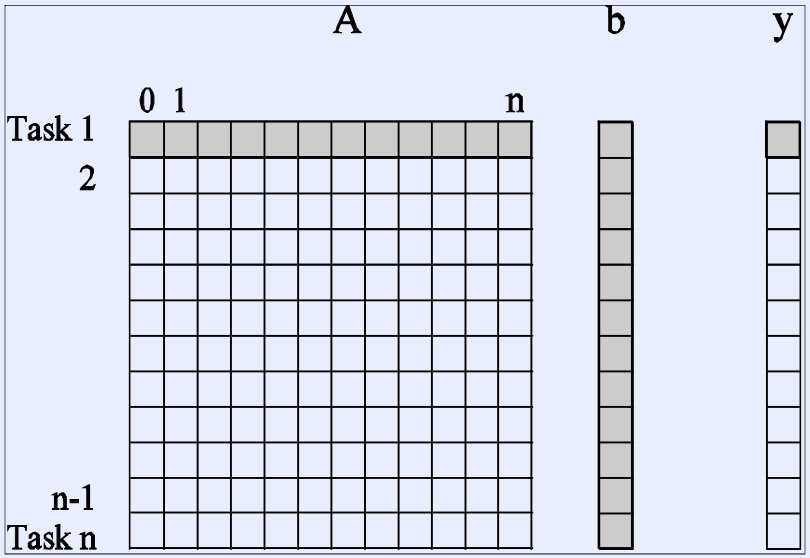
- Dividing the problem into tasks that can be executed concurrently/ in parallel → decomposition.
- Task may be small or large. (programmer defined)
- Tasks may be of same, different, or even intermediate sizes.

- A decomposition can be represented 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***.

- Reduction example. Matrix-vector multiplication.

Task dependency graph or DAG (directed acyclic graph).

Example: Multiplying a Dense Matrix with a Vector



Computing elements (n) of output vector ***y*** is independent of other elements. So, a matrix-vector product can be decomposed into ***n*** tasks.

Grey → portion of the matrix & vector accessed by Task 1.

observation

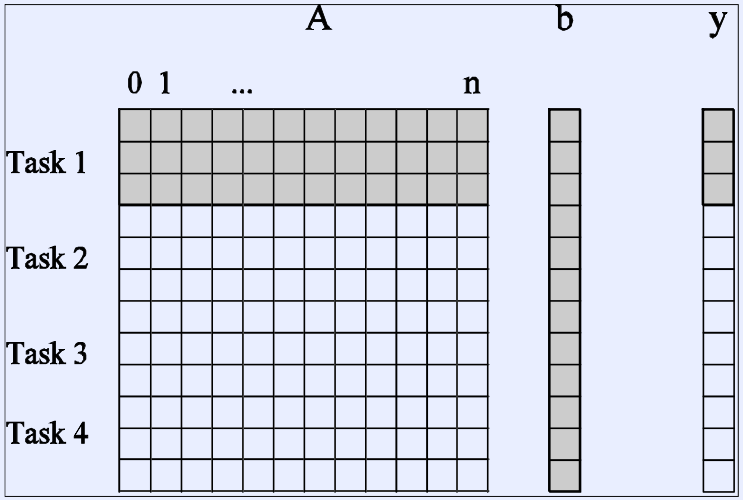
- While tasks share data (namely, the vector ***b***), they do not have any control dependencies - i.e., no task needs to wait for the (partial) completion of any other.

- 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?*

Different ways of arranging computation → 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 dense matrix-vector product example.
Each task → 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. (reduction example)
- The ***average degree of concurrency*** is the average number of tasks that can be processed in parallel over the execution of the program.
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.
- Degree of concurrency depends on shape of task dependency graph (can be different for same granurality).

Cost model

Each node → task (more !); edge → dependency (less !)

Complexity analysis - Cost model (PRAM – shared memory) :

- Each operation takes one unit of time
- Same speed of all processor
- No edge cost
- **No synchronization, communication overhead**

Summary: ignores practical issues of memory access time, neglects parallel overhead, however **problem size** and no of **processors** are there.

Serial cost/Time complexity (order of time)
Parallel cost/time complexity (order of time x no. Of processors)

Work span law

Work (w)=

1. Total no of nodes/taks/vertices if all node does same and unit amount of work. Or
2. Summation of work done by each node/task if all node does not do the same amount of work

Span/depth (D):
Longest path in the graph

W/D=average amount of parallelism (optimal number of processor to keep all processors busy on average)

Work span law

- Depth/span is lower bound (min possible time)
- When all processors(p) does same work → W/p is also lower bound
- Combine both to get work-span law for lower bound

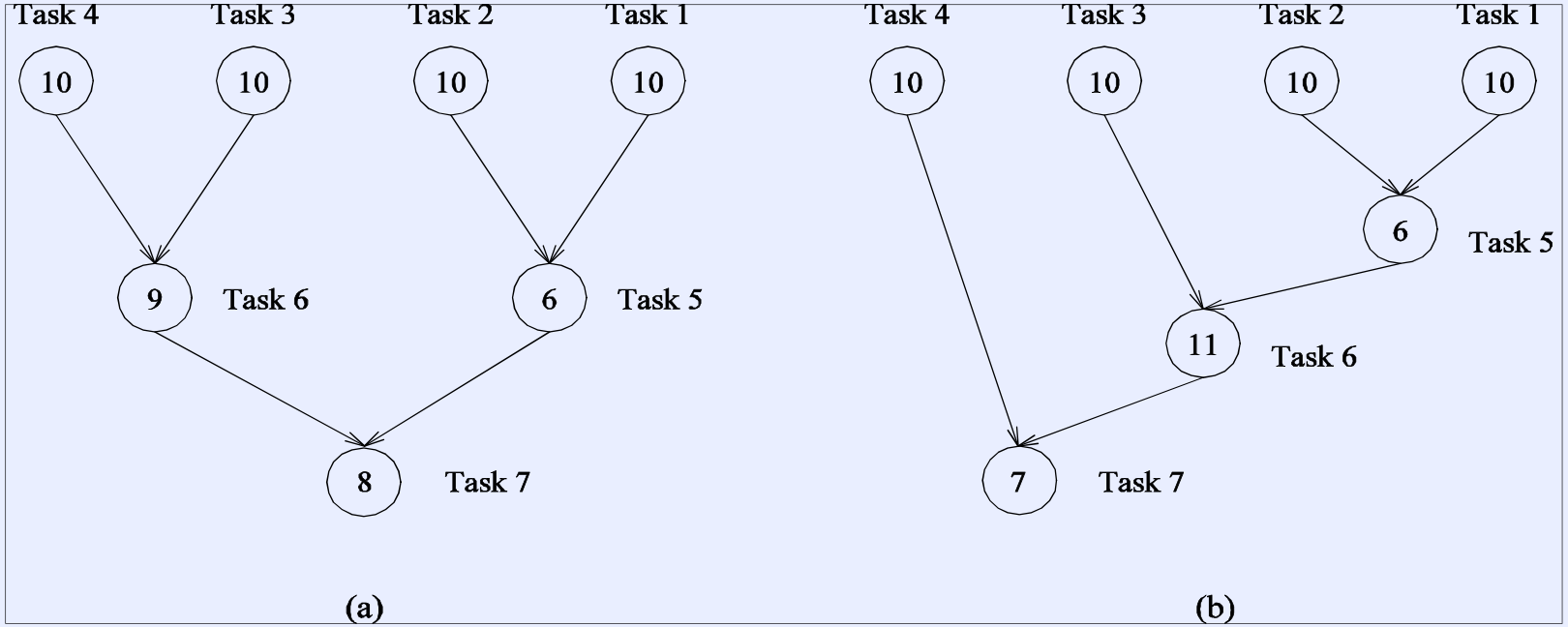
Parallel time >= max(depth, ceiling(W/p))

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 → sum of the weights (size of work) of nodes along the critical path.**
- Ratio of total work to the critical path length → **average degree of concurrency.**

Critical Path Length

Consider the following 2 task dependency graphs (decompositions):



- What are the critical path lengths for the two task dependency graphs?
- If each task takes 10 time units, what is the shortest parallel execution time for each decomposition? How many processors are needed in each case to achieve this minimum parallel execution time?
- What is the maximum degree of concurrency?
- 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 a dense matrix multiplication, there can be no more than* ***(n^2)*** *concurrent tasks.*
- Concurrent tasks may also have to exchange data with other tasks (distributed). 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, in the decomposition of a 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***.

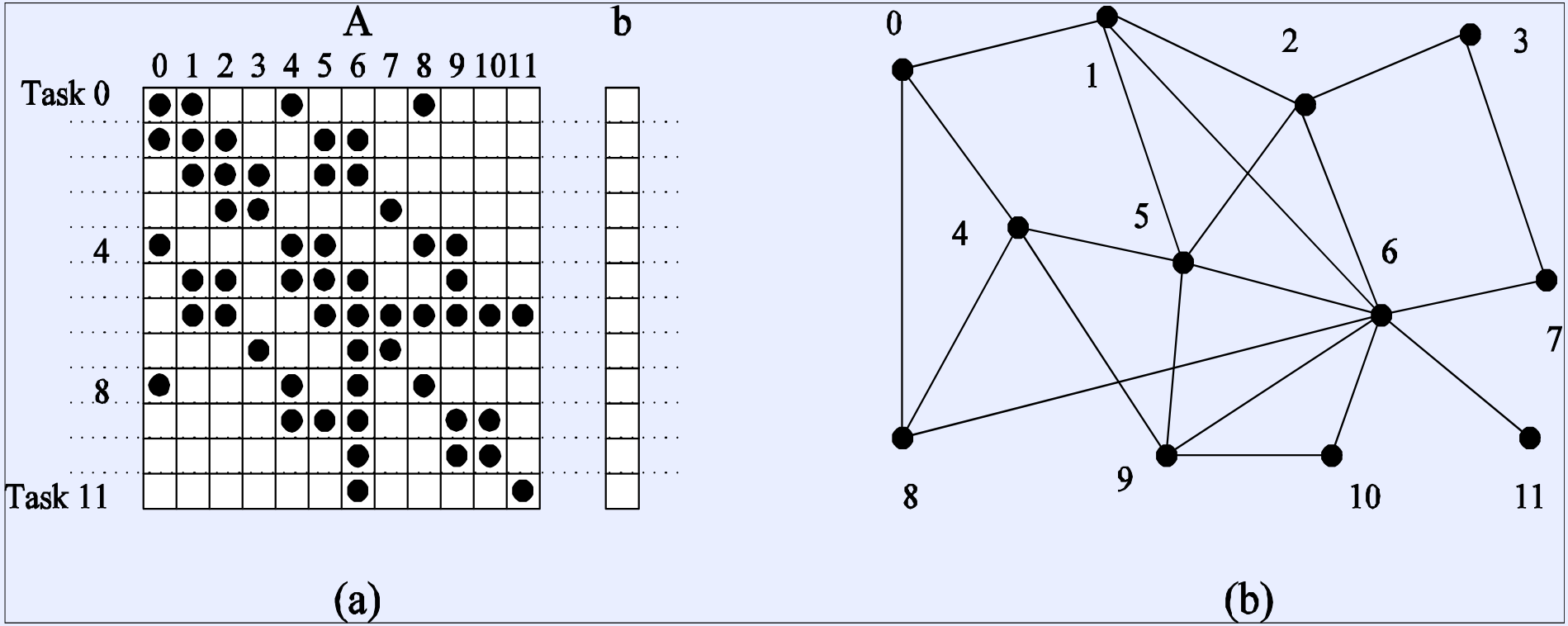
Task Interaction Graphs: An Example

Multiplication of a **sparse matrix *A*** with a vector ***b***. *Result R*

- computation of each element of the result vector is an independent task.
- Unlike a dense matrix-vector product though, only non-zero elements of matrix ***A*** participate in the computation.
- If, for memory optimality, we also partition ***b*** across tasks, then we can make a task interaction graph.

Task Interaction Graph: An Example

Multiplication of a **sparse matrix *A*** with a vector ***b***. *Result R*



Task interaction graph of the computation is identical to the graph of the matrix ***A.***

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

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

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 also provide a mapping of tasks to processes.

tasks → processes → physical processors.

Given the two graphs – what is the strategy ?

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 minimize parallel execution time by:

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

Decomposition Techniques

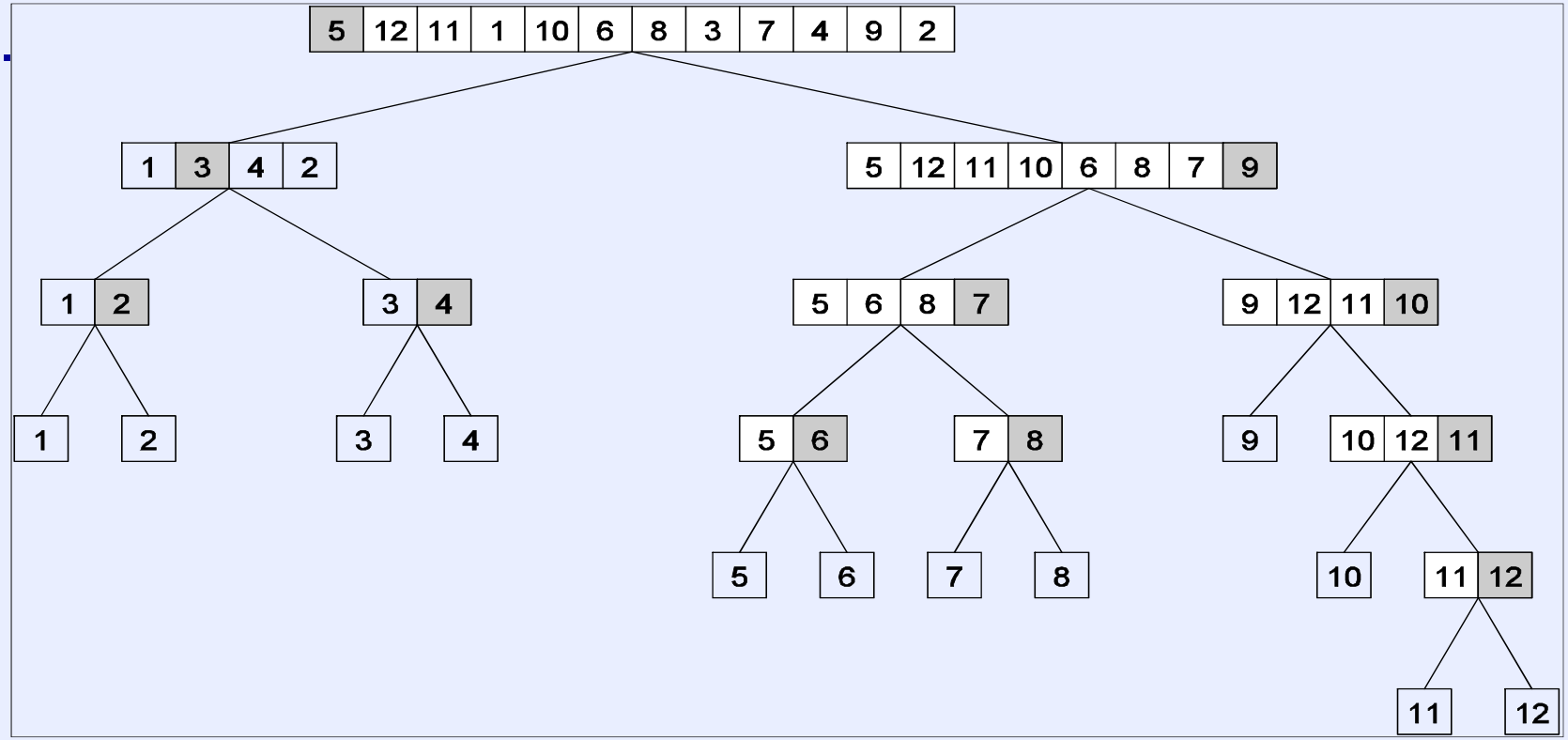
- recursive decomposition
- data decomposition
- exploratory decomposition
- speculative decomposition

Learning by example case studies.

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.
- Example - quicksort

Recursive Decomposition: Example - Quicksort



once the list has been partitioned around the pivot, each sub-list can be processed concurrently (each sublist - an independent subtask). This can be repeated recursively.

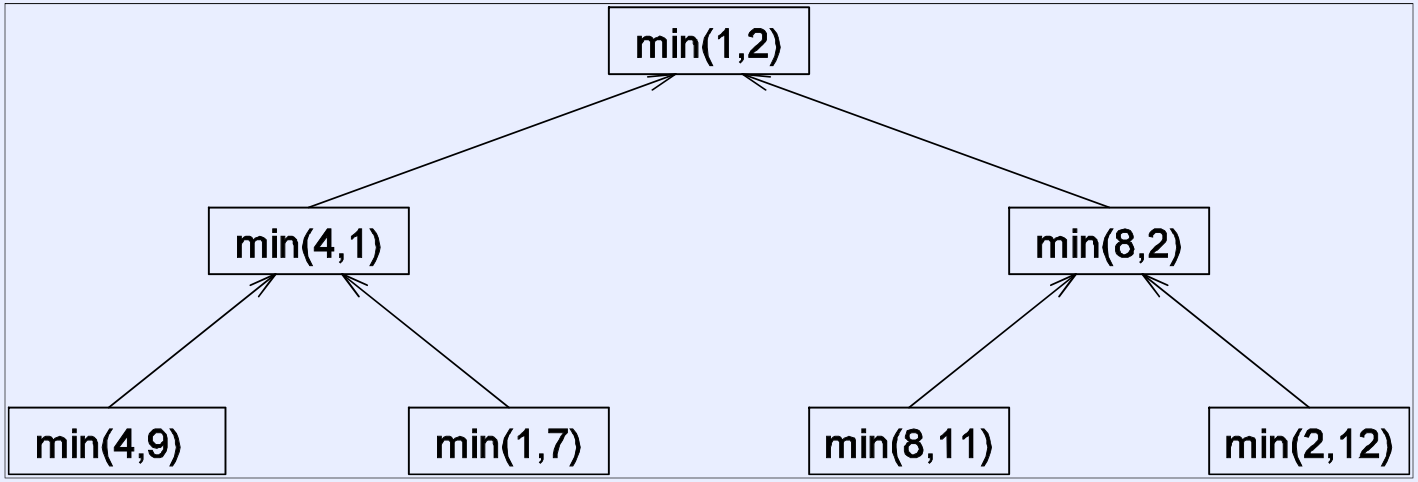
Recursive Decomposition: another Example

The problem of finding the minimum number in a given list (or indeed any other associative operation such as sum, AND, etc.) can be fashioned as a divide-and-conquer algorithm.

Recursive Decomposition: Example

recursive decomposition strategy for min.

finding the minimum number in the set {4, 9, 1, 7, 8, 11, 2, 12}. The task dependency graph associated with this computation is as follows:



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. Example?

- Example of input data partition?

Output Data Decomposition: Example

Consider the problem of multiplying two ***n*** x ***n*** matrices ***A*** and ***B*** to yield matrix ***C***. The output matrix ***C*** can be partitioned into four tasks as follows:

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

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

Output Data Decomposition: Example

A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem, with identical output data distribution, we can derive the following two (other) decompositions:

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

Input Data Partitioning

- Generally applicable if each output can be naturally computed as a function of the input.
- In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).
- A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing combines these partial results.

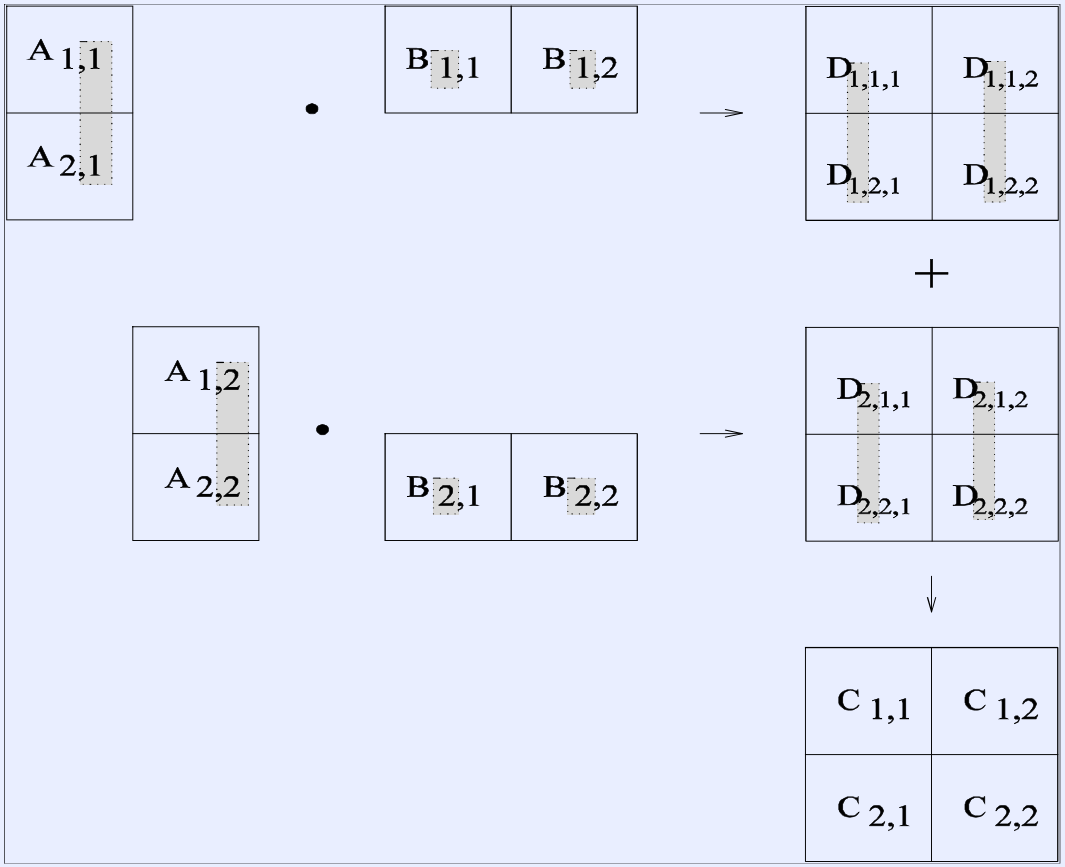
Intermediate Data Partitioning

- 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

example :dense matrix multiplication.

visualize this computation in terms of intermediate matrices ***D***.



Shape of the task dependency graph?

Intermediate Data Partitioning: Example

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

Stage I

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

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

Stage II

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

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

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

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

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

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

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

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

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

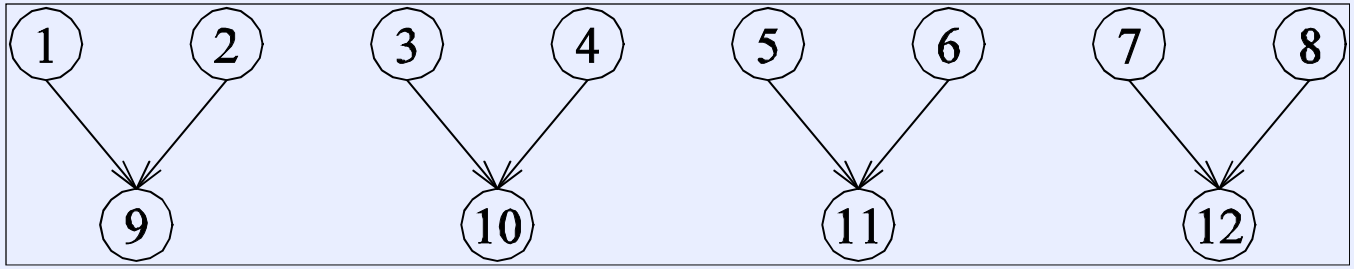
Task 10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

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

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

Intermediate Data Partitioning: Example

The task dependency graph for the decomposition (shown in previous foil) into 12 tasks is as follows:



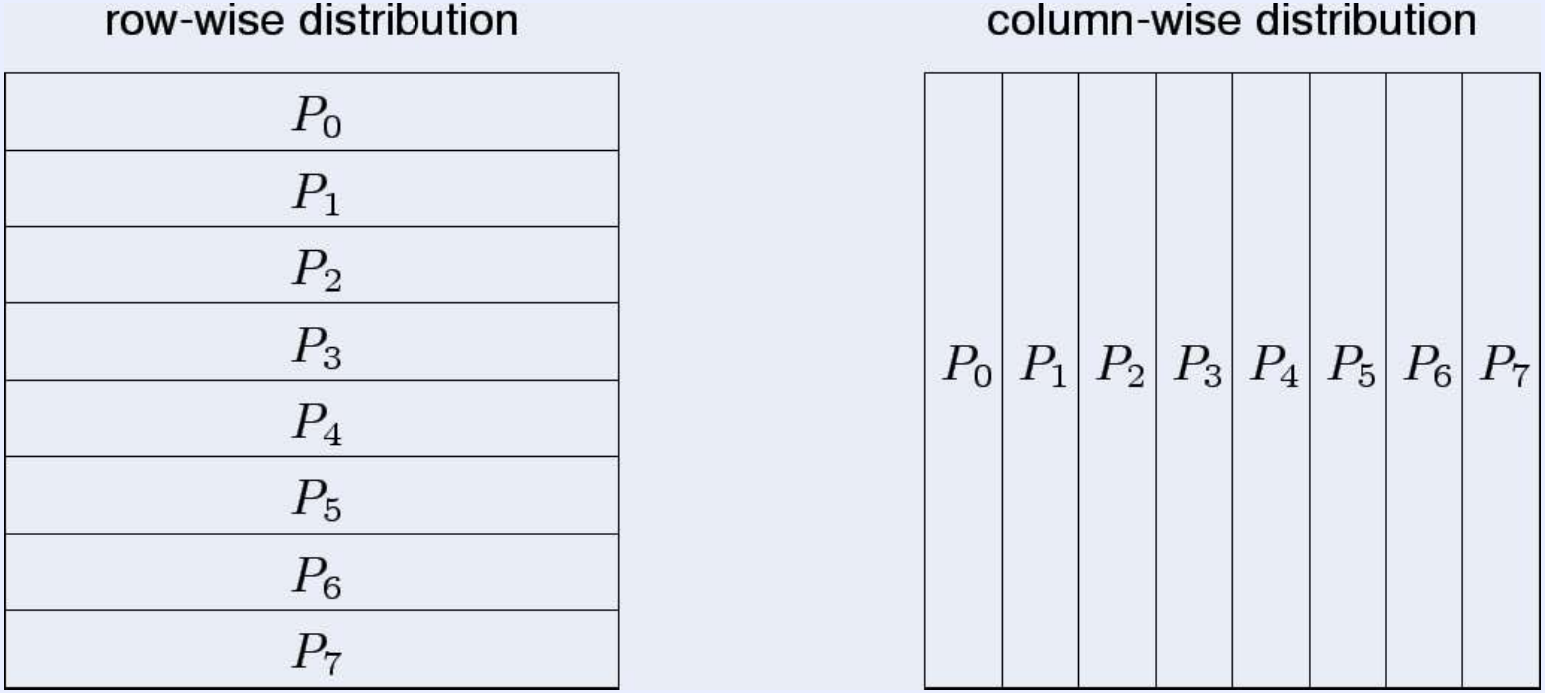
The Owner Computes Rule

- 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.

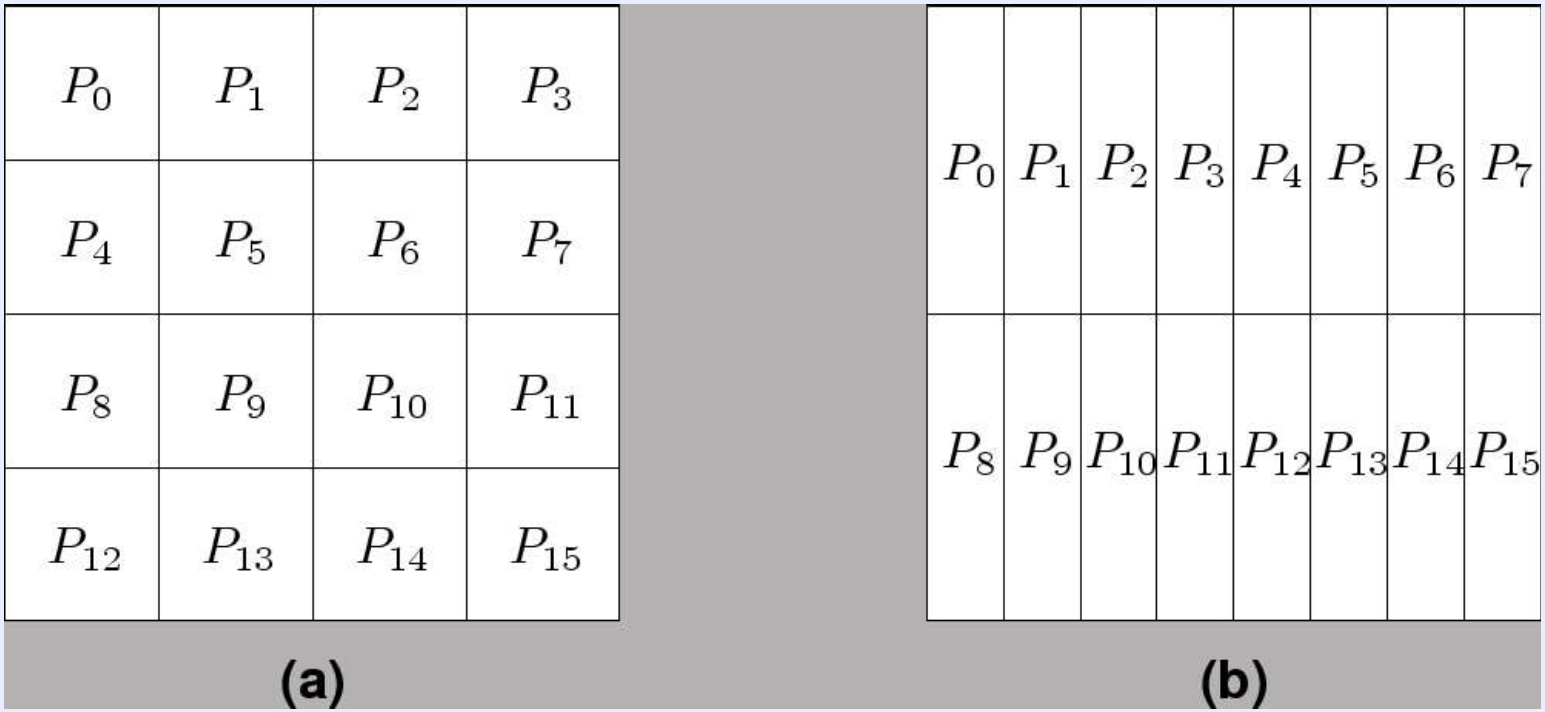
Static Mapping

➢ Mappings based on data partitioning.

The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.



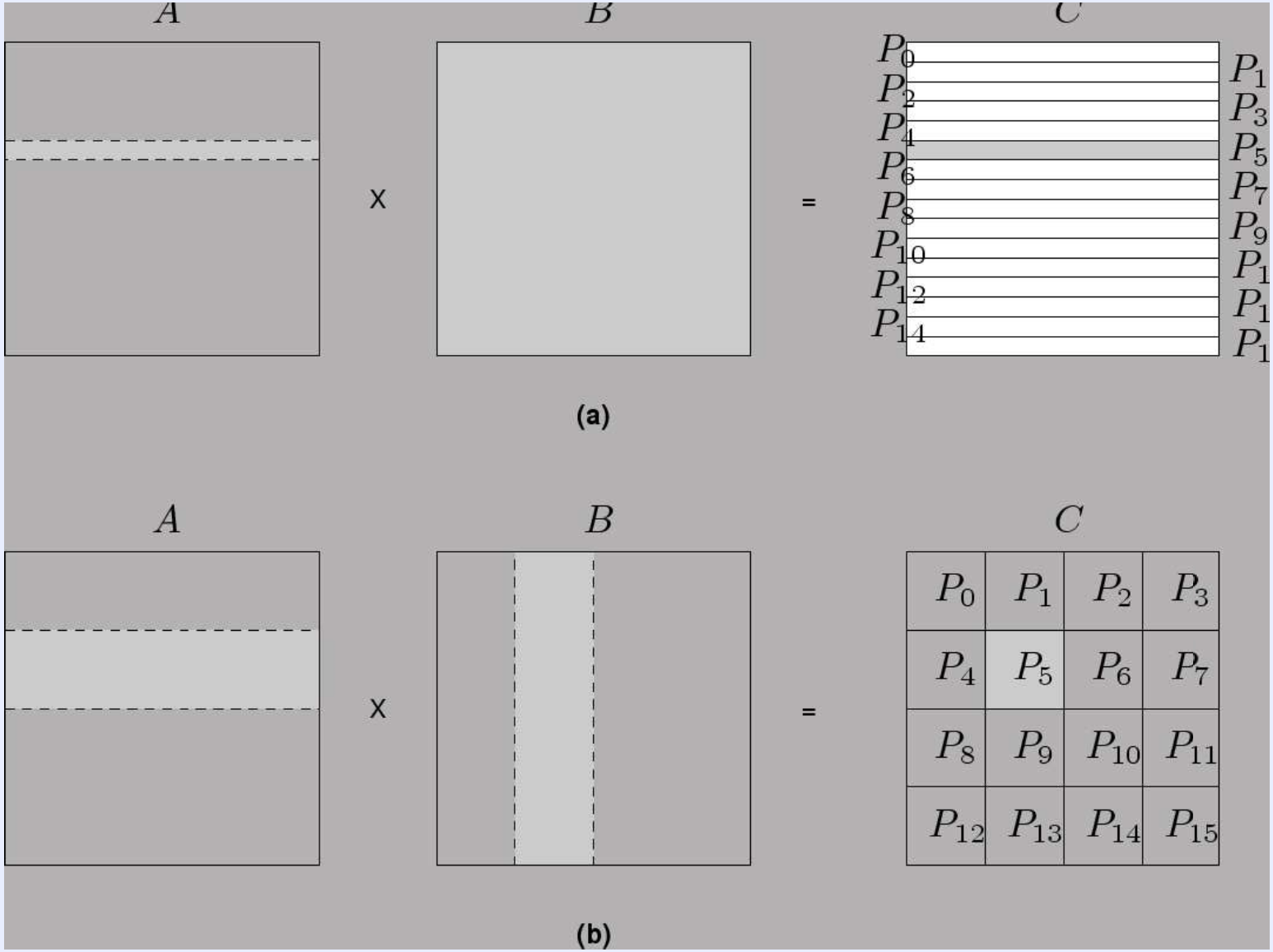
2D Block Array Distribution Schemes



Block Array Distribution Schemes: Examples

- For multiplying two dense matrices ***A*** and ***B***, we can partition the output matrix ***C*** using a block decomposition.
- For load balance, we give each task the same number of elements of ***C***.
- The choice of precise decomposition scheme determines the associated communication overhead.
- In general, higher dimension decomposition allows the use of larger number of processes.

Data Sharing in Dense Matrix Multiplication

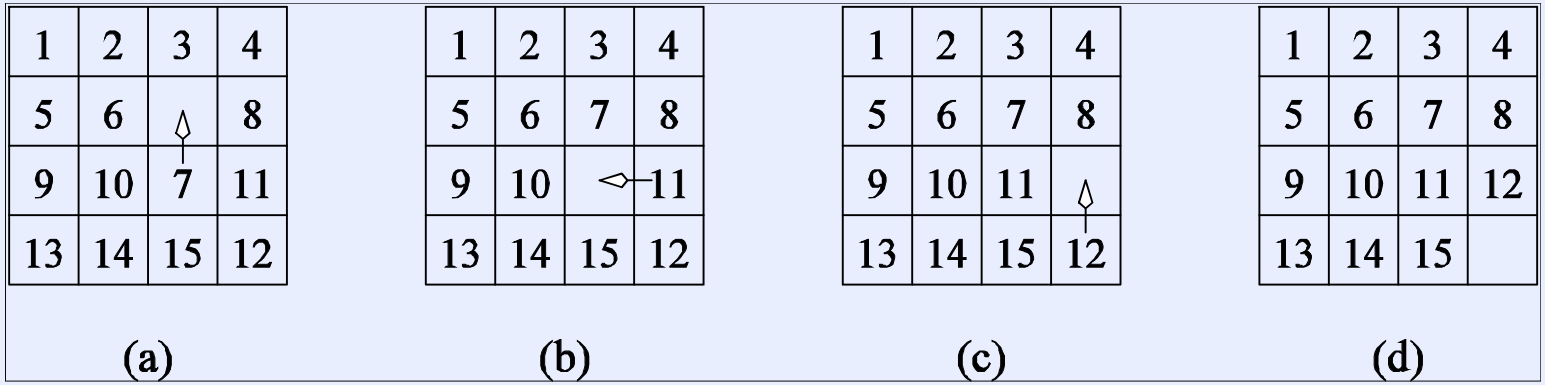


Exploratory Decomposition

- 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

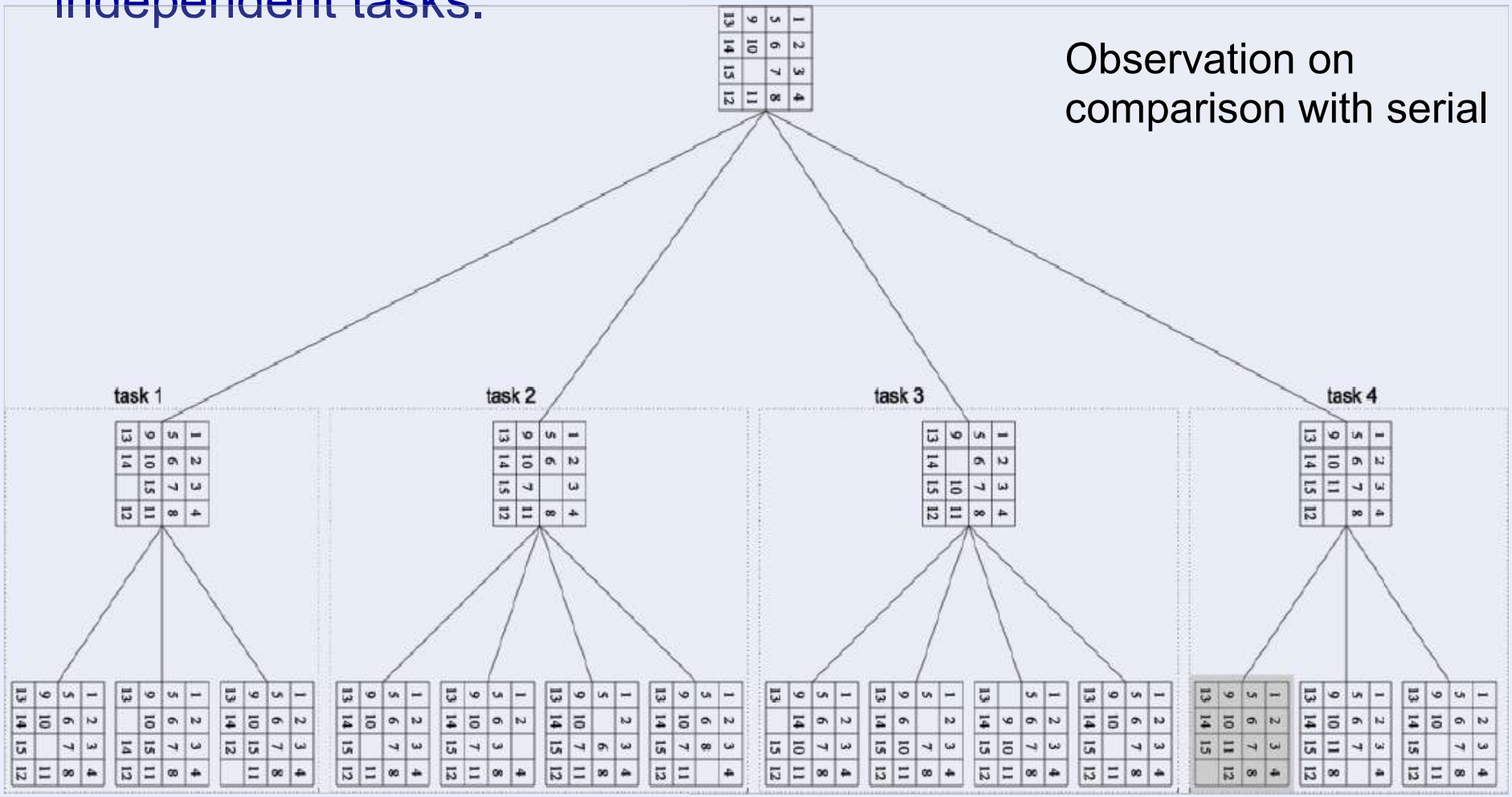
A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). Fig. show a sequence of three moves that transform a given initial state (a) to desired final state (d).



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

Exploratory Decomposition: Example

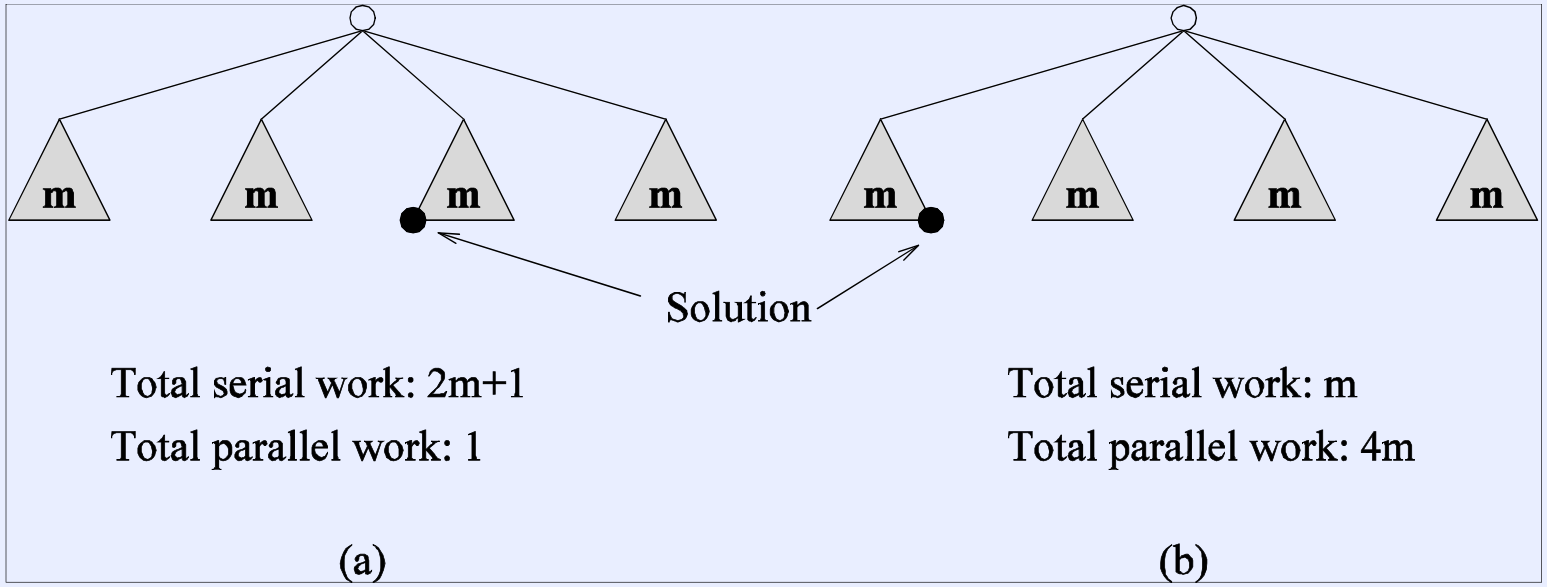
The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



Observation on comparison with serial

Exploratory Decomposition: Anomalous Computations

- In many instances of exploratory decomposition, the decomposition technique may change the amount of work done by the parallel formulation.
- This change results in super- or sub-linear speedups.



Speculative Decomposition

- Applications where dependencies between tasks are not known a-priori and it is impossible to identify independent tasks.

- 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.

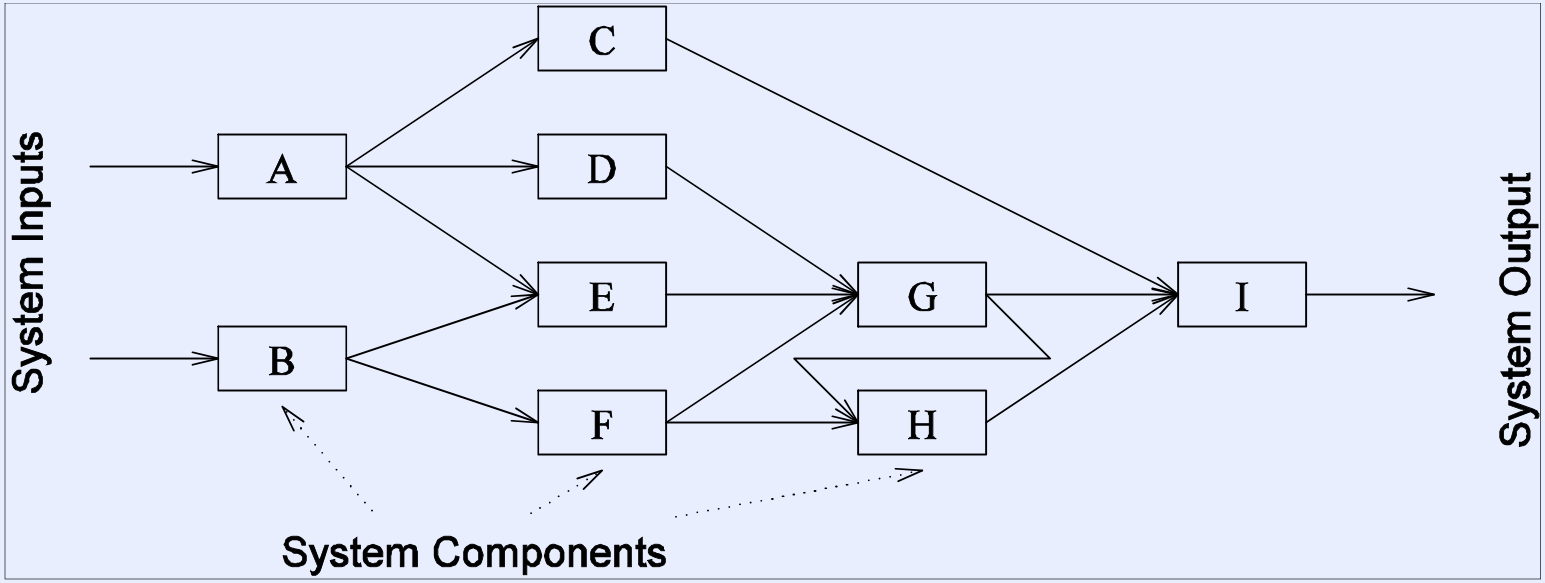
Speculative Decomposition: Example

Discrete event simulation

- The central data structure in a discrete event simulation is a time-ordered event list.
- 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.

Speculative Decomposition: Example

The task is to simulate the behavior of this network for various inputs and node delay parameters.



Hybrid Decompositions

mix of decomposition for decomposing a problem:

In quicksort, recursive decomposition alone limits concurrency (Why?).

A mix of data and recursive decompositions is more desirable.

- In discrete event simulation, there might be concurrency in task processing. A mix of speculative decomposition and data decomposition may work well.
- finding a minimum of a list of numbers, a mix of data and recursive decomposition works well.