TEXAS TECH UNIVERSITY

CS4379: Parallel and Concurrent Programming
CS5379: Parallel Processing

Lecture 10

Dr. Yong Chen

Associate Professor

Computer Science Department

Texas Tech University

Course Info

- **Lecture Time**: TR, 12:30-1:50
- **Lecture Location**: ECE 217
- **Sessions**: CS4379-001, CS4379-002, CS5379-001, CS5379-D01
- **Instructor:** Yong Chen, Ph.D., Associate Professor
- **Email:** yong.chen@ttu.edu
- **Phone:** 806-834-0284
- **Office**: Engineering Center 315
- **Office Hours**: 2-4 p.m. on Wed., or by appointment
- **TA:** Mr. Ghazanfar Ali, Ghazanfar.Ali@ttu.edu
- **TA Office hours:** Tue. and Fri., 2-3 p.m., or by appointment
- **TA Office:** EC 201 A
- **More info:**
  - http://www.myweb.ttu.edu/yonchen
  - http://discl.cs.ttu.edu; http://cac.ttu.edu/; http://nsfcac.org

Outline

- Questions?

- Quiz #1 and #2 review
- Principles of parallel algorithm design
- Decomposition techniques

Outline

- Questions?
- Quiz #1 and #2 review
- Principles of parallel algorithm design
- Decomposition techniques

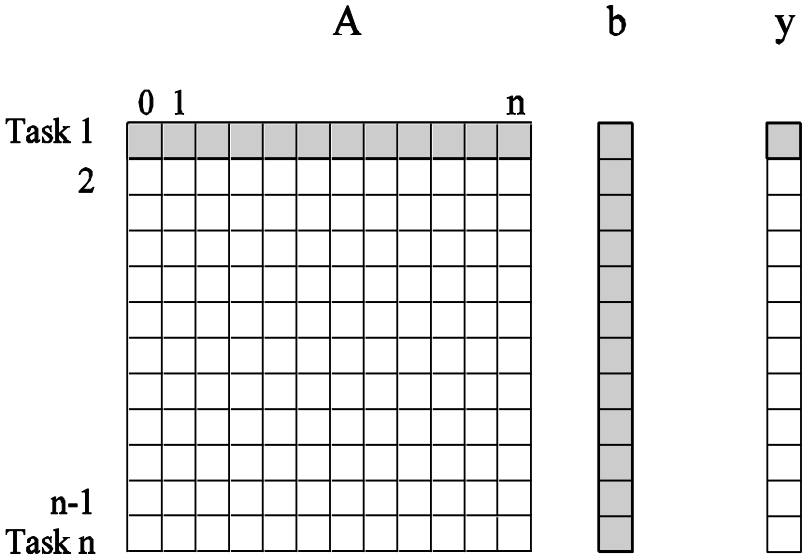
Overview of Decomposition

- A critical 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 ways
- Tasks may be of same or different 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

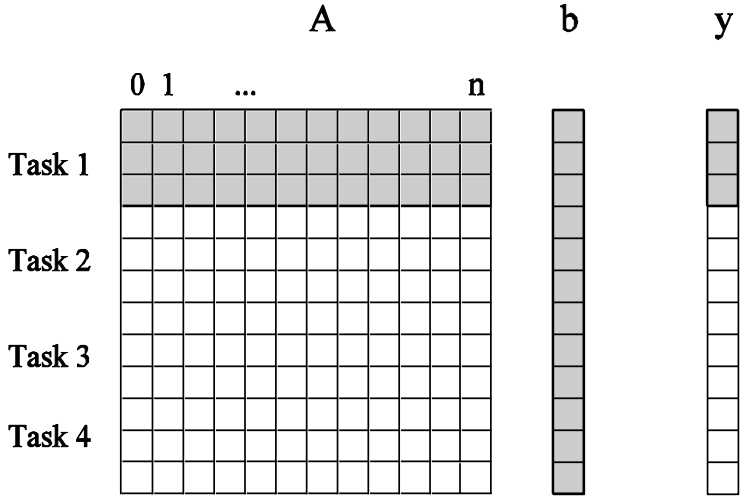


Computation of each element of output vector ***y*** is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into ***n*** tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

Observations: While tasks share data (namely, the vector ***b***), they do not have any 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

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

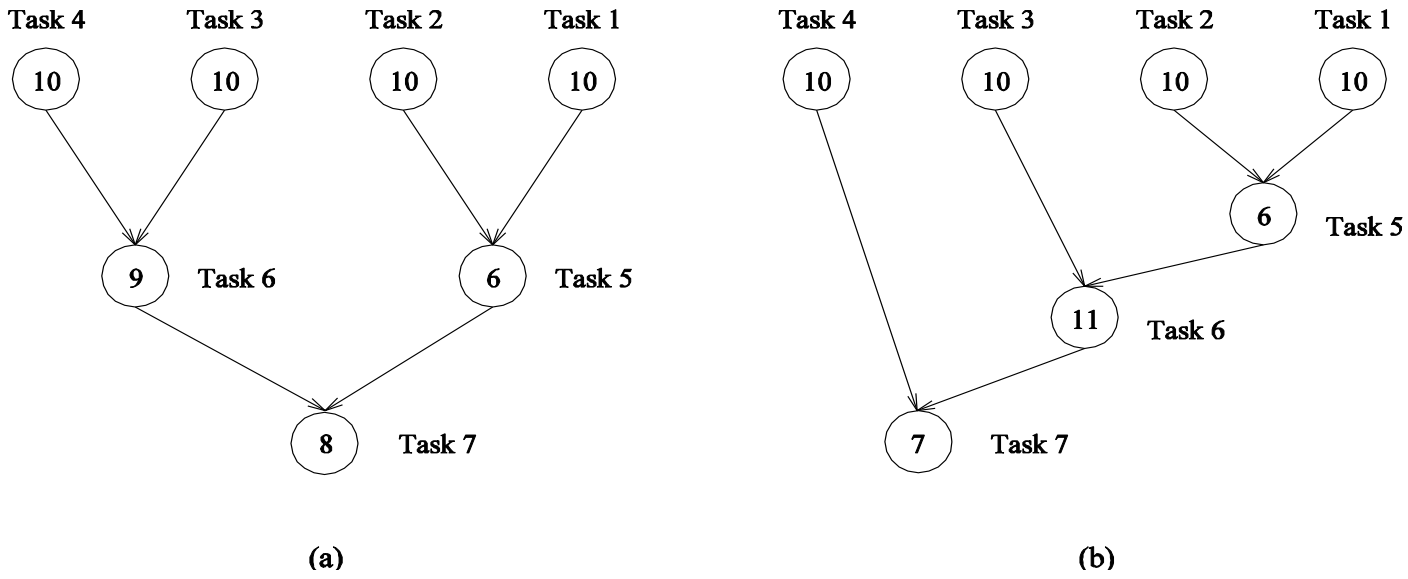
- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.
- Since the number of tasks that can be executed in parallel may change over program execution, the maximum degree of concurrency is the maximum number of such tasks at any point during execution
- 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

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 two decompositions:



What are the critical path lengths for the two task dependency graphs?

Decomposition Techniques

So how does one decompose a problem into various tasks?

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:

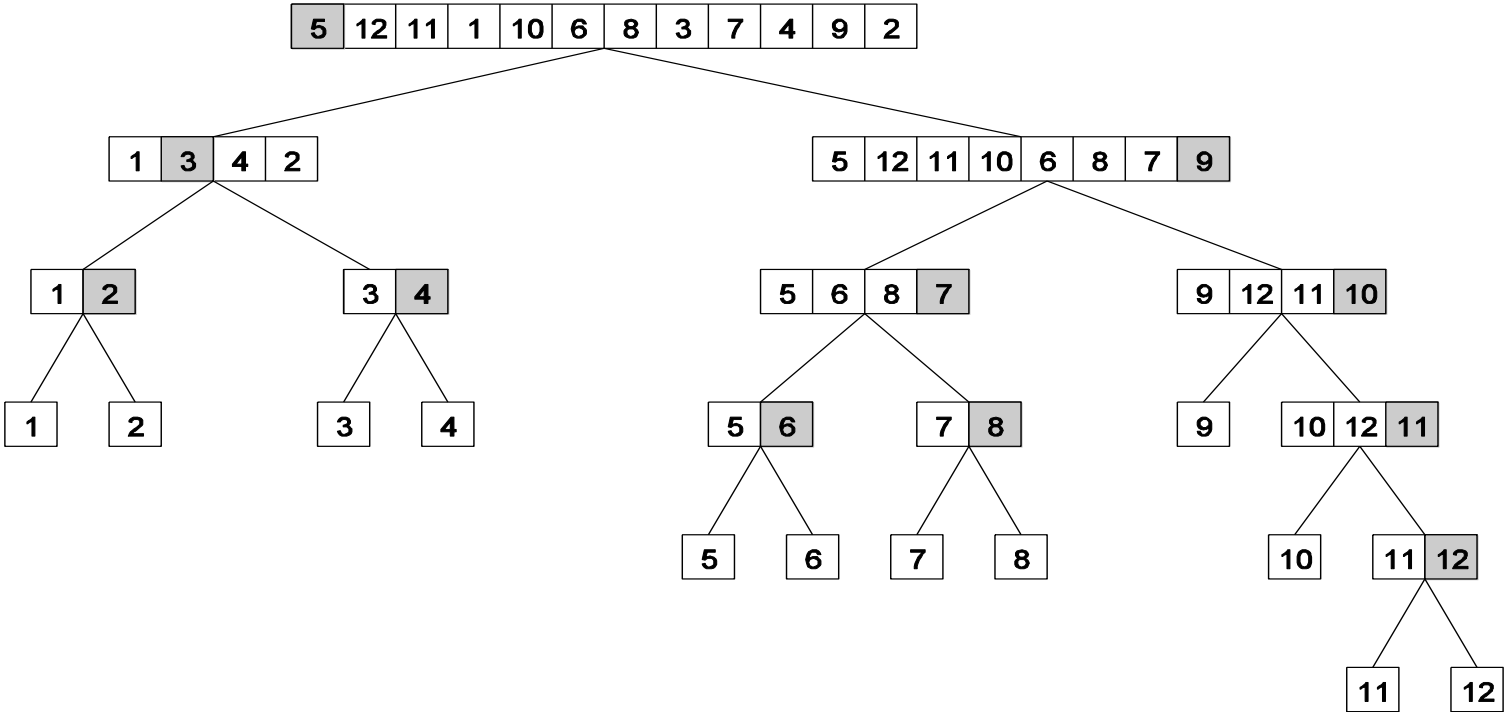
- Recursive decomposition
- Data decomposition
- Exploratory decomposition
- Speculative decomposition

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.

Recursive Decomposition: Example

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

We first start with a simple serial loop for computing the minimum entry in a given list:

```
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
```

Recursive Decomposition: Example

We can rewrite the loop as follows:

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3. if ( n = 1 ) then
4.     min := A [0]  ;
5. else
6.     lmin := RECURSIVE_MIN ( A, n/2 );
7.     rmin := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
8.     if (lmin  < rmin) then
9.                 min := lmin;
10.    else
11.                min := rmin;
12.    endelse;
13. endelse;
14. return min;
15. end RECURSIVE_MIN
```

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.

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.

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

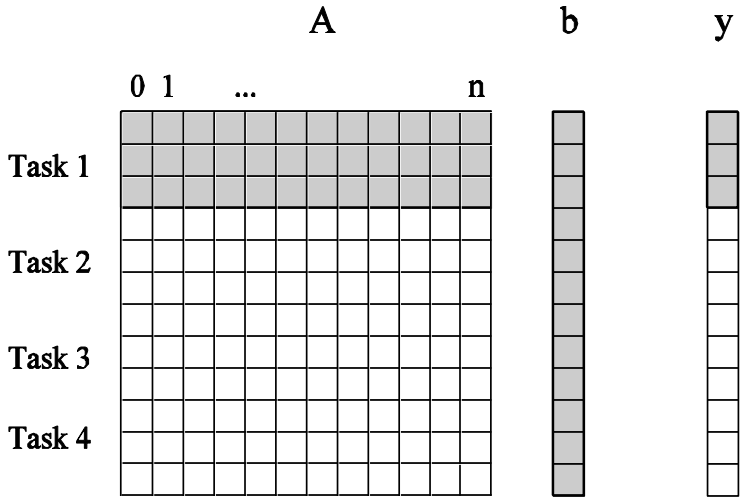
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

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

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

Input Data Decomposition

- 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.

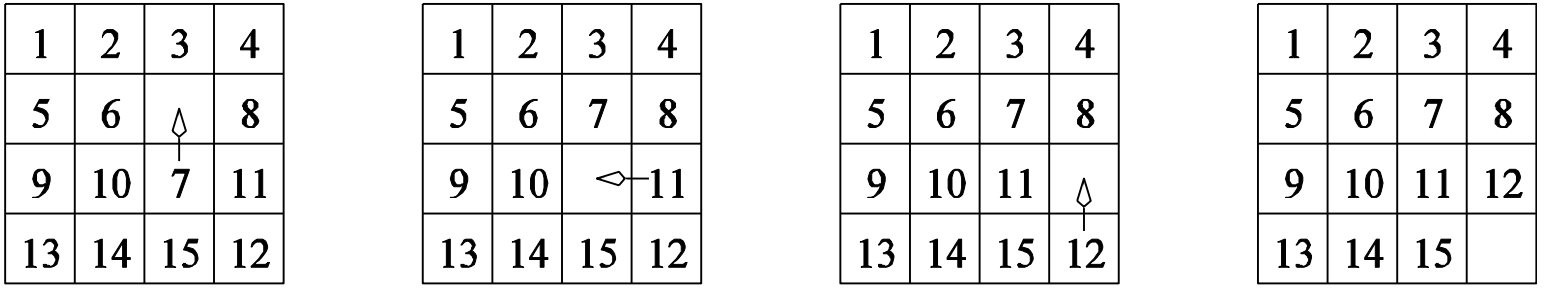


Exploratory Decomposition

- 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

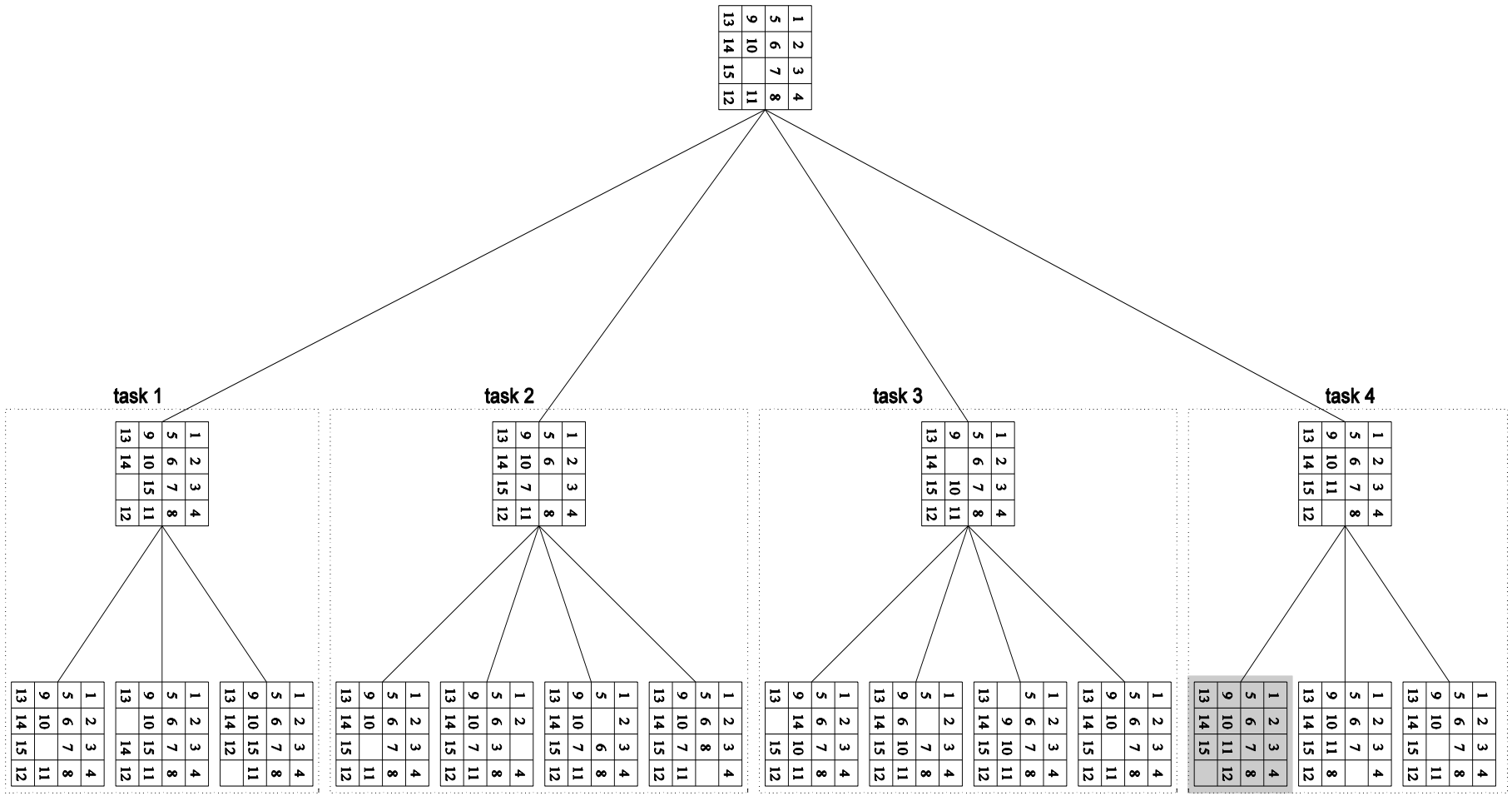
A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).



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

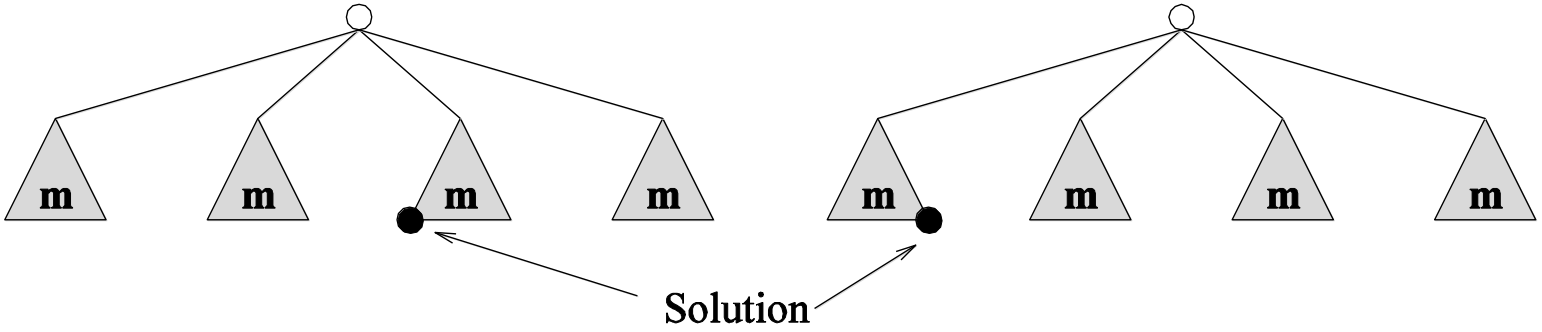
Exploratory Decomposition: Example

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



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.



Total serial work: 2m+1
Total parallel work: 1

(a)

Total serial work: m
Total parallel work: 4m

(b)

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
  - 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.

Readings

- Reference book ITPC – Chapter 3

Questions?

Questions/Suggestions/Comments are always welcome!

Write me: yong.chen@ttu.edu
Call me: 806-834-0284
See me: ENGCTR 315

If you write me an email for this class, please start the email subject with [CS4379] or [CS5379].