CZ4102 – High Performance Computing

Lectures 4 and 5: **Principles of Parallel Algorithm Design**

- Dr Tay Seng Chuan

Reference: ``Introduction to Parallel Computing'' – Chapter 3.

1

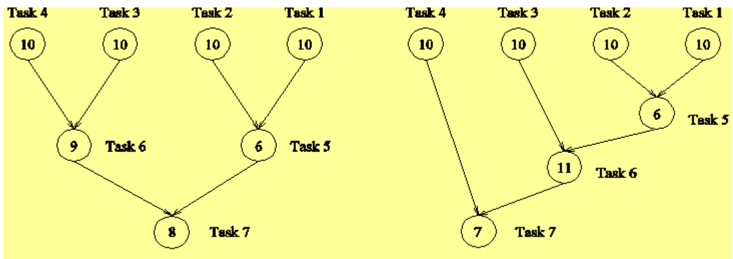
Chapter Overview: Algorithms and Concurrency

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

2

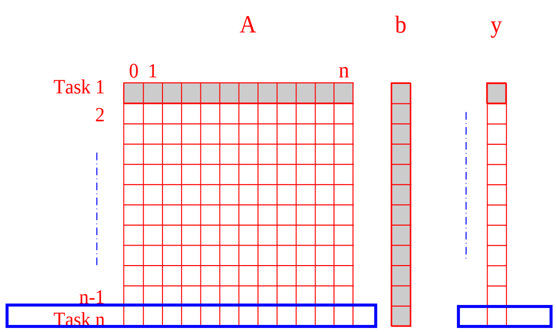
Preliminaries: Decomposition, Tasks, and Dependency Graphs

- 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 and 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***. Eg:



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 in this example, 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?*** **Answer?**

Example: Database Query Processing

Consider the execution of the query:

MODEL = ``CIVIC'' AND YEAR = 2001 AND (COLOR = ``GREEN'' OR COLOR = ``WHITE)

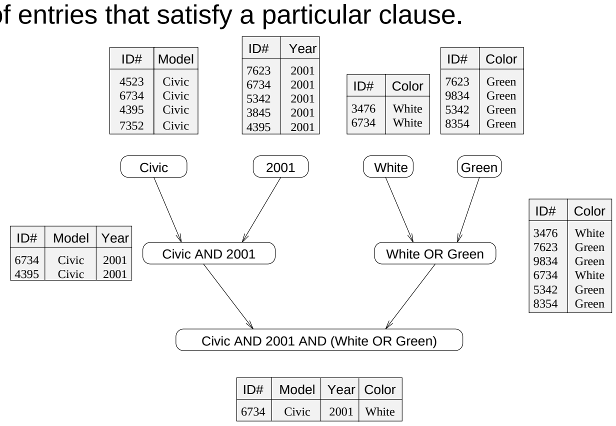
on the following database:

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

5

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.

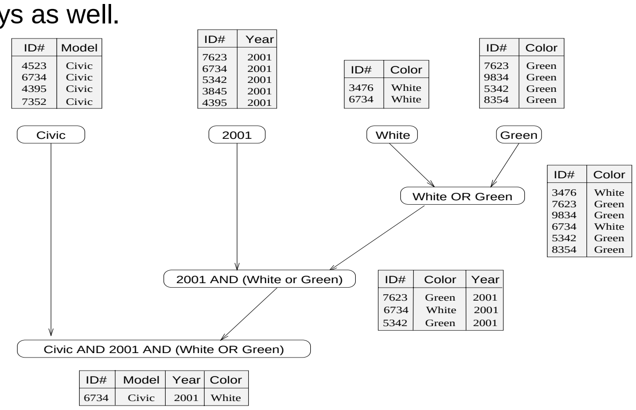


Decomposing the given query into a number of tasks. Edges in this graph denote that the **output of one task is needed to accomplish the next**.

6

Example: Database Query Processing

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



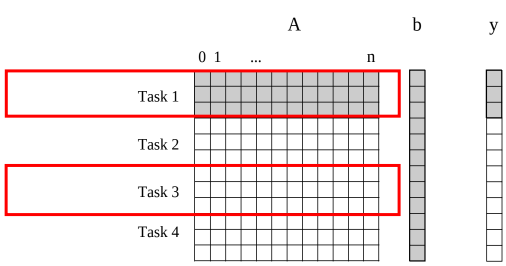
An alternate decomposition of the given problem into subtasks, along with their data dependencies.

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

7

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.

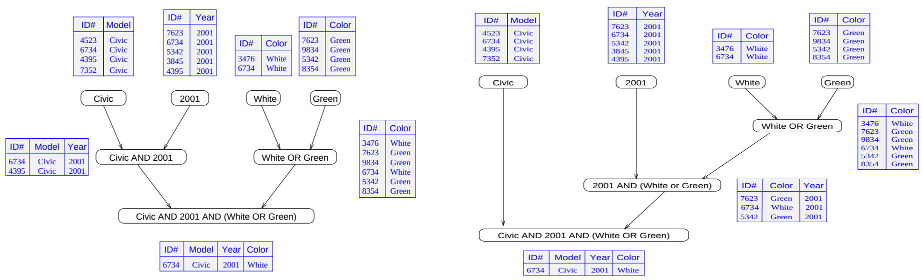


A coarse grained counterpart to the dense matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.

8

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?*

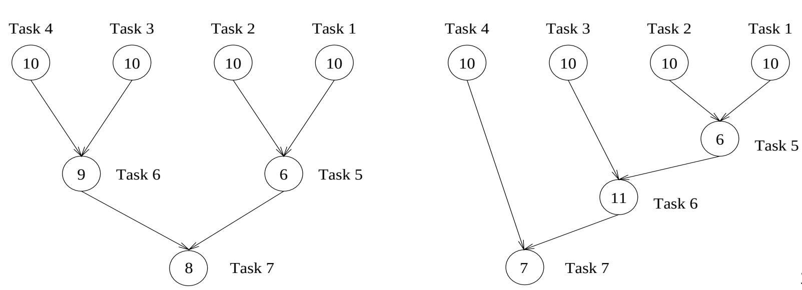


Answers?

9

Critical Path Length

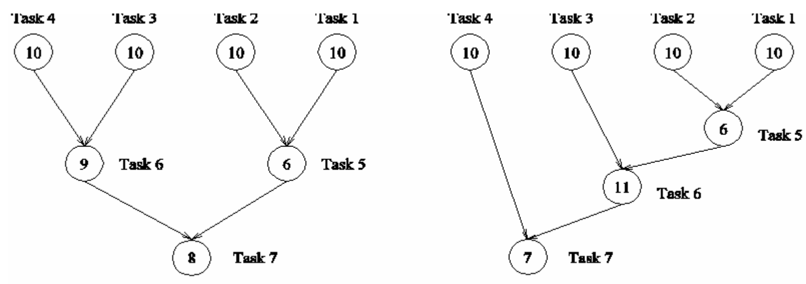
- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other. The number in the node represents the workload.
- The longest such path determines the shortest time (lower bound) in which the program execution can be completed in parallel.
- The length of the longest path (sum of the workload of the nodes) in a task dependency graph is called the critical path length. What is the critical path lengths in the following 2 directed graphs?



10

Average Degree of Concurrency

- Defined as the ratio of total amount of work to the critical path length.



Total amount of work =

Average degree of concurrency =

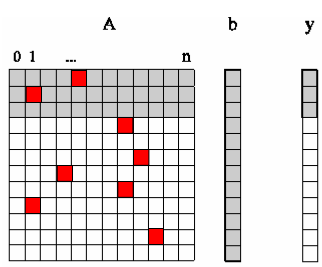
Total amount of work =

Average degree of concurrency =

11

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.*

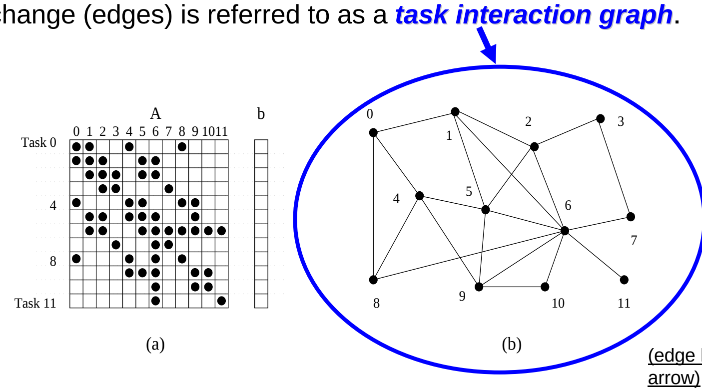


- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds.

12

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***.



(a) (b)

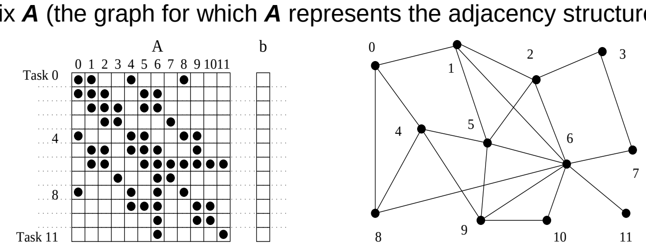
(edge has no arrow)

13

Task Interaction Graphs: An Example

Consider the problem of multiplying a sparse matrix ***A*** with a vector ***b****.* The following observations can be made:

- As before, the computation of each element of the result vector can be viewed as an independent task.
- Unlike a dense matrix-vector product though, only non-zero elements of matrix ***A*** participate in the computation. Blank in A represents 0.
- If, for memory optimality, we also partition ***b*** across tasks where each task-i contain b[i] and is responsible to multiply with A[i,*], then one can see that the task interaction graph of the computation is identical to the graph of the matrix ***A*** (the graph for which ***A*** represents the adjacency structure).

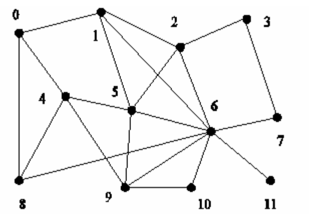


14

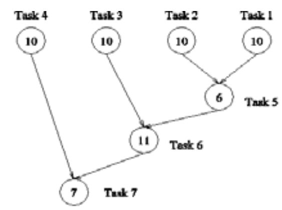
(a) (b)

Task Interaction Graphs

- Note that *task interaction graphs* represent data dependencies,



whereas *task dependency graphs* represent control dependencies.



15

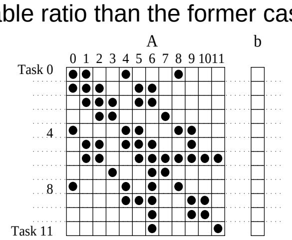
Task Interaction Graphs, Granularity, and Communication

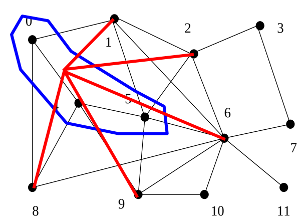
In general, if the granularity of a decomposition is finer, the associated overhead (as a ratio of useful work associated with a task) increases.

Example: Consider the sparse matrix-vector product example from previous foil. Assume that each node takes unit time to process and each interaction (edge) causes an overhead of a unit time.

Viewing node 0 as an independent task involves a useful computation of one time unit and overhead (communication) of three time units. (1:3)

Now, if we consider nodes 0, 4, and 5 as one task, then the task has useful computation totaling to three time units and communication corresponding to five time units (five edges) (3:5). Clearly, this is a more favorable ratio than the former case.

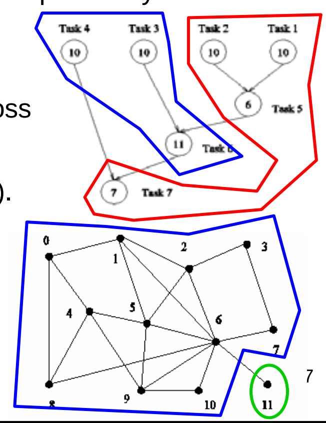




16

Processes and Mapping

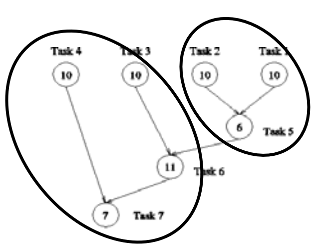
- In general, the number of tasks in a decomposition exceeds the number of processing elements available.
- Appropriate mapping of tasks to processes is critical to the parallel performance of an algorithm.
- Mappings are determined by both the task dependency and task interaction graphs.
- Task dependency graphs can be used to ensure that work is as equally spread across all processes at any point as possible (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).

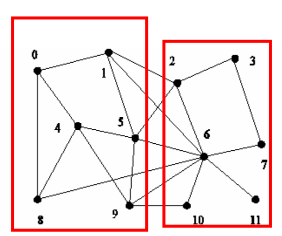


Processes and Mapping

An appropriate mapping must minimize parallel execution time by:

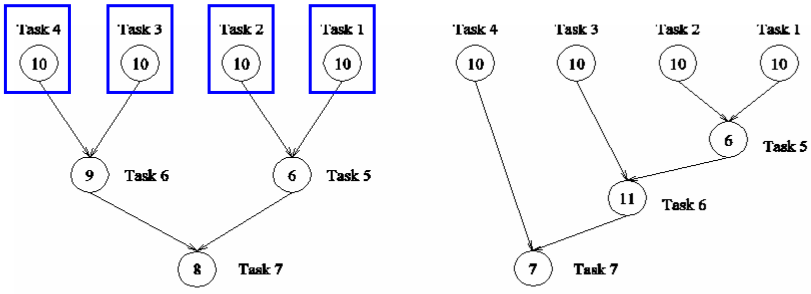
- Mapping independent tasks to different processes.
- Assigning tasks on critical path to processes as soon as they become available.
- Minimizing interaction between processes by mapping tasks with dense interactions to the same process.
- **<u>These criteria often conflict each other (see the partitions in the previous slides)</u>**





18

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.

19

Decomposition Techniques

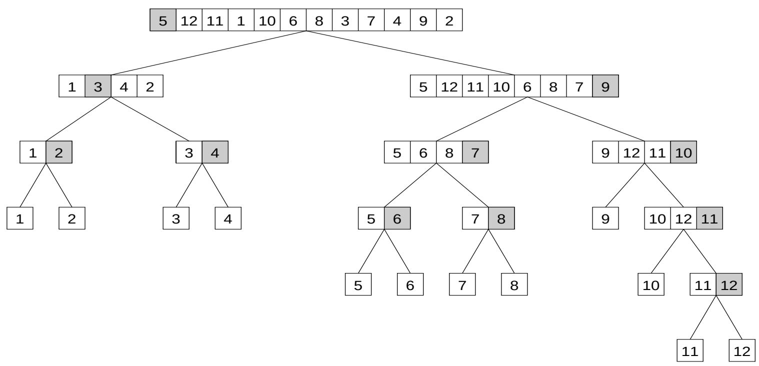
Commonly used techniques:

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

20

Recursive Decomposition

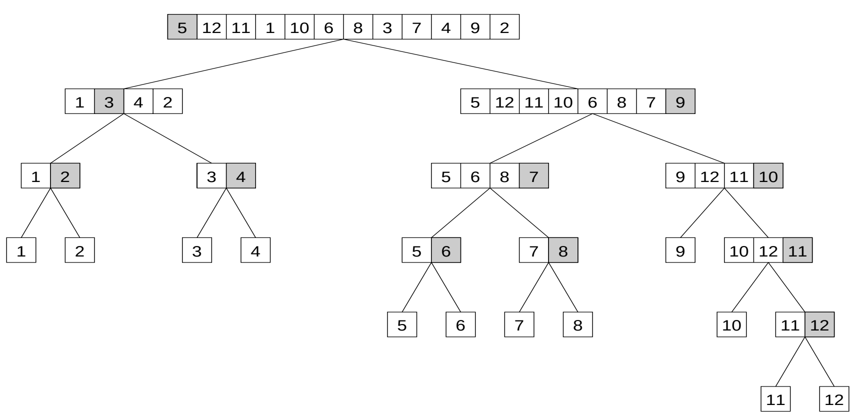
- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.
- A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.



21

Recursive Decomposition: Quicksort

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



22

Recursive Decomposition: Finding Minimum

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. The following algorithm illustrates this.

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

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

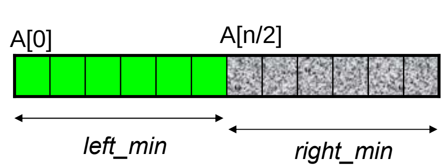
5	6	3	54	77	24	8	3	59	2	5	72

23

Recursive Decomposition: Example

We can rewrite the loop as follows:

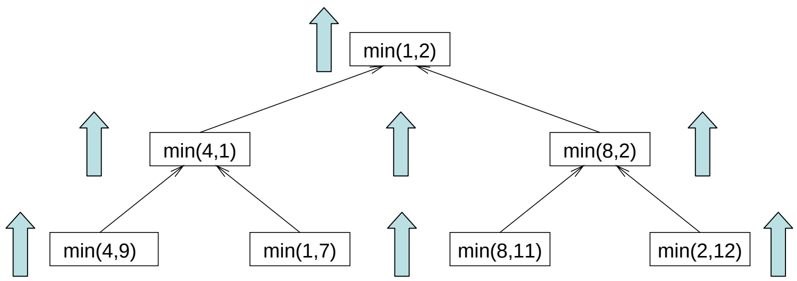
```
procedure RECURSIVE_MIN (A, n)
begin
if ( n = 1 ) then
    min := A [0] ;
else
    left_min := RECURSIVE_MIN ( A, n/2 );
    right_min := RECURSIVE_MIN (  &(A[n/2]), n - n/2 );
    if (left_min  < right_min) then
            min := left_min;
    else
            min := right_min;
    endelse;
endelse;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



24

Recursive Decomposition: Example

The code in the previous parallel can be decomposed naturally using a recursive decomposition strategy. We illustrate this with the following example of 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:



25

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.

26

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: Non unique

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

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

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. E.g., the 15-square puzzle:

(solved state)

Exploratory Decomposition: Example

A simple application of exploratory decomposition is in the solution to a 15-square 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)

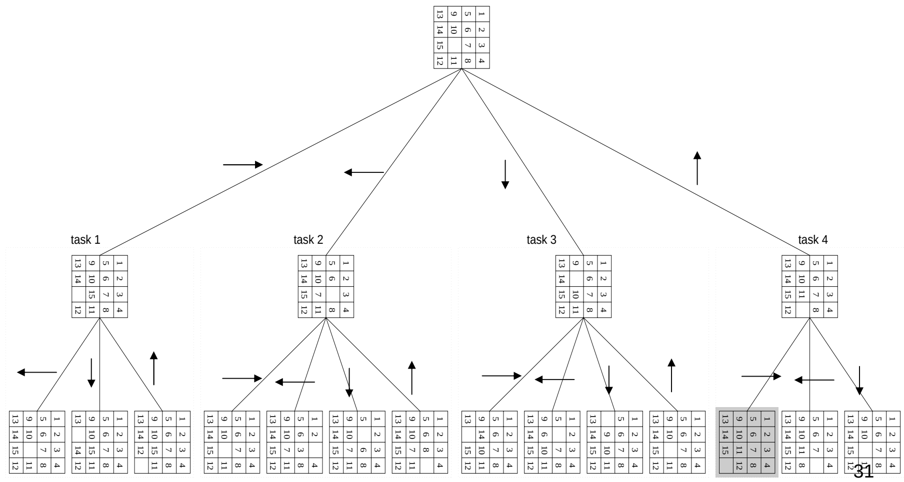
15-square puzzle is a NP-complete problem. Some initial arrangements do not have a solution. E.g.,

	1
2	3

1	2
3	

Exploratory Decomposition: Example

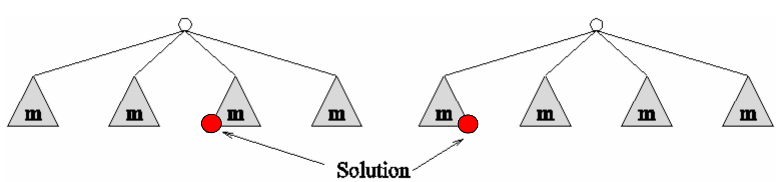
The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



31

Exploratory Decomposition: Anomalous Computations

- In many instances of exploratory decomposition, the decomposition technique may change the performance for different instance.
- This change results in different computation times thus different speedups.
- E.g.. Computation in left-to-right order.



Serial time:

Parallel time:

Serial time:

Parallel time:

32

Speculative Decomposition

- In some applications, dependencies between tasks are not known in advance.
- 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.

33

Speculative Decomposition: Example

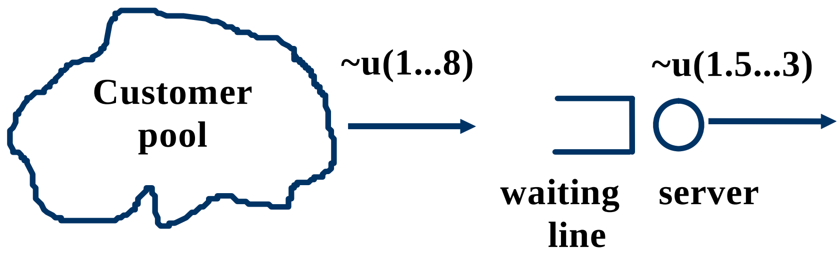
A classic example of speculative decomposition is in discrete event simulation.

- 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.

34

Example: Simple Queue

Event: Arrival, Departure



How and when to schedule arrival events?

```
increment the number of arrivals by 1;
schedule the next arrival event;
if server is idle
  {
    set server to busy;
    schedule a departure event for this arrival;
  }
else
  increment queue length by 1;
```

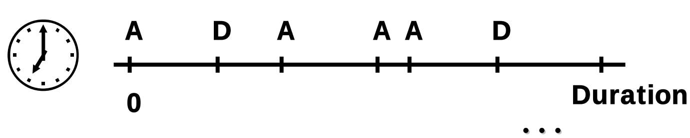
How and when to schedule departure events?

```
if queue length is 0
  {
    set  server to idle;
    set departure time to infinity;
  }
else
  {
    decrement queue length by 1;
    schedule the next departure event;
  }
```

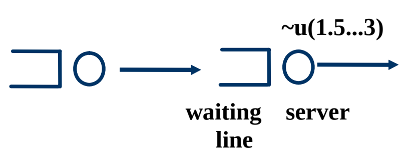
35 37

The Mechanism

1. The increment of clock time is discrete.
2. We are doing almost the same things in each iteration:
   - select the event (arrival or departure) of least time;
   - update the system state
   - advance clock time
   - execute event



What if arrival is from a service station?



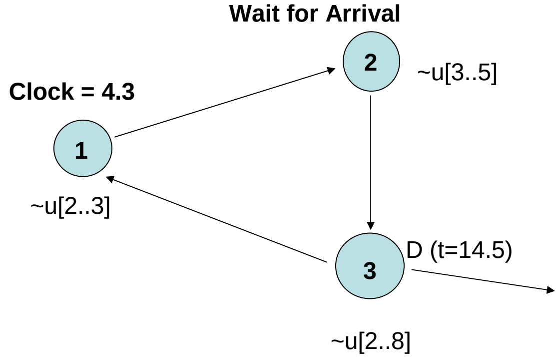
38

Conservative Approach

(Null Message)

Clock = 2.1

Wait for Arrival



P2 send null (t=5.1) to P3
P3 send null (t=7.1) to P1
P1 send null (t=9.1) to P2
P2 send null (t=12.1) to P3
P3 send null (t=14.1) to P1
P1 send null (t=16.1) to P2
P2 send null (t=19.1) to P3
D (t=14.5) is executed by P3

Clock = 4.2

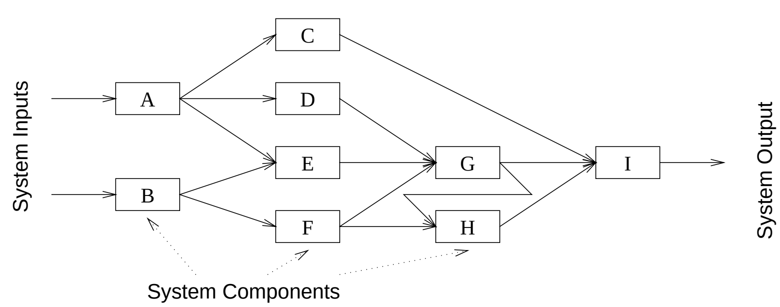
Departure event to be execute at time 14 but

to execute or not?

39

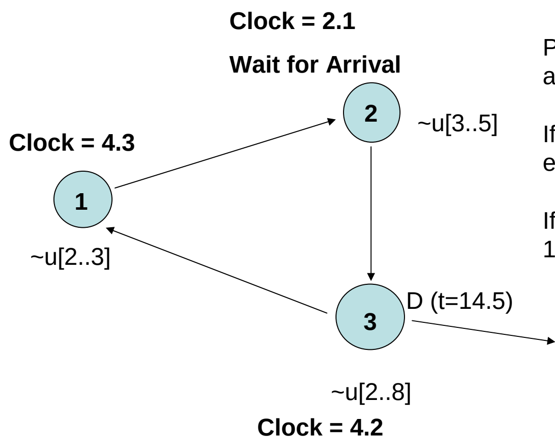
Speculative Decomposition: Example

Another example is the simulation of a network of nodes (for instance, an assembly line or a computer network through which packets pass). The task is to simulate the behavior of this network for various inputs and node delay parameters (note that networks may become unstable for certain values of service rates, queue sizes, etc.).



40

Optimistic (Speculative) Approach



P3 executes Departure and advance clock to 14.5

If the arrival time > 14.5, no error.

If the arrival time is less than 14.5, rollback.

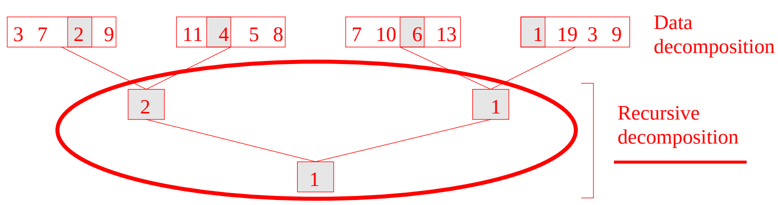
Departure event to be execute at time 14.

41

Hybrid Decompositions

Often, a mix of decomposition techniques is necessary for decomposing a problem. Consider the following examples:

- In quick sort, 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.
- Even for simple problems like finding a minimum of a list of numbers, a mix of data and recursive decomposition works well.



42

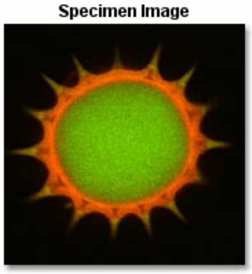
Task Generation

- Static task generation: Concurrent tasks can be identified a-priori. Typical matrix operations, graph algorithms, image processing applications, and other regularly structured problems fall in this class. These can typically be decomposed using data or recursive decomposition techniques.
- Dynamic task generation: Tasks are generated as we perform computation. A classic example of this is in game playing - each 15 puzzle board is generated from the previous one. These applications are typically decomposed using exploratory or speculative decompositions.

43

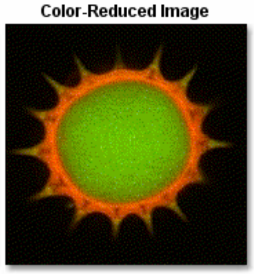
Characteristics of Task Interactions: Example

Specimen Image



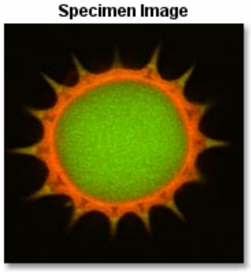
8704 Colors

Color-Reduced Image



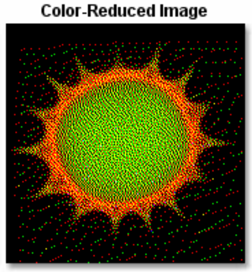
56 Colors

Specimen Image



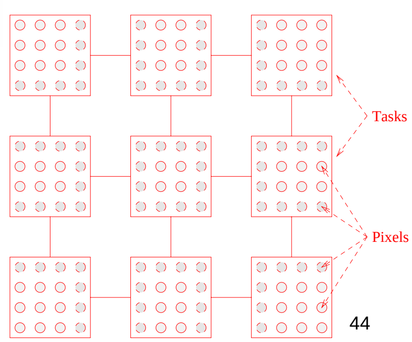
8704 Colors

Color-Reduced Image



8 Colors

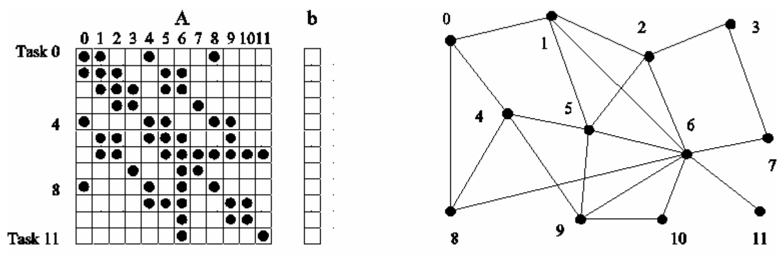
A simple example of a regular static interaction pattern is in image dithering used when display capacity is less than representation capacity. The underlying communication pattern is a structured (2-D mesh) one as shown here:



44

Characteristics of Task Interactions: Example

The multiplication of a sparse matrix with a vector is a good example of a static irregular interaction pattern. Here is an example of a sparse matrix and its associated interaction pattern.



Characteristics of Task Interactions

- Interactions may be read-only or read-write.
- In read-only interactions, tasks just read data items associated with other tasks.
- In read-write interactions, tasks read as well as modify data items associated with other tasks.
- In this course these interactions will be implemented by message passing.