

# CS3006 Parallel and Distributed Computing

---

FALL 2022

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES



---

# Chapter 3. Principles of Parallel Algorithm Design

- 
- A Parallel algorithm is a recipe that tells us how to solve a given problem using multiple processors.
- At the very least, **concurrency** should be there and the algorithm designer must specify sets of steps that can be executed **simultaneously**.

# Constructing a Parallel Algorithm

---

- Identifying portions of the work that can be performed concurrently
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

# Decomposition, Tasks, and Dependency Graphs

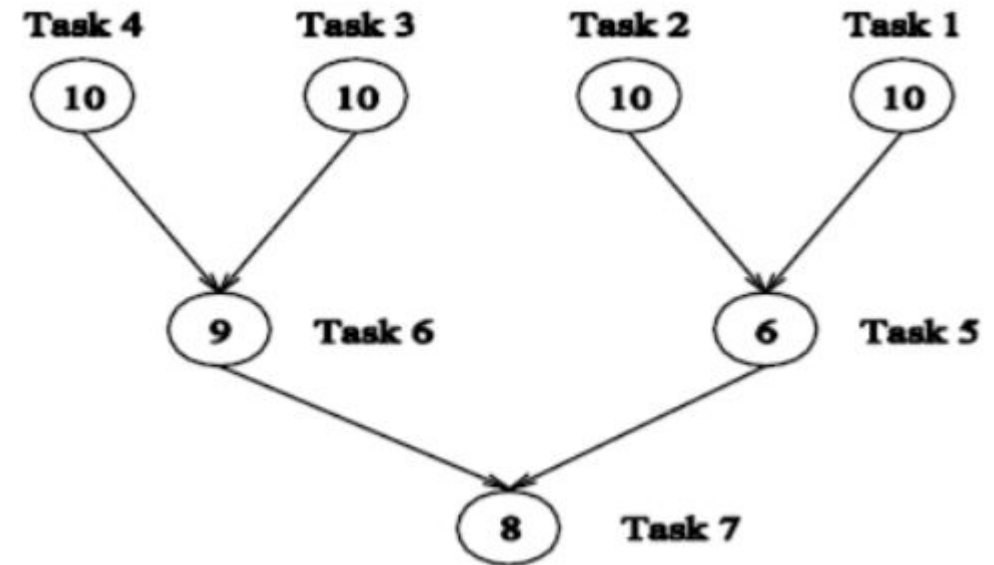
---

- **Decomposition:** dividing a computation into smaller parts, some or all of which may potentially be executed in parallel.
- **Tasks** are programmer-defined units of computation .
  - Tasks can be of arbitrary size, but once defined, they are regarded as **indivisible** units of computation.
- In general, some tasks may use data produced by other tasks and thus may need to wait for these tasks to finish execution.
  - An abstraction used to express such dependencies among tasks and their relative order of execution is known as a ***task dependency graph***.

# Dependency Graph

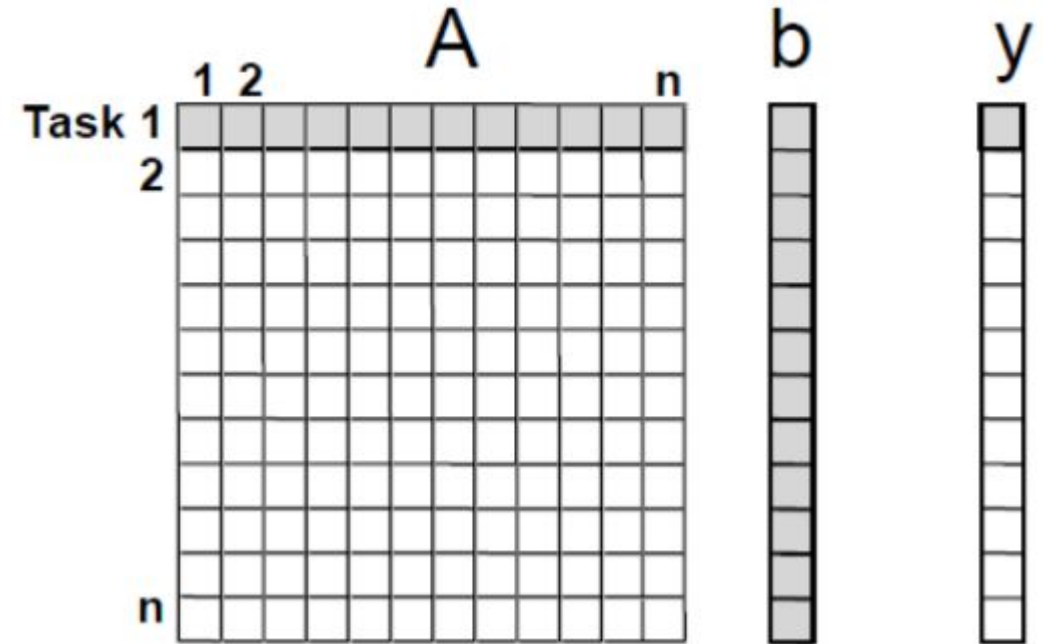
---

- **Node** represents task.
- **Directed edge** represents control dependence.



# Example 1: Dense Matrix-Vector Multiplication

- The computation of each  $y[i]$  can be regarded as a task.
- All tasks are independent and can be performed all together or in any sequence.



$$y[i] = \sum_{j=1}^n A[i, j].b[j]$$

# Example 3.2 Database query processing

Executing the query: (on the following database)

Model = "civic" **AND** Year = "2001" **AND** (Color = "green" **OR** Color = "white")

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

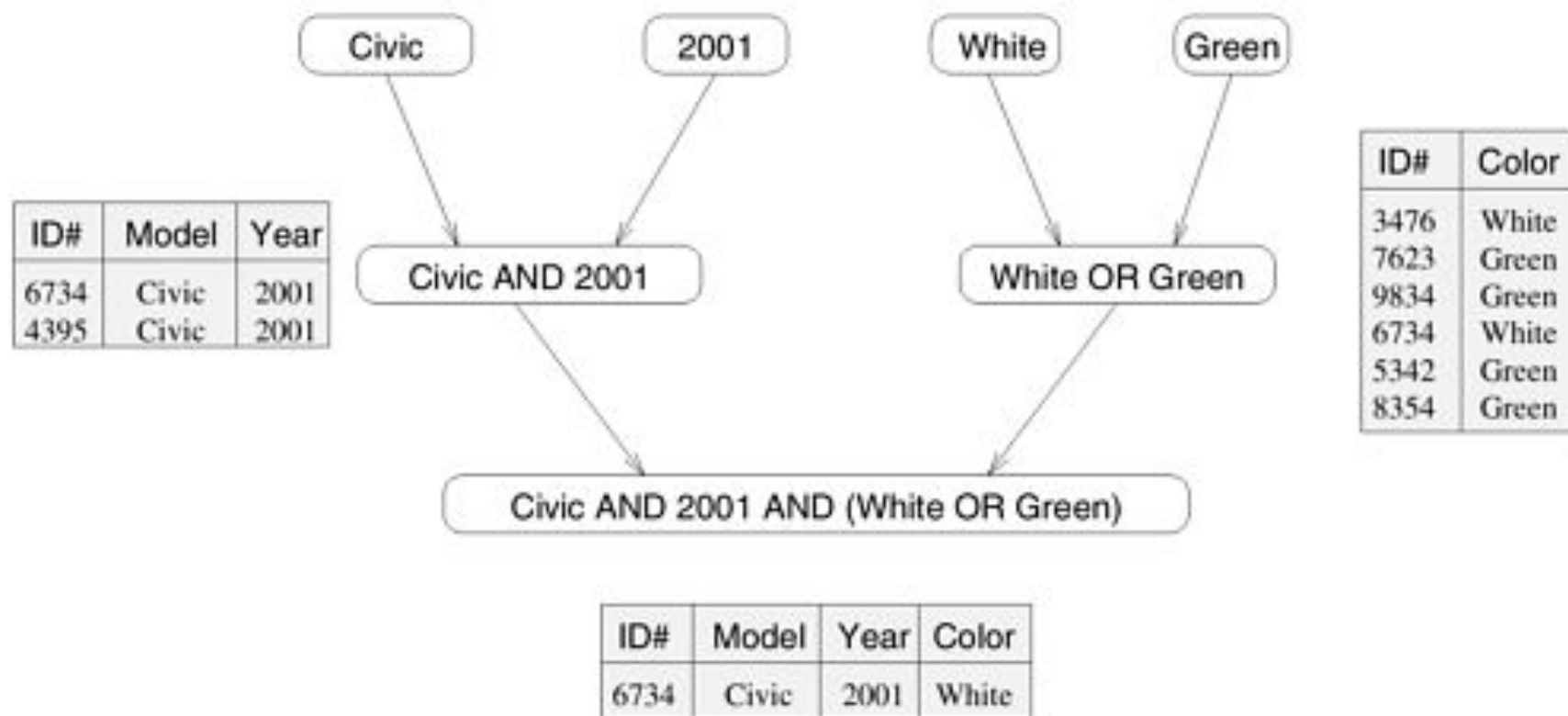


ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green



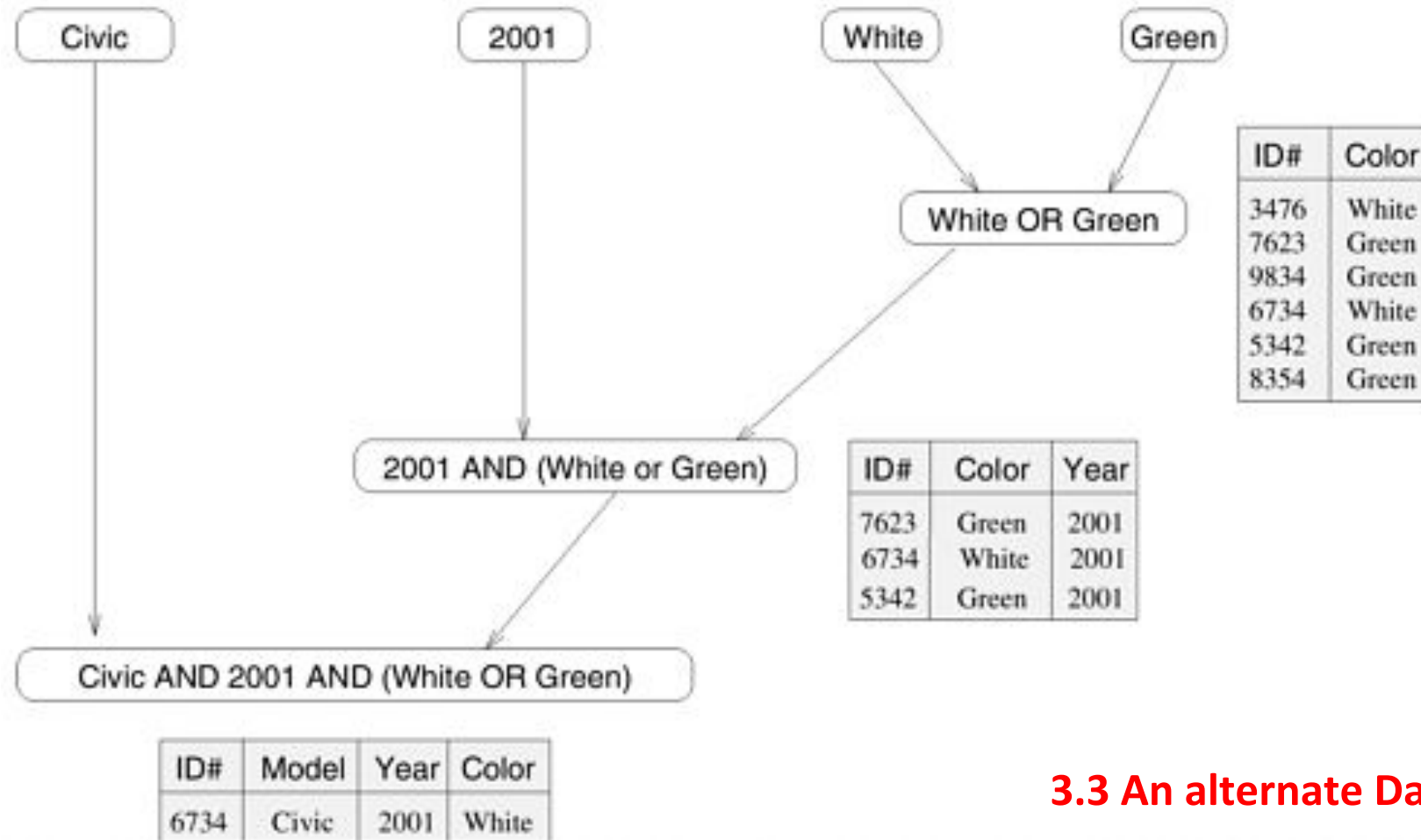
- 
- Often there are multiple ways of expressing certain computations, especially those involving associative operators such as addition, multiplication, and logical AND or OR.
  - Different ways of arranging computations can lead to different task-dependency graphs with different characteristics.

ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

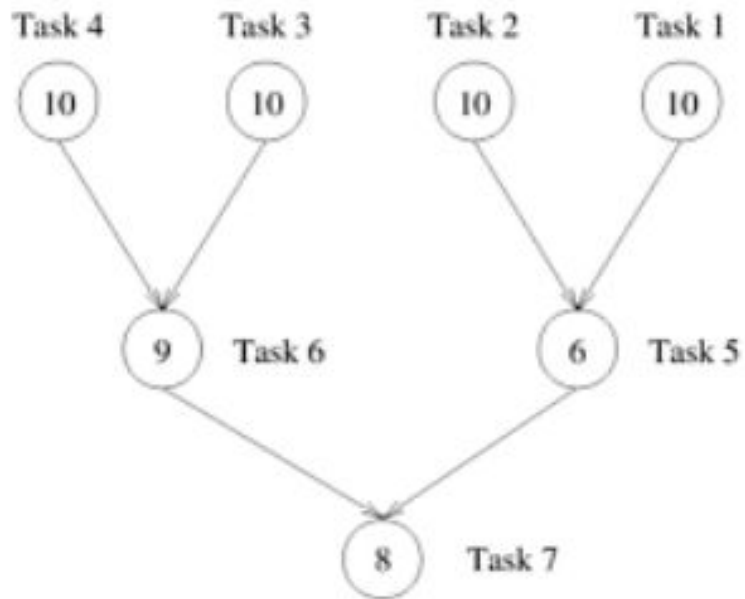
ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

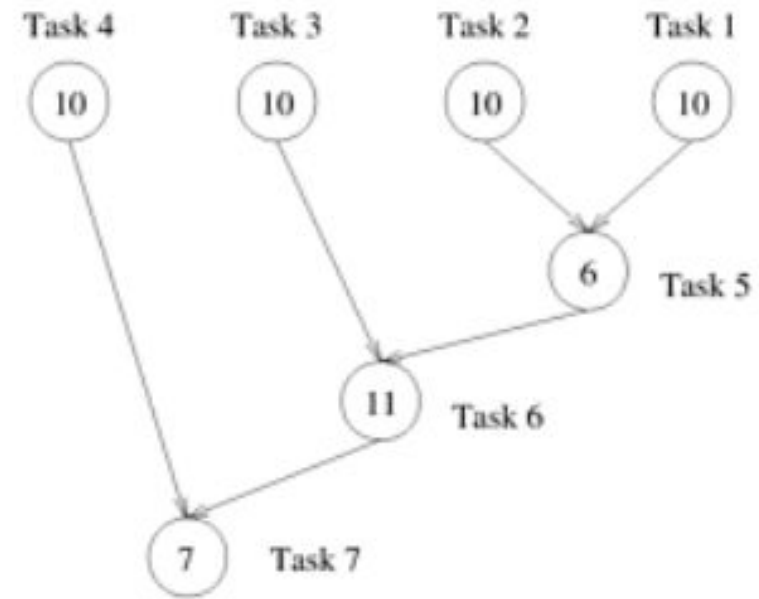


### 3.3 An alternate Data-Dependency Graph

**Figure 3.5** Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.



(a)

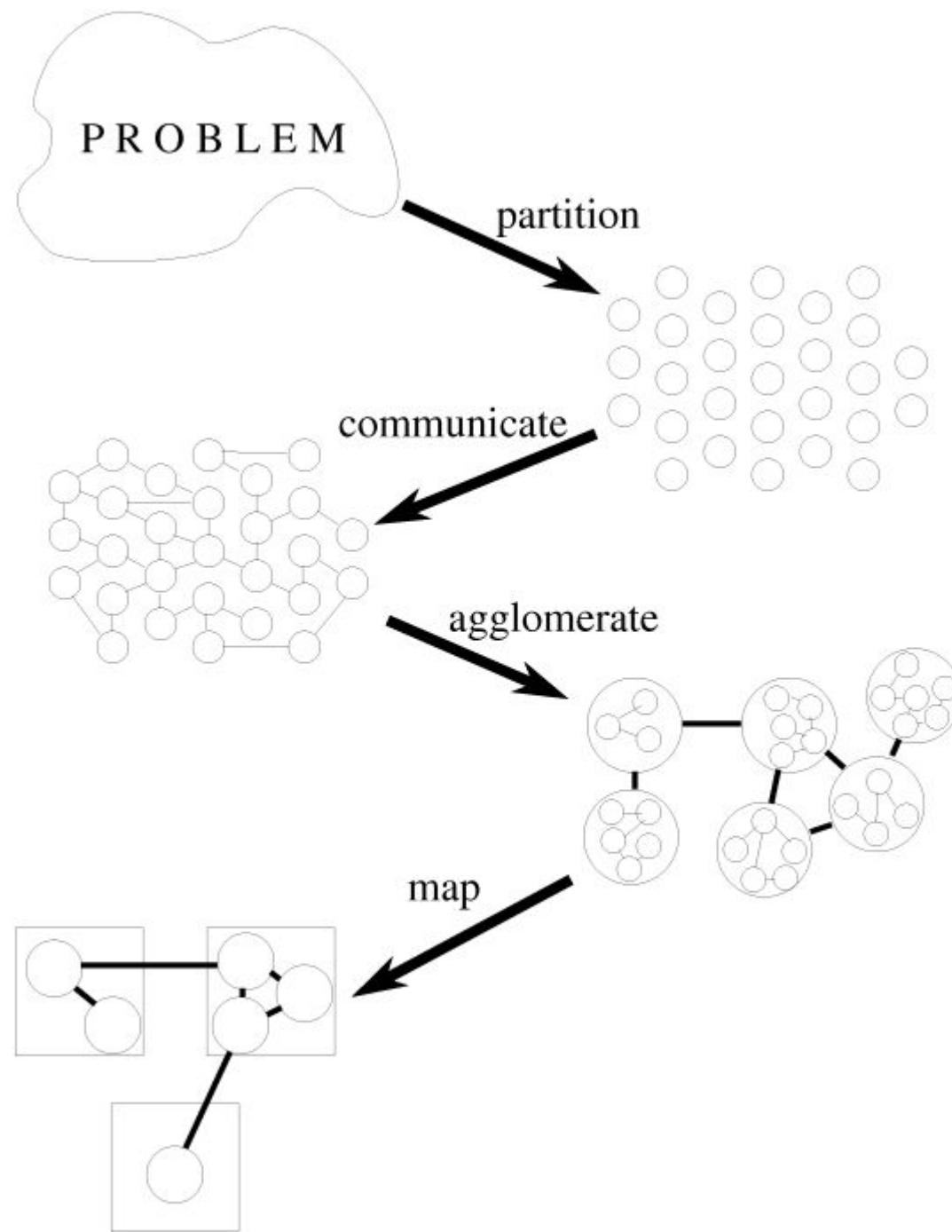


(b)

# Agglomeration

---

- **Agglomeration.** The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, **tasks are combined** into larger tasks to improve performance or to reduce development costs.
- Agglomeration is almost always beneficial if analysis of communication requirements reveals that a set of tasks cannot execute concurrently.



# Granularity, Concurrency, and Task-Interaction

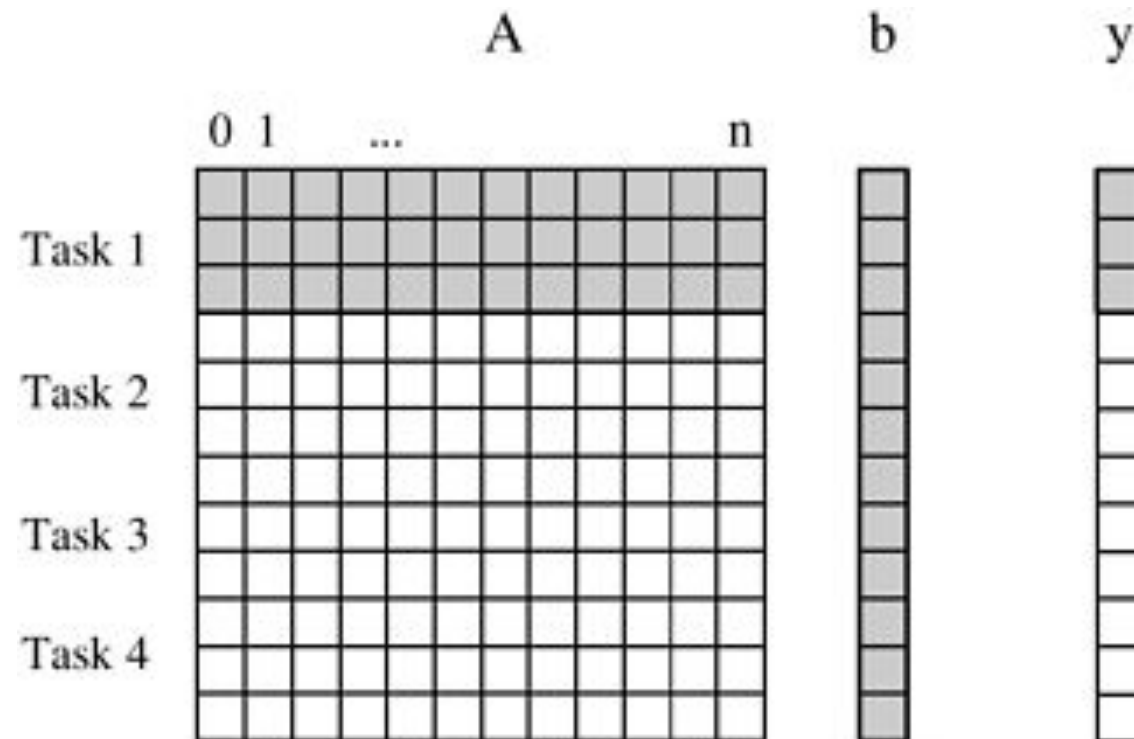
---

- The number and size of tasks into which a problem is decomposed determines the ***granularity*** of the decomposition.
  - A decomposition into a large number of small tasks is called ***fine-grained*** and a decomposition into a small number of large tasks is called ***coarse-grained***.

---

□ Matrix-vector multiplication example –

- **coarse-grain**: each task computes 3 elements of  $y[]$





---

□ **Degree of Concurrency:** # of tasks that can execute in parallel

- **maximum degree of concurrency:** largest # of concurrent tasks at any point of the execution
- **average degree of concurrency:** average # of tasks that can be executed concurrently

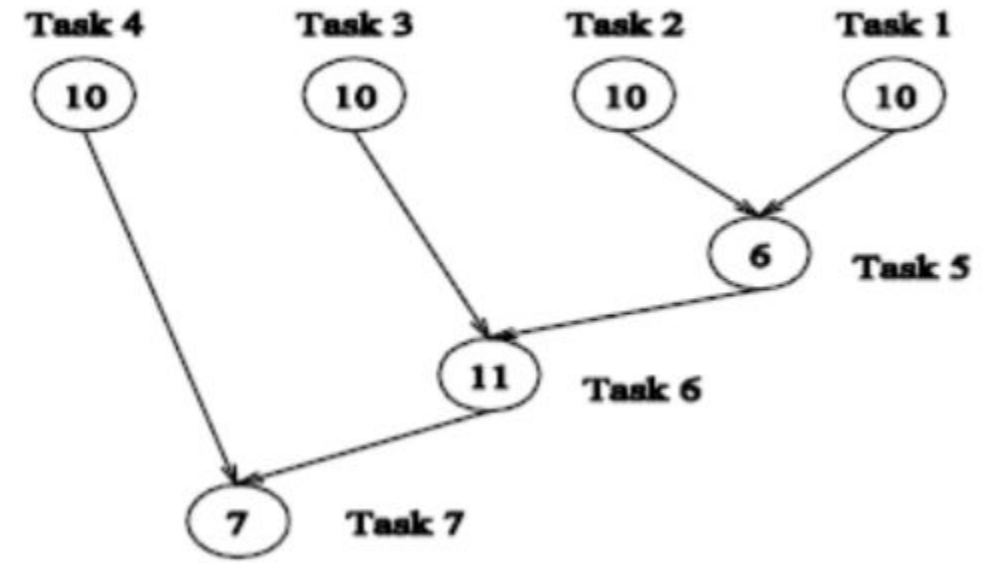
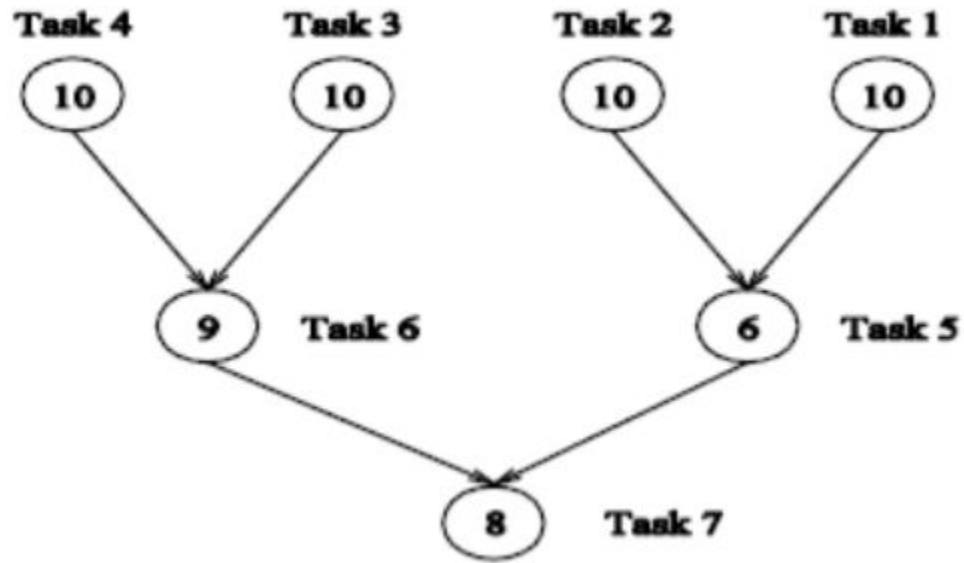
□ **Degree of Concurrency vs. Task Granularity** – Inverse relation

# Increasing Granularity

---

- A large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm.
- *Communication costs* and *task creation* costs are overhead that can be reduced by increasing granularity.

- 
- **Critical path:** The longest directed path between any pair of start node (node with no incoming edge) and finish node (node with no outgoing edges).
  - **Critical path length:** The sum of weights of nodes along critical path.
    - The weights of a node is the size or the amount of work associated with the corresponding task
  - **Average degree of concurrency** = total amount of work / critical path length



### □ Left graph:

- Critical path length = 27
- Average degree of concurrency =  $63/27 = 2.33$

### □ Right graph:

- Critical path length = 34
- Average degree of concurrency =  $64/34 = 1.88$

# Task Interaction Graph

---

- Tasks often **share** input, output, or intermediate data, which may lead to interactions not shown in task-dependency graph.
  - there may be interactions among tasks that appear to be independent in a task dependency graph
  - E.g. For the matrix-vector multiplication problem, all tasks are independent, and all need access to the entire input vector  $b$ .
- The pattern of interaction among tasks is captured by what is known as a ***task-interaction graph***.
  - The edges in a task interaction graph are usually undirected, but directed edges can be used to indicate the direction of flow of data, if it is unidirectional

## Example 3.3 Sparse matrix-vector multiplication

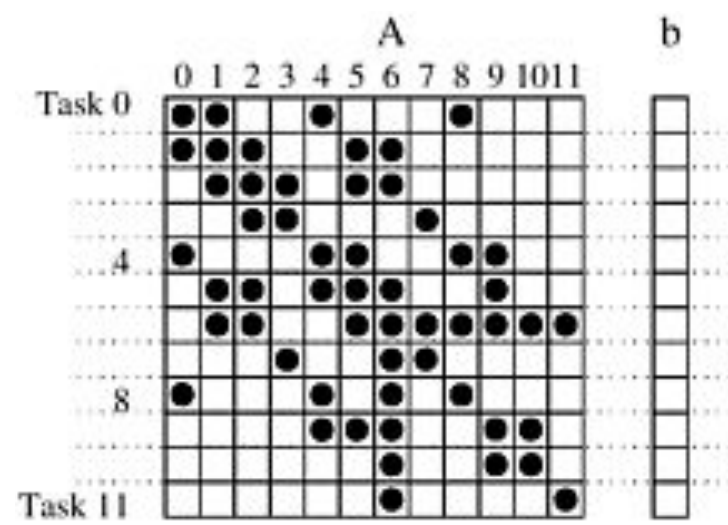
---

- Consider the problem of computing the product  $\mathbf{y} = \mathbf{A}\mathbf{b}$  of a sparse  $n \times n$  matrix  $\mathbf{A}$  with a dense  $n \times 1$  vector  $\mathbf{b}$

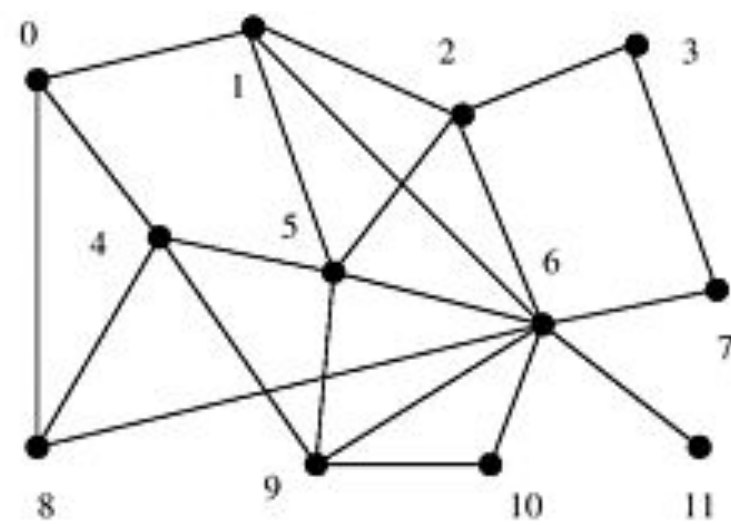
$$y[i] = \sum_{j=1}^n (A[i, j] \times b[j])$$

- We need to compute the products  $A[i, j] \times b[j]$  for only those values of  $j$  for which  $A[i, j] \neq 0$
- For example,  $y[0] = A[0, 0].b[0] + A[0, 1].b[1] + A[0, 4].b[4] + A[0, 8].b[8]$
- One possible way of decomposing this computation is to partition the output vector  $y$  and have each task compute an entry in it

- 
- **Task  $i$** , computes the element  $y[i]$  and also make it the "**owner**" of row  $A[i, *]$  of the matrix and the element  $b[i]$  of the input vector.
  
  - Note that the computation of  $y[i]$  requires access to many elements of  $b$  that are owned by other tasks. So **Task  $i$**  must get these elements from the appropriate locations.
  
  - In the message-passing paradigm, with the ownership of  $b[i]$ , **Task  $i$**  also inherits the responsibility of sending  $b[i]$  to all the other tasks that need it for their computation.
    - For example, Task 4 must send  $b[4]$  to Tasks 0, 5, 8, and 9 and must get  $b[0]$ ,  $b[5]$ ,  $b[8]$ , and  $b[9]$  to perform its own computation .



(a)



(b)

$$\sum_{0 \leq j \leq 11, A[i,j] \neq 0} A[i,j] \cdot b[j]$$



# Processes and Mapping

---

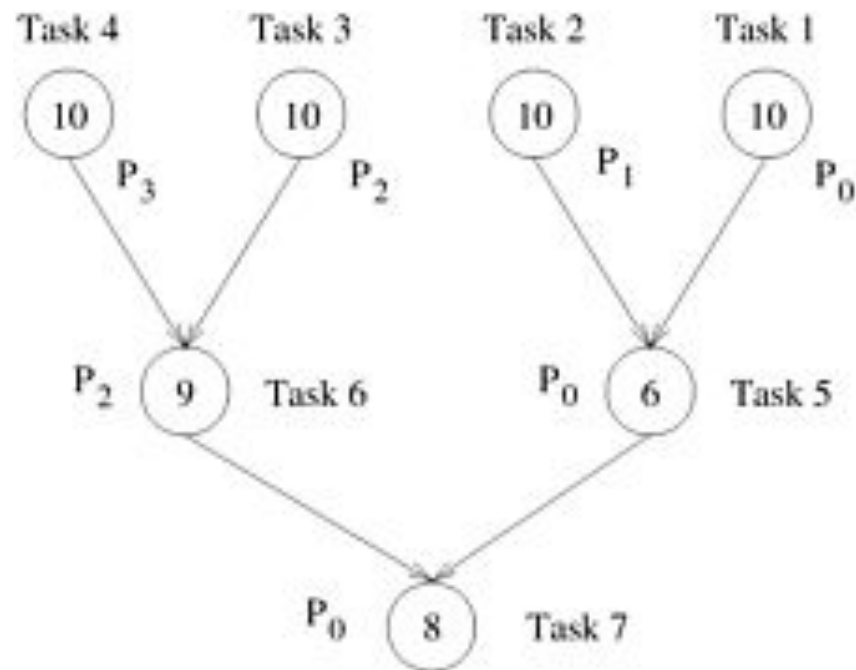
- **Process:** is a program, its local memory, and its communication *inports* and *outports*
- **Mapping.** Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs.
  - In general, the number of tasks in a decomposition exceeds the number of processing elements available.
  - For this reason, a parallel algorithm must also provide a mapping of tasks to processes.

# Processes and Mapping

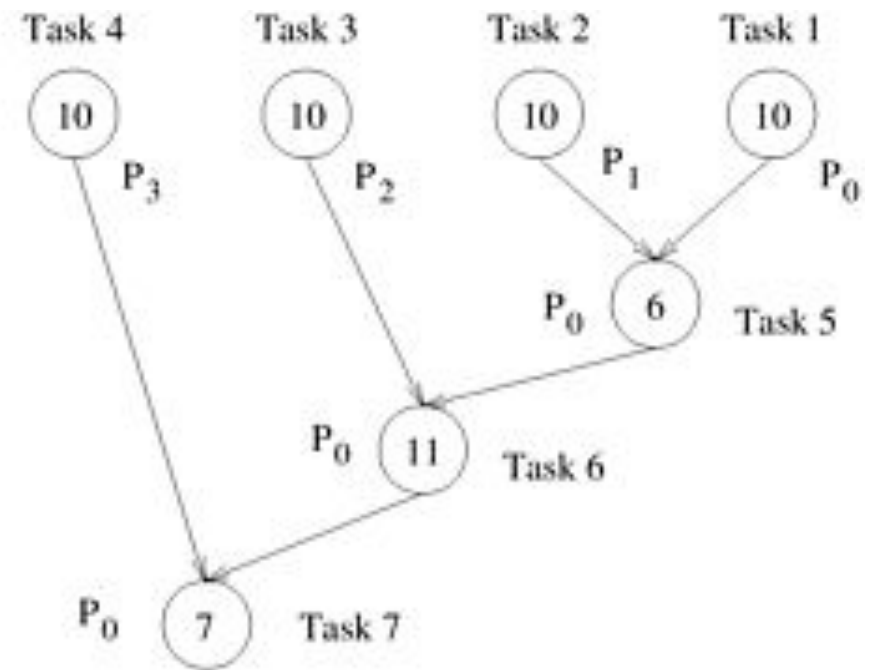
---

- **Note:** We refer to the mapping as being from tasks to processes, as opposed to processors. This is because typical programming APIs, as we shall see, do not allow easy binding of tasks to physical processors. Rather, we aggregate tasks into processes and rely on the system to map these processes to physical processors. We use processes, not in the UNIX sense of a process, rather, simply as a collection of tasks and associated data.

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



(a)



(b)

# Processes versus Processors

---

- **Processes** are logical computing agents that perform tasks.
- **Processors** are the hardware units that physically perform computations
  - In most cases, when we refer to processes in the context of a parallel algorithm, there is a one-to-one correspondence between processes and processors and it is appropriate to assume that there are as many processes as the number of physical CPUs on the parallel computer

# Decomposition Techniques

---

- 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
- **Recursive Decomposition**
- **Data Decomposition**
- **Exploratory Decomposition**
- **Speculative Decomposition**

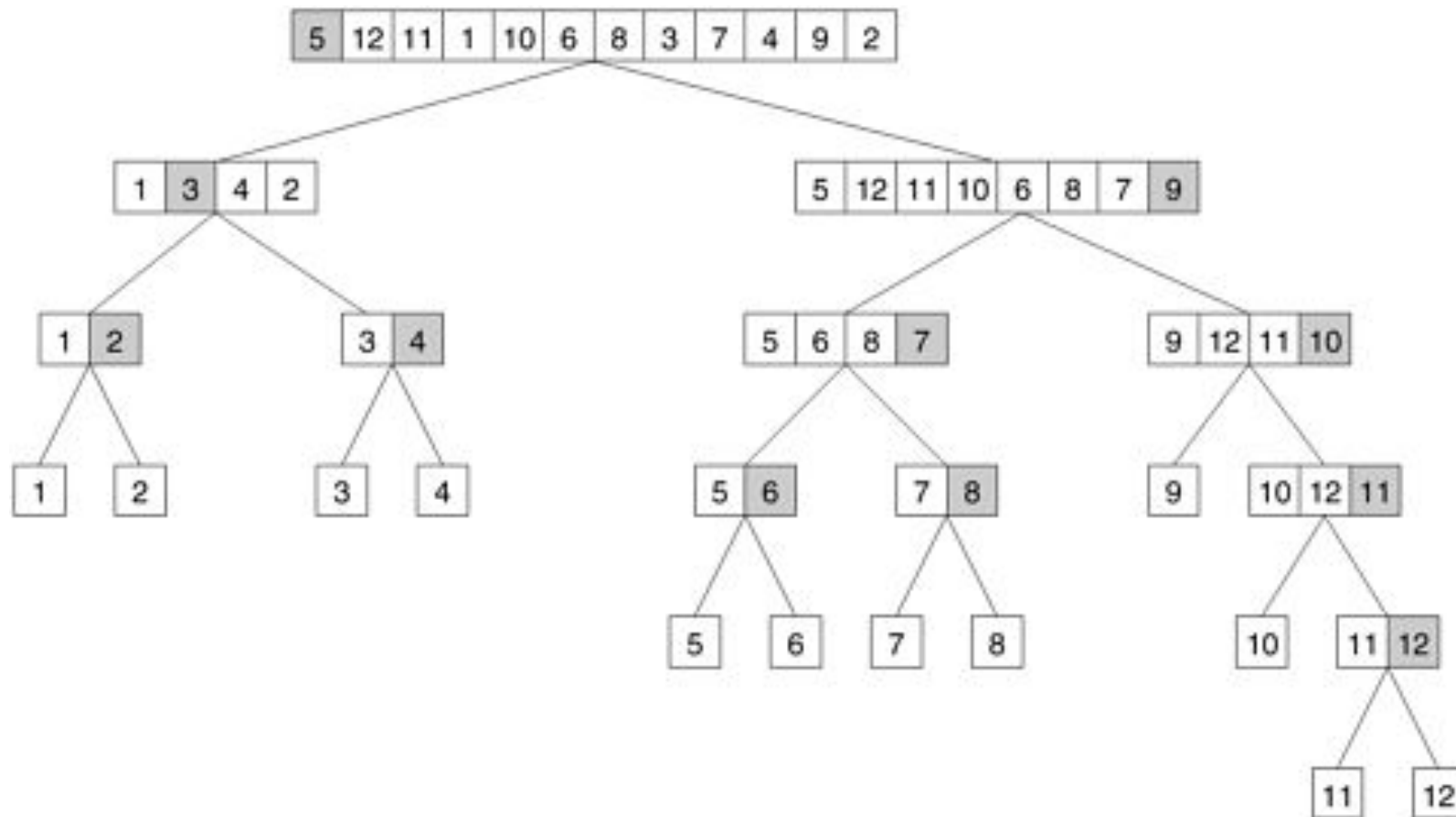
# Recursive Decomposition

---

- A method for inducing concurrency in problems that can be solved using the **divide-and-conquer** strategy.
- A problem is solved by first dividing it into a set of independent sub-problems

# Example: ...

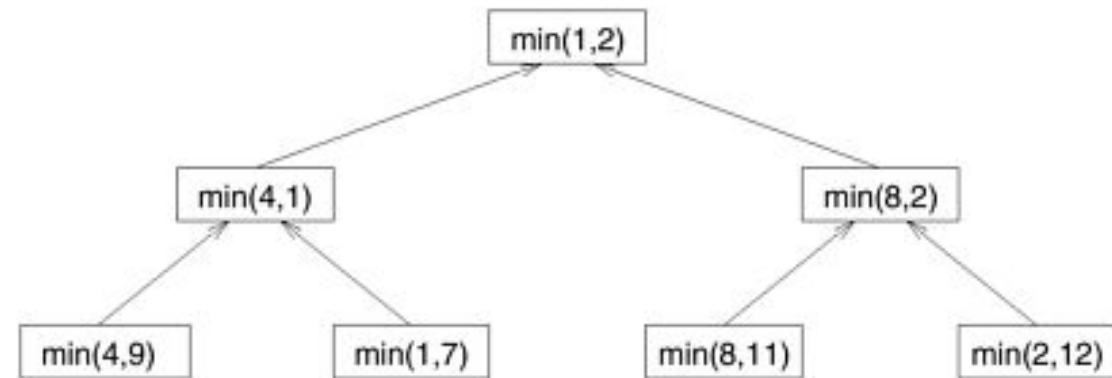
---



---

□ Sometimes, it is possible to restructure a computation to make it amenable to recursive decomposition even if the commonly used algorithm for the problem is not based on the divide-and-conquer strategy. For example:

```
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
```





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

---

```
1. procedure RECURSIVE_MIN ( $A$ ,  $n$ )
2. begin
3. if ( $n = 1$ ) then
4.      $min := A[0]$ ;
5. else
6.      $lmin := \text{RECURSIVE\_MIN} (A, n/2)$ ;
7.      $rmin := \text{RECURSIVE\_MIN} (\&(A[n/2]), n - n/2)$ ;
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

---

- In this method, the decomposition of computations is done in two steps:
  - **Step I:** The data on which the computations are performed is partitioned, and
  - **Step II:** this data partitioning is used to induce a partitioning of the computations into tasks.
- The operations that these tasks perform on different data partitions are usually similar (e.g., matrix multiplication) or are chosen from a small set of operations.

---

## Partitioning Output Data

- In many computations, each element of the output can be computed independently of others as a function of the input.
- In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks, where each task is assigned the work of computing a **portion of the output**.

# Example: 2 X 2 Matrices Multiplication

---

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

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

Two examples of decomposition of matrix multiplication into eight tasks.

---

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

(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				

**Example: Computing frequencies of itemsets in a Transaction Database**

(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

---

## Partitioning Input Data

- Partitioning of output data can be performed only if **each output** can be naturally computed as a function of the input.
- In many algorithms, it is not possible or desirable to partition the output data. For example, while **finding the minimum, maximum**, or the **sum** of a set of numbers, the output is a single unknown value.
- In such cases, it is sometimes possible to **partition the input data**, and then use this partitioning to induce concurrency.

- 
- A task is created for each partition of the input data and this task performs as much computation as possible using these **local data**.
  - 
  - The solutions to tasks induced by input partitions may not directly solve the original problem. In such cases, a follow-up computation is needed to combine the results.
  - The problem of computing the frequency of a set of itemsets in a transaction database described in previous example can also be decomposed based on a partitioning of input data:



(a) Partitioning the transactions among the tasks

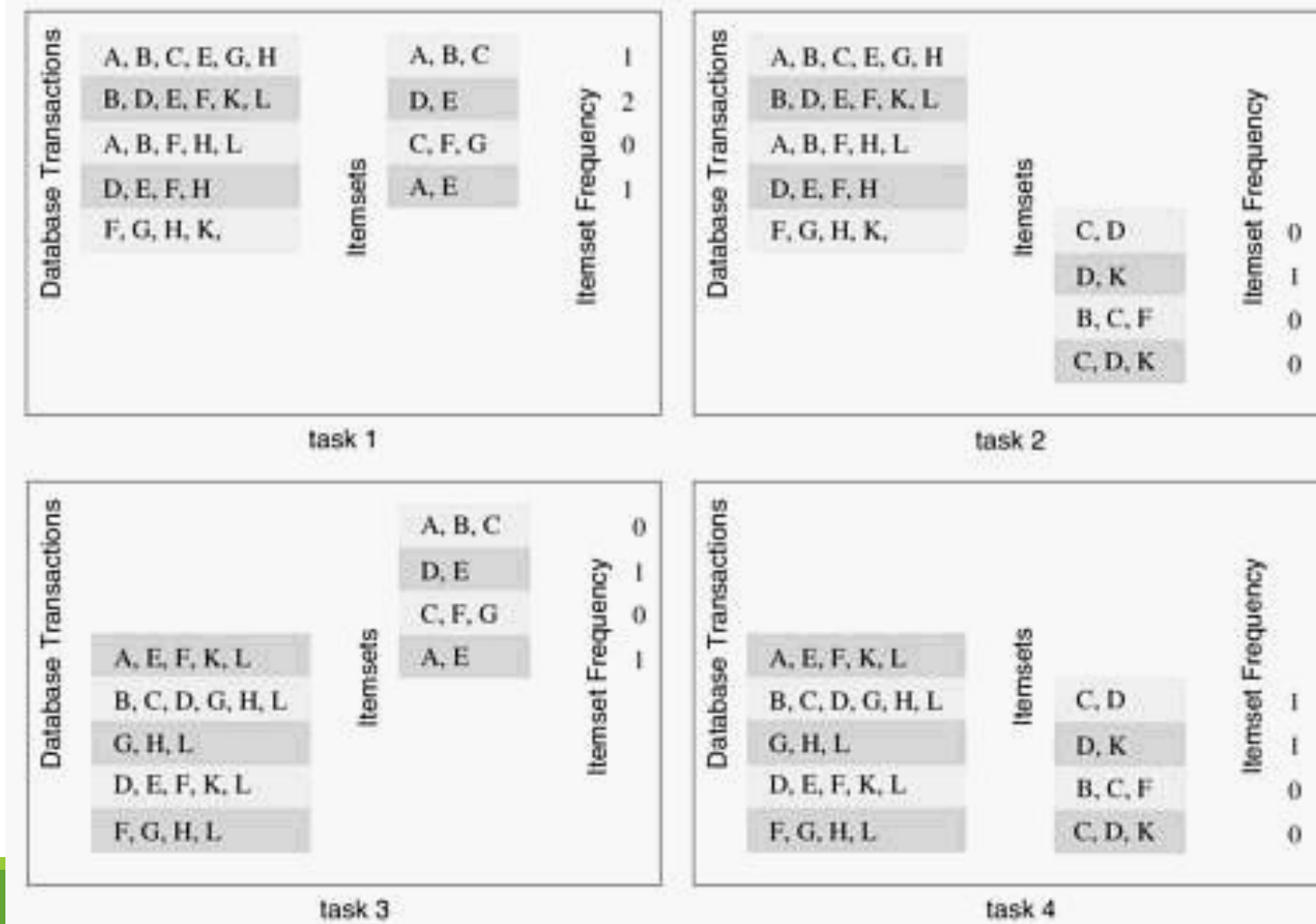
Database Transactions	A, B, C, E, G, H	Itemsets	A, B, C	Itemset Frequency	1
	B, D, E, F, K, L		D, E		2
	A, B, F, H, L		C, F, G		0
	D, E, F, H		A, E		1
	F, G, H, K,		C, D		0
			D, K		1
			B, C, F		0
			C, D, K		0

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 both Input and Output Data



# Partitioning Intermediate Data

---

- Algorithms are often structured as multi-stage computations such that the output of one stage is the input to the subsequent stage.
- Such an algorithm may be decomposed by partitioning the input or the output data of an intermediate stage of the algorithm.
- Often, the intermediate data are not generated explicitly in the serial algorithm for solving the problem and some restructuring of the original algorithm may be required to use intermediate data partitioning
- Partitioning intermediate data can sometimes lead to higher concurrency than partitioning input or output data

## 2 X 2 Matrices Multiplication of Output Data Partitioning

---

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

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

**Max. Degree of Concurrency = 4**

## Reconsidering.... ( with intermediate data decomposition)

---

Stage I

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

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:  $D_{1,1,1} = A_{1,1} B_{1,1}$

Task 02:  $D_{2,1,1} = A_{1,2} B_{2,1}$

Task 03:  $D_{1,1,2} = A_{1,1} B_{1,2}$

Task 04:  $D_{2,1,2} = A_{1,2} B_{2,2}$

Task 05:  $D_{1,2,1} = A_{2,1} B_{1,1}$

Task 06:  $D_{2,2,1} = A_{2,2} B_{2,1}$

Task 07:  $D_{1,2,2} = A_{2,1} B_{1,2}$

Task 08:  $D_{2,2,2} = A_{2,2} B_{2,2}$

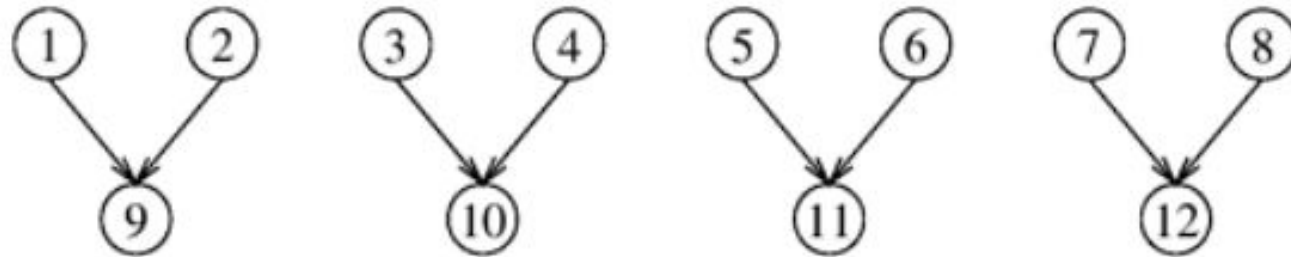
Task 09:  $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

Task 10:  $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 11:  $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Task 12:  $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

**Figure 3.16.** The task-dependency graph of the decomposition shown in [Figure 3.15](#).



# Exploratory Decomposition

---

- ***Exploratory decomposition*** is used to decompose problems whose underlying computations correspond to a **search of a space** for solutions.
- In exploratory decomposition, we partition the search space into smaller parts, and search each one of these parts concurrently, until the desired solutions are found.




# Example: the 15-Puzzle Problem

---

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

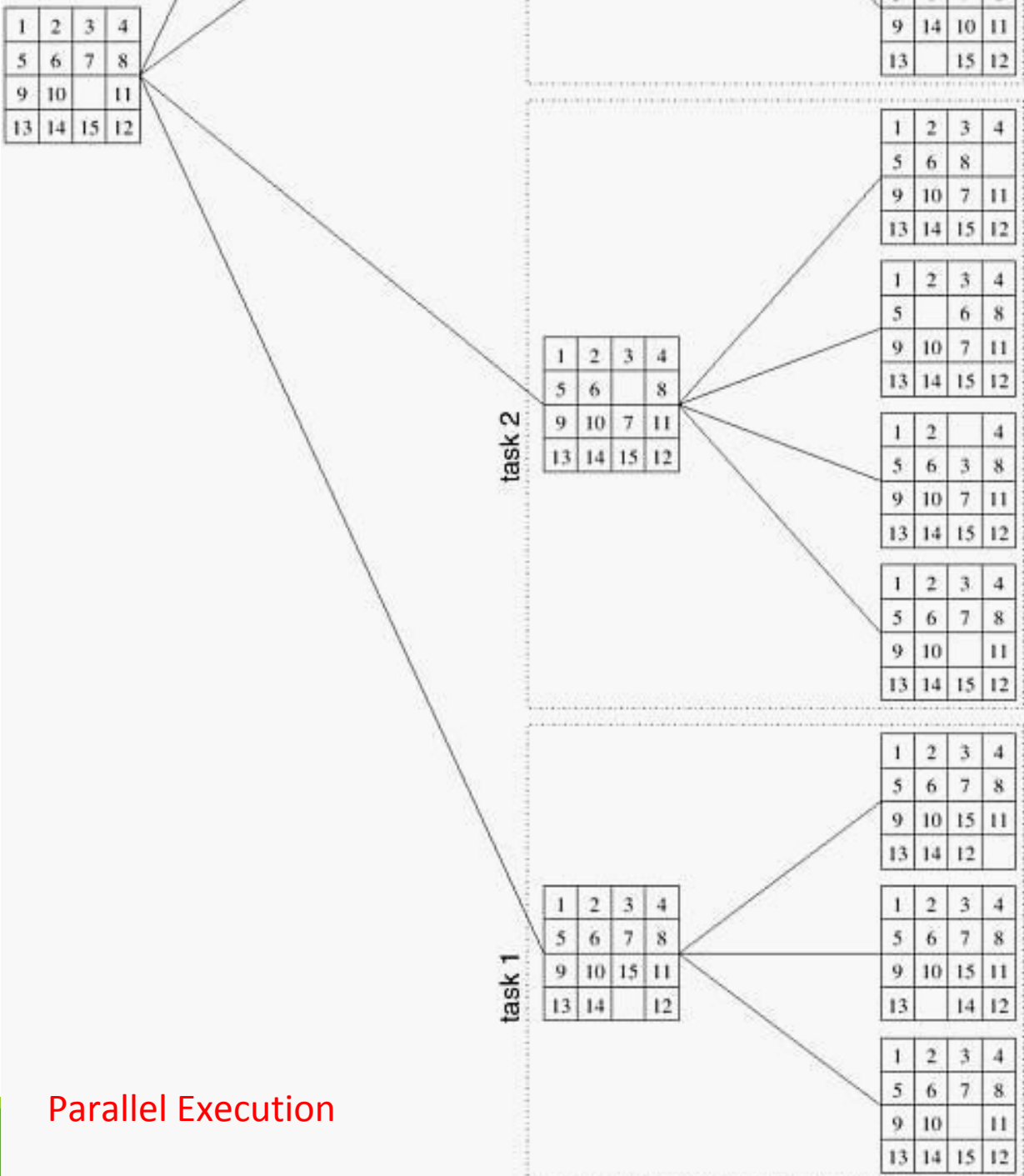
(b)

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

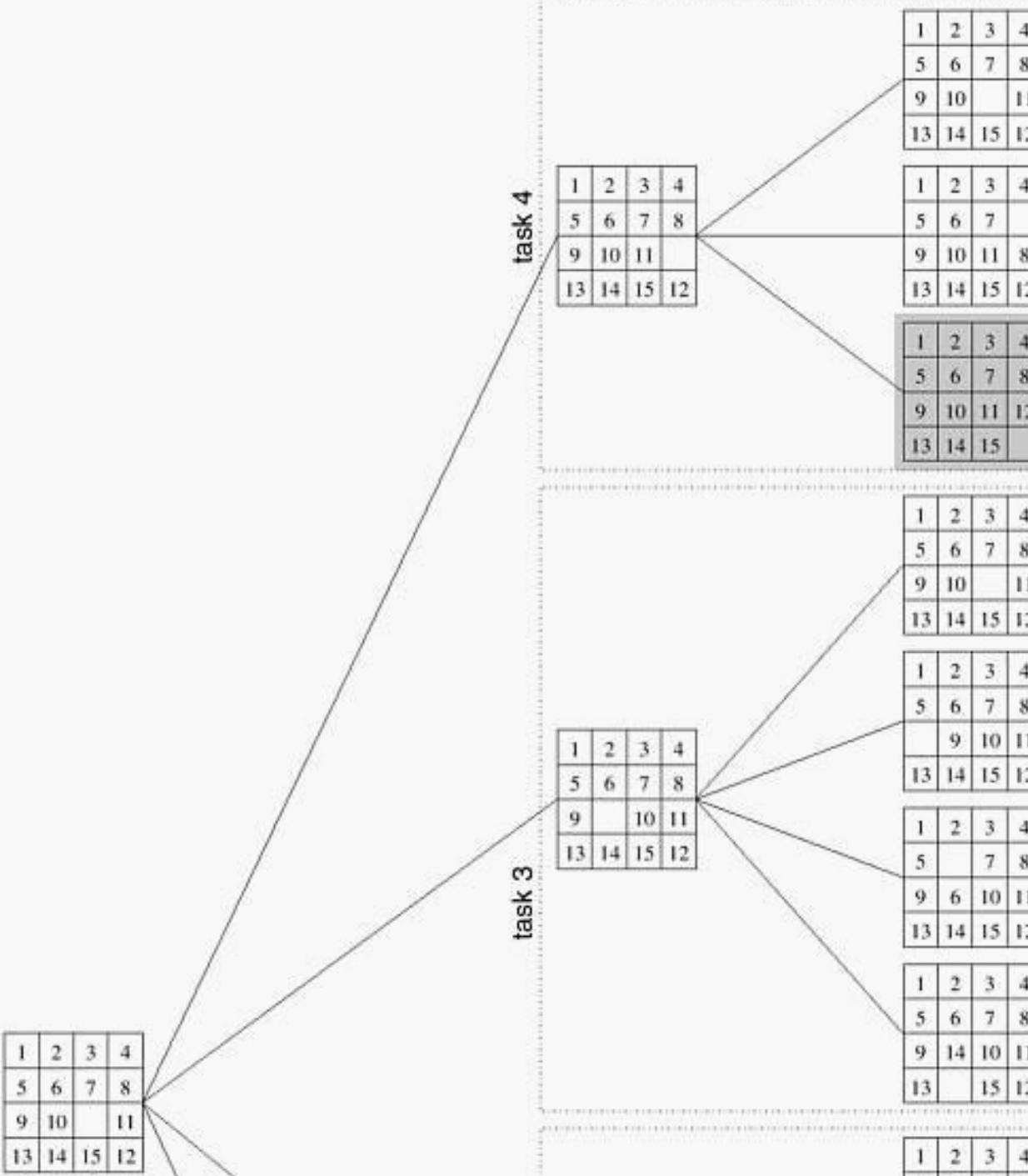
(c)

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

(d)



Parallel Execution



# Data Decomposition Vs Exploratory Decomposition

---

- The tasks induced by **data-decomposition** are performed in their entirety and each task performs useful computations towards the solution of the problem.
- On the other hand, in exploratory decomposition, unfinished tasks can be terminated as soon as an overall solution is found

- 
- The portion of the search space searched (and the aggregate amount of work performed) by a parallel formulation can be very different from that searched by a serial algorithm.
  - The work performed by the parallel formulation can be either smaller or greater than that performed by the serial algorithm.
    - Consider the different positions of solution.

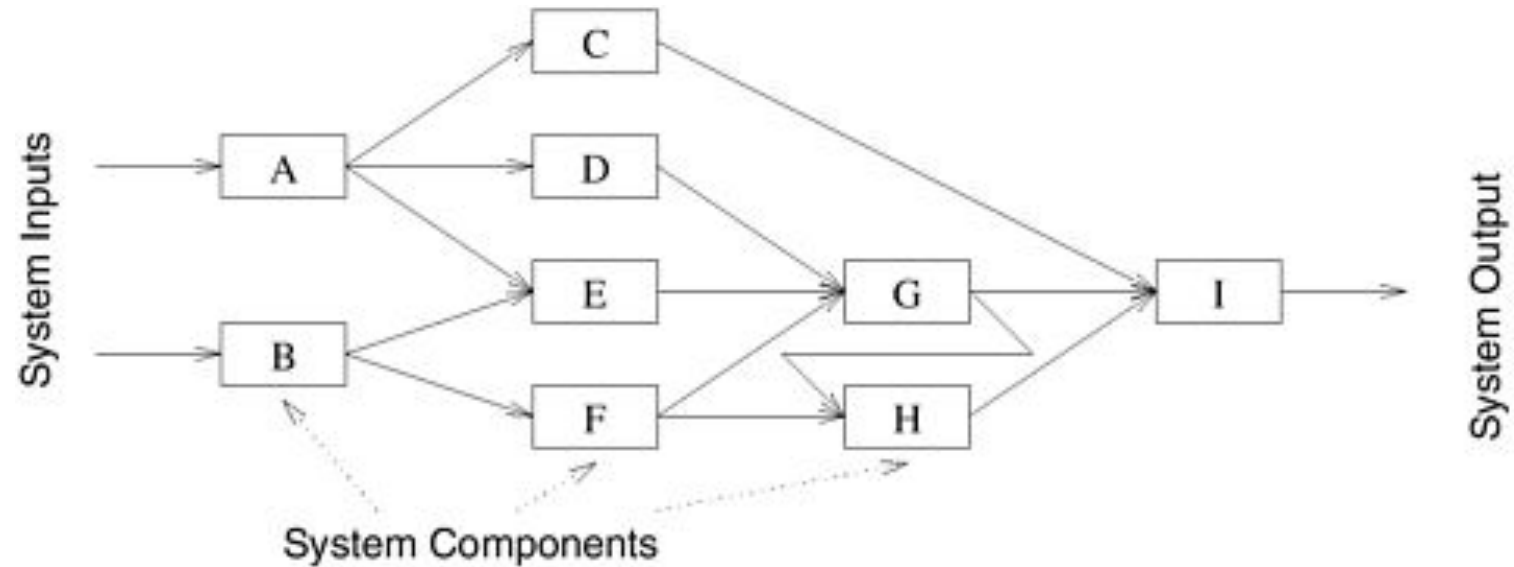
# Speculative Decomposition

---

- ***Speculative decomposition*** is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it.
- Scenario is similar to evaluating one or more of the branches of a *switch* statement in C in parallel before the input for the *switch* is available.
  - While one task is performing the computation that will eventually resolve the switch, other tasks could pick up the multiple branches of the switch in parallel.
  - When the input for the *switch* has finally been computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded.

## Example 3.8 Parallel discrete event simulation

---



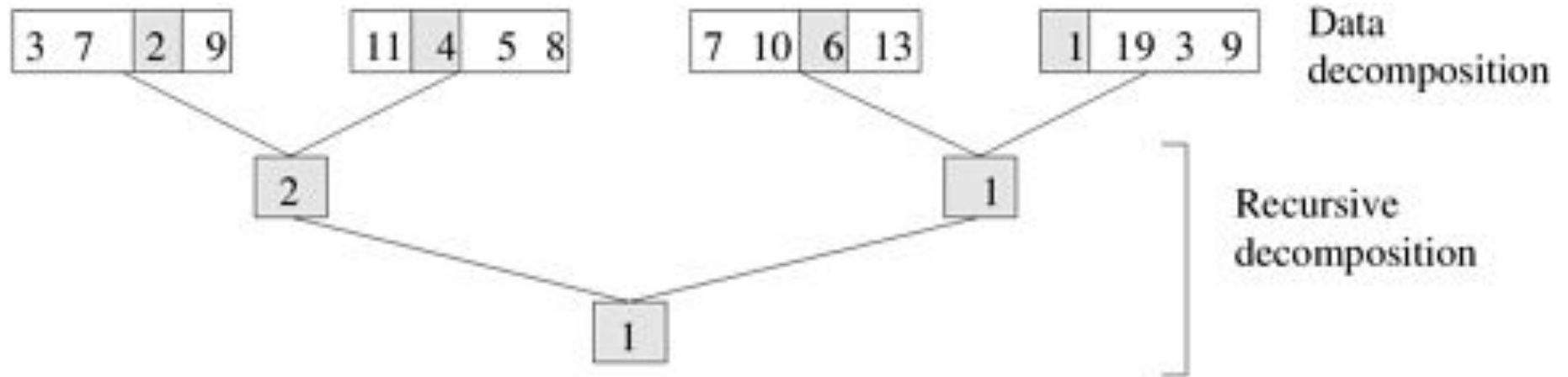
# Activity

---

□ Find the minimum of an array of size 16 using four tasks.

# Hybrid Decompositions

□ Example: finding the minimum of an array of size 16 using four tasks.





# PRACTICE TASK:

Propose a Parallel Solution to the following

minsup = 2

Database TDB

Tid	Items
10	A, C, D
20	B, C, E
30	A, B, C, E
40	B, E

1<sup>st</sup> scan

$C_1$

Itemset	sup
{A}	2
{B}	3
{C}	3
{D}	1
{E}	3

$L_1$

Itemset	sup
{A}	2
{B}	3
{C}	3
{E}	3

2<sup>nd</sup> scan

$C_2$

Itemset	sup
{A, B}	1
{A, C}	2
{A, E}	1
{B, C}	2
{B, E}	3
{C, E}	2

$C_2$

Itemset
{A, B}
{A, C}
{A, E}
{B, C}
{B, E}
{C, E}

$L_2$

Itemset	sup
{A, C}	2
{B, C}	2
{B, E}	3
{C, E}	2

$C_3$

Itemset
{B, C, E}

3<sup>rd</sup> scan

$L_3$

Itemset	sup
{B, C, E}	2

# Mapping Techniques for Load Balancing

---

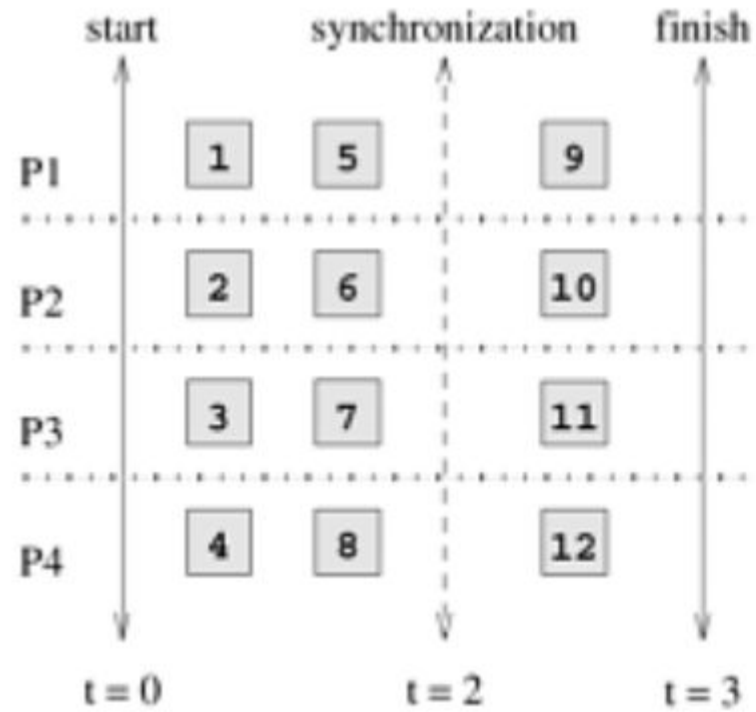
- Once a computation has been decomposed into tasks, these tasks are **mapped** onto processes with the objective that all tasks complete in the shortest amount of elapsed time.
  
- In order to achieve a small execution time, the ***overheads*** of executing the tasks in parallel must be minimized. The two key sources of overhead are:
  1. The time spent in inter-process interaction.
  2. The time that some processes may spend being **idle**.

- 
- Uneven load distribution may cause some processes to finish earlier than others.
  - At times, all the unfinished tasks mapped onto a process may be waiting for tasks mapped onto other processes to finish in order to satisfy the constraints imposed by the task-dependency graph.
  - Both interaction and idling are often a function of mapping, and it requires to achieve:
    - Reduce the amount of time processes spend in interacting with each other.
    - Reduce the total amount of time some processes are idle while the others are engaged in performing some tasks.

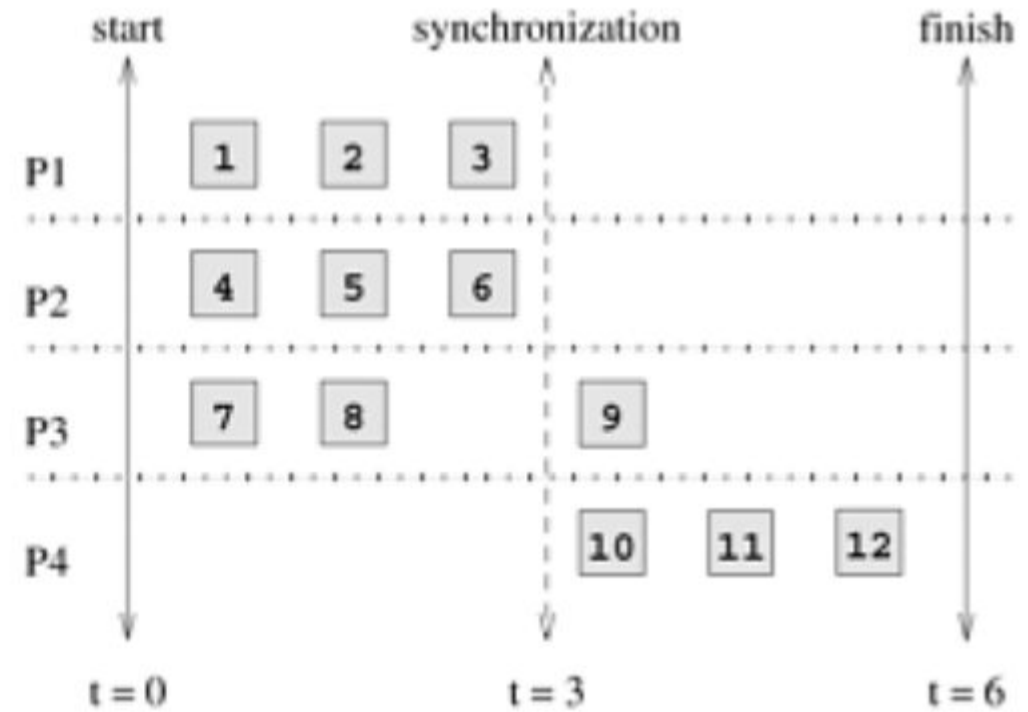
---

□ A good mapping must ensure that the computations and interactions among processes at each stage of the execution of the parallel algorithm are well balanced

Figure 3.23



(a)



(b)

# Static and Dynamic Mappings

---

- **Static Mapping** techniques distribute the tasks among processes prior to the execution of the algorithm. **Dynamic Mapping** techniques distribute the work among processes during the execution of the algorithm.
- For statically generated tasks, either static or dynamic mapping can be used. The choice of a good mapping in this case depends on several factors, including the **knowledge of task sizes**, the **size of data** associated with tasks, the characteristics of **inter-task interactions**, and even the parallel programming paradigm.

---

□ If tasks are generated dynamically, then they must be mapped dynamically too.

□ If task sizes are unknown, then a static mapping can potentially lead to serious **load-imbalances** and dynamic mappings are usually more effective.

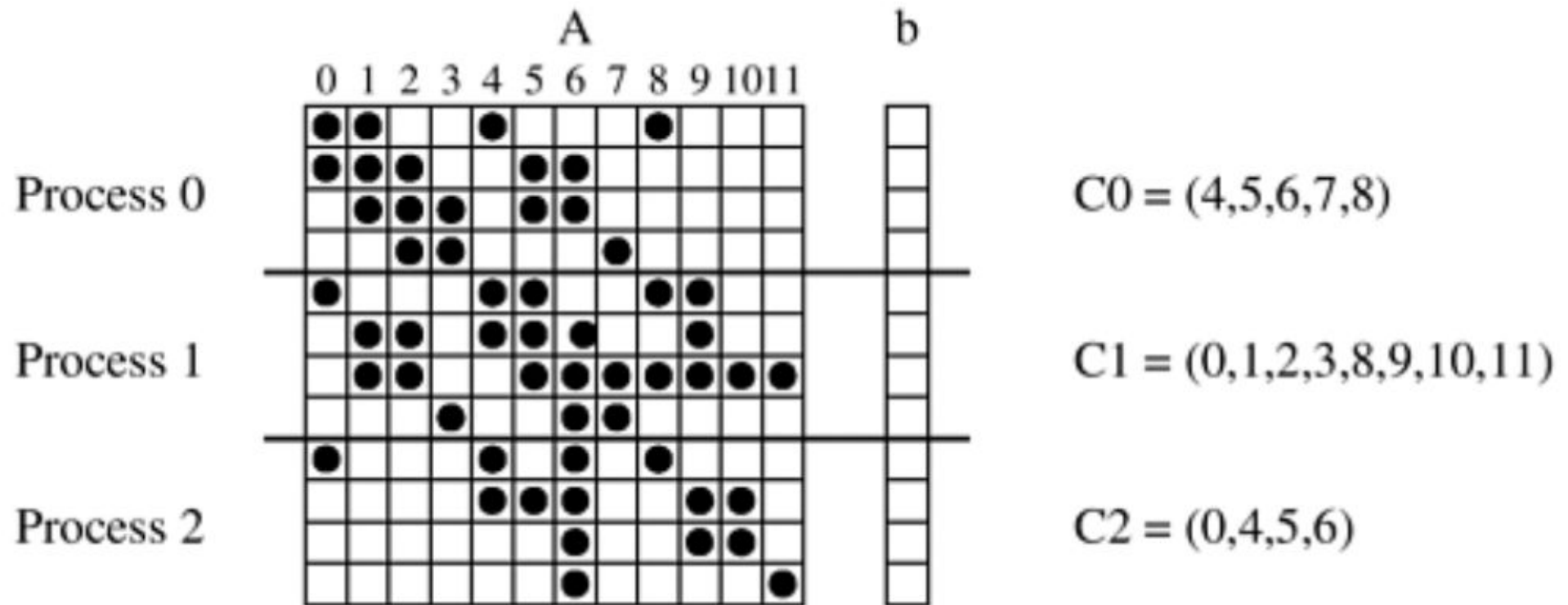


# Mappings Based on Task Partitioning

---

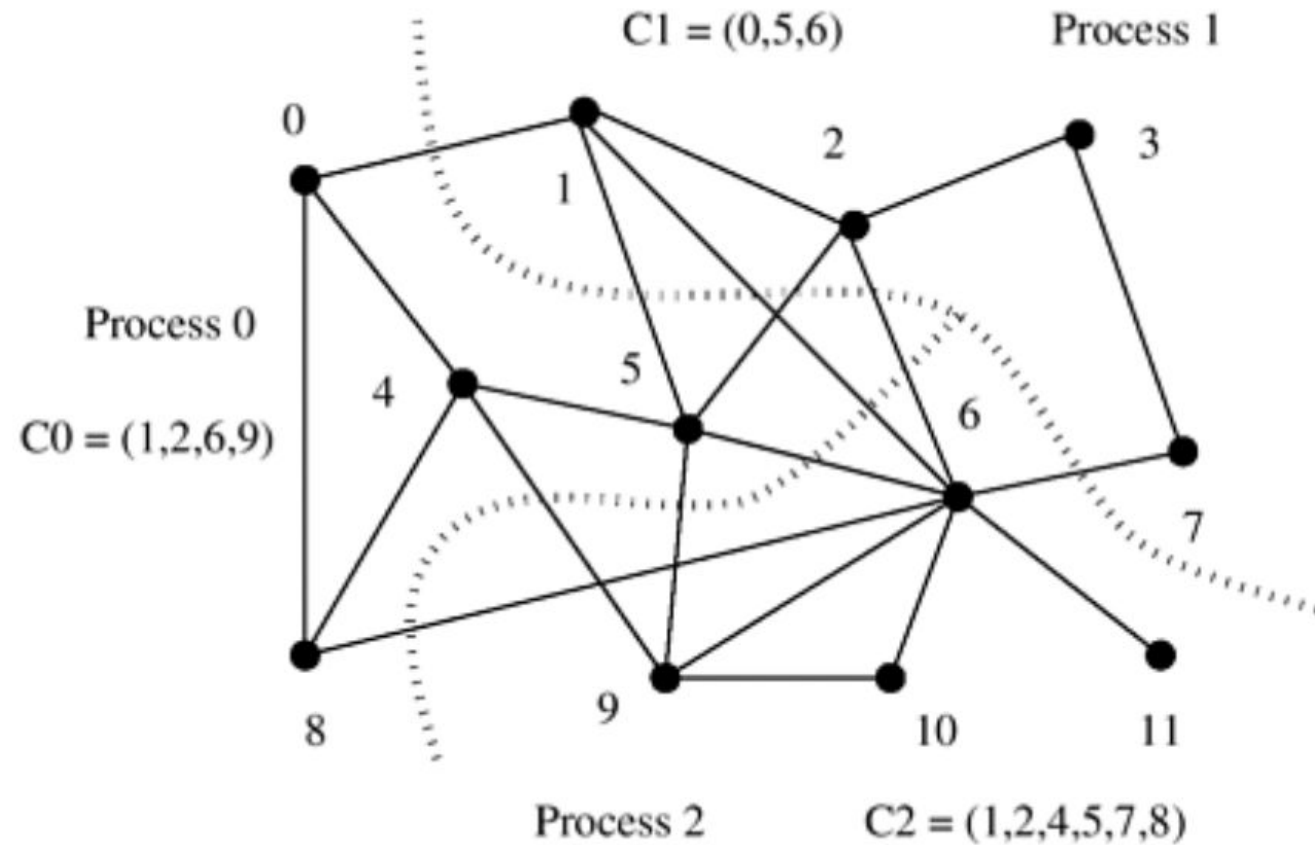
- 
- Figure shows a mapping of this task-dependency graph onto eight processes.
  - It is easy to see that this mapping minimizes the interaction overhead by mapping many interdependent tasks onto the same process (i.e., *the tasks along a straight branch of the tree*) and others on processes only one communication link away from each other.
  - The mapping shown in figure does not introduce any further idling and all tasks that are permitted to be concurrently active by the task-dependency graph are mapped onto different processes for parallel execution.

# Example: Sparse Matrix Multiplication



Reducing interaction overhead in sparse matrix-vector multiplication by partitioning the task-interaction graph.

---



# Parallel Algorithm Models: **The Data-Parallel Model**

---

- In this model, the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data
- This type of parallelism that is a result of identical operations being applied concurrently on different data items is called ***data parallelism***
  - The work may be done in phases and the data operated upon in different phases may be different
  - Typically, data-parallel computation phases are combined with interactions to synchronize the tasks or to get fresh data to the tasks.

---

□ Since all tasks perform similar computations, the decomposition of the problem into tasks is usually based on **data partitioning** because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance

# Parallel Algorithm Models: Task Graph Model

---

- In the *task graph model*, the **interrelationships** among the tasks are utilized to promote locality or to reduce interaction costs.
- This model is typically employed to solve problems in which the *amount of data associated with the tasks is large relative to the amount of computation* associated with them
- Usually, tasks are mapped statically to help optimize the cost of data movement among tasks.
- Examples: parallel quicksort, sparse matrix factorization, and many parallel algorithms derived via divide-and-conquer decomposition
- This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called ***task parallelism***

# Parallel Algorithm Models: The Work Pool Model

---

- The *work pool* or the *task pool* model is characterized by a dynamic mapping of tasks onto processes for load balancing in which **any task may potentially be performed by any process.**
- The work may be statically available in the beginning, or could be dynamically generated; i.e., the processes may generate work and add it to the global (possibly distributed) work pool
- In the message-passing paradigm, the work pool model is typically used when the amount of data associated with tasks is relatively small compared to the computation associated with the tasks
  - As a result, tasks can be readily moved around without causing too much data interaction overhead



# Parallel Algorithm Models: The Work Pool Model

---

□ E.g.

- Parallelization of loops by chunk scheduling is an example of the use of the work pool model with centralized mapping when the tasks are statically available.
- Parallel tree search where the work is represented by a centralized or distributed data structure is an example of the use of the work pool model where the tasks are generated dynamically

# Parallel Algorithm Models: The Master-Slave Model

---

- In the *master-slave* or the *manager-worker* model, one or more master processes generate work and allocate it to worker processes
  - The tasks may be allocated *a priori* if the manager can estimate the size of the tasks or if a random mapping can do an adequate job of load balancing.
  - In another scenario, workers are assigned smaller pieces of work at different times. This scheme is preferred if it is time consuming for the master to generate work and hence it is not desirable to make all workers wait until the master has generated all work pieces.

- 
- In some cases, work may need to be performed in phases, and work in each phase must finish before work in the next phases can be generated
    - In this case, the manager may cause all workers to synchronize after each phase.

# Parallel Algorithm Models: The Pipeline or Producer-Consumer Model

---

- In the *pipeline model*, a stream of data is passed on through a succession of processes, each of which perform some task on it.
  - This simultaneous execution of different programs on a data stream is called ***stream parallelism***
- With the exception of the process initiating the pipeline, the arrival of new data triggers the execution of a new task by a process in the pipeline.
- The processes could form such pipelines in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles

# Parallel Algorithm Models: The Pipeline or Producer-Consumer Model

---

- A pipeline is a chain of producers and consumers.
  - Each process in the pipeline can be viewed as a consumer of a sequence of data items for the process preceding it in the pipeline and as a producer of data for the process following it in the pipeline.
- The pipeline model usually involves a static mapping of tasks onto processes
- Load balancing is a function of task granularity. The larger the granularity, the longer it takes to fill up the pipeline

# Parallel Algorithm Models: **Hybrid Models**

---

- In some cases, more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model.
- A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.