Parallel Programming Examples

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.

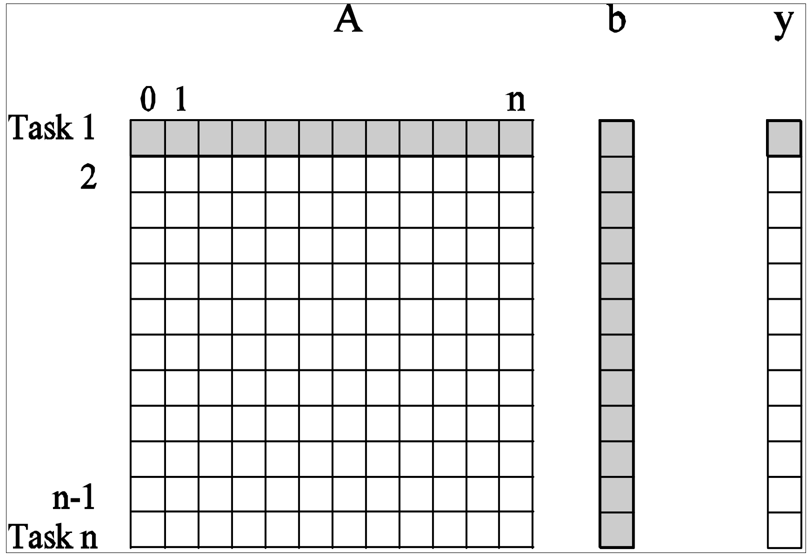
Concurrency and Mapping

- **Mapping Techniques for Load Balancing**
  - Static and Dynamic Mapping
- **Methods for Minimizing Interaction Overheads**
  - Maximizing Data Locality
  - Minimizing Contention and Hot-Spots
  - Overlapping Communication and Computations
  - Replication vs. Communication
  - Group Communications vs. Point-to-Point Communication
- **Parallel Algorithm Design Models**
  - Data-Parallel, Work-Pool, Task Graph, Master-Slave, Pipeline, and Hybrid Models

Preliminaries: Decomposition, Tasks, and Dependency Graphs

- The first step in developing a parallel algorithm is to decompose the problem
- A given problem may be docomposed into tasks in many different ways.
- Tasks may be of same, different, or even interminate sizes.
- A decomposition can be illustrated in the form of a directed 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 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?*

Example: Database Query Processing

Consider the execution of the query:

MODEL = ``CIVIC'' AND YEAR = 2001 AND

(COLOR = ``GREEN'' OR COLOR = ``WHITE)

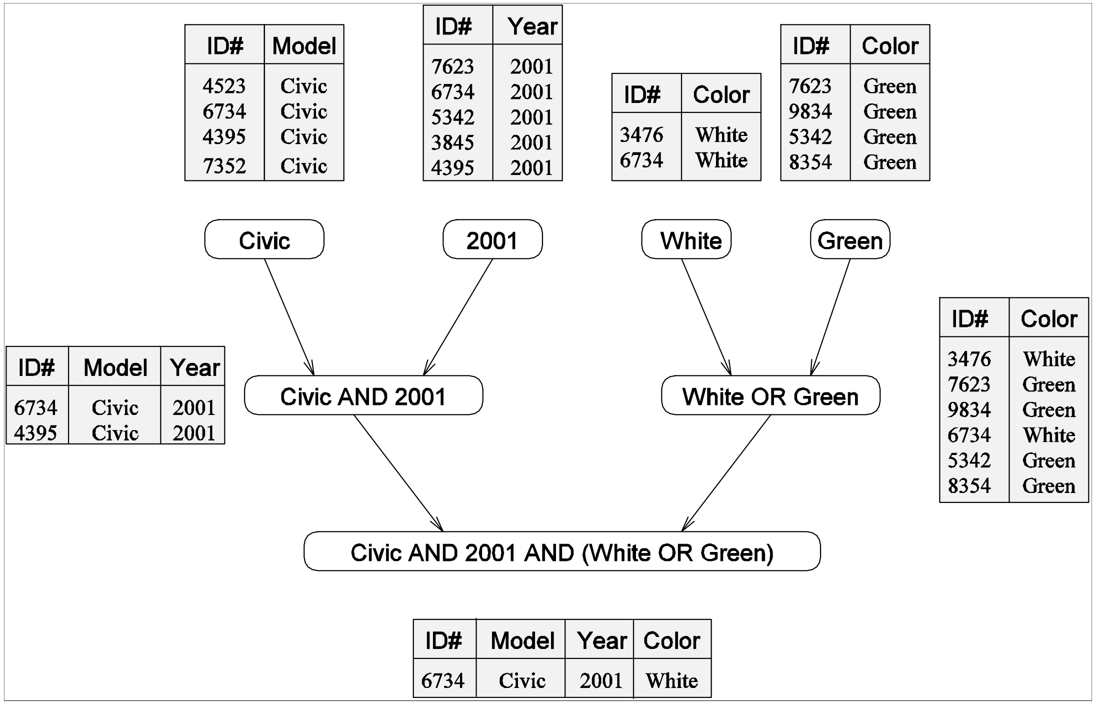
on the following database:

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

Example: Database Query Processing

The execution of the query can be divided into subtasks.

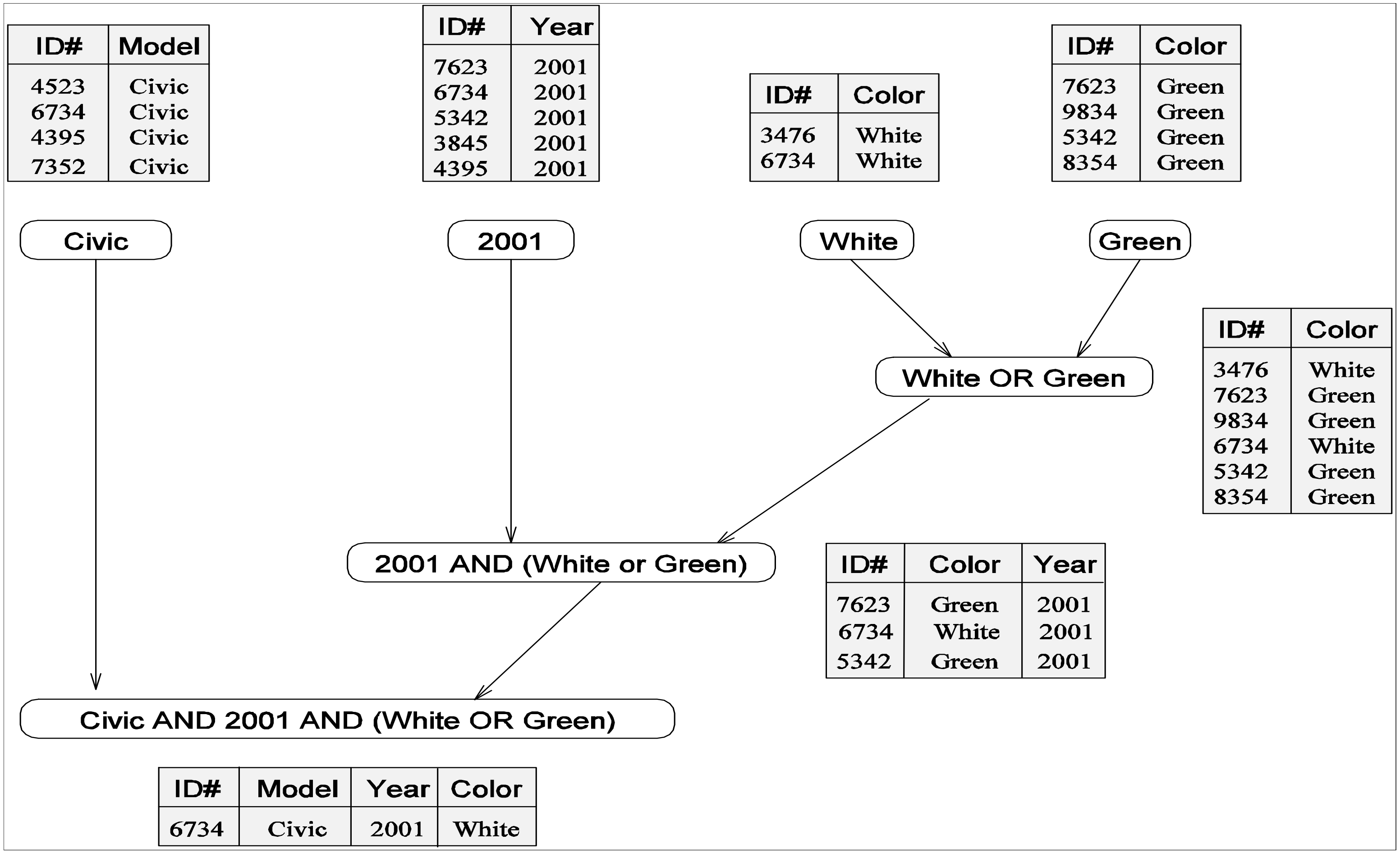
Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.



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.

Example: Database Query Processing

Note that the same problem can be decomposed in multiple ways



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

ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Color	Year
7623	Green	2001
6734	White	2001
5342	Green	2001

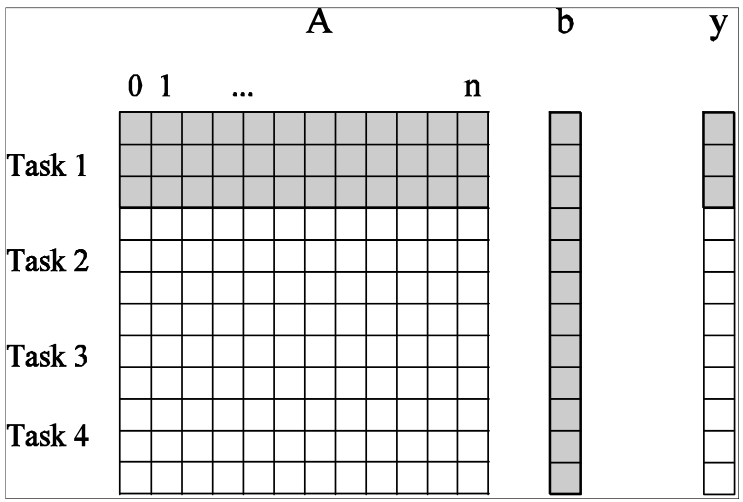
ID#	Model	Year	Color
6734	Civic	2001	White

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**



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.

- 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.

- *Assuming that each tasks in the database example takes identical processing time, what is the average degree of concurrency in each decomposition?*

- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

Critical Path Length

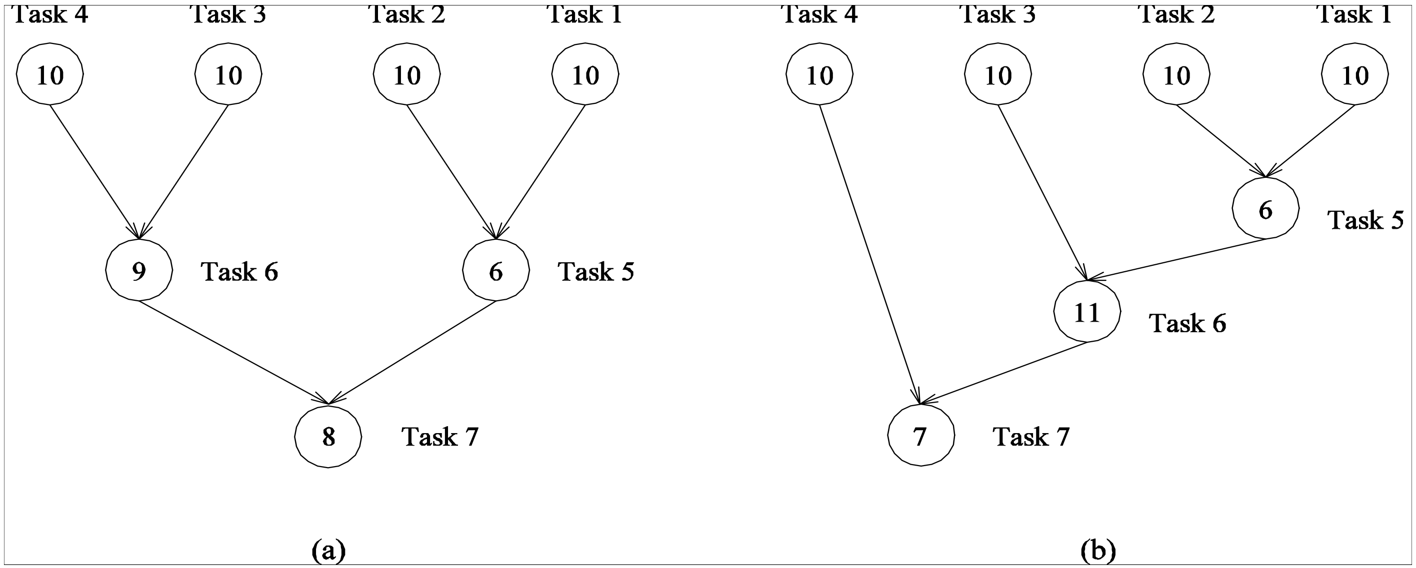
- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.

- 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 the two database query 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?

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.

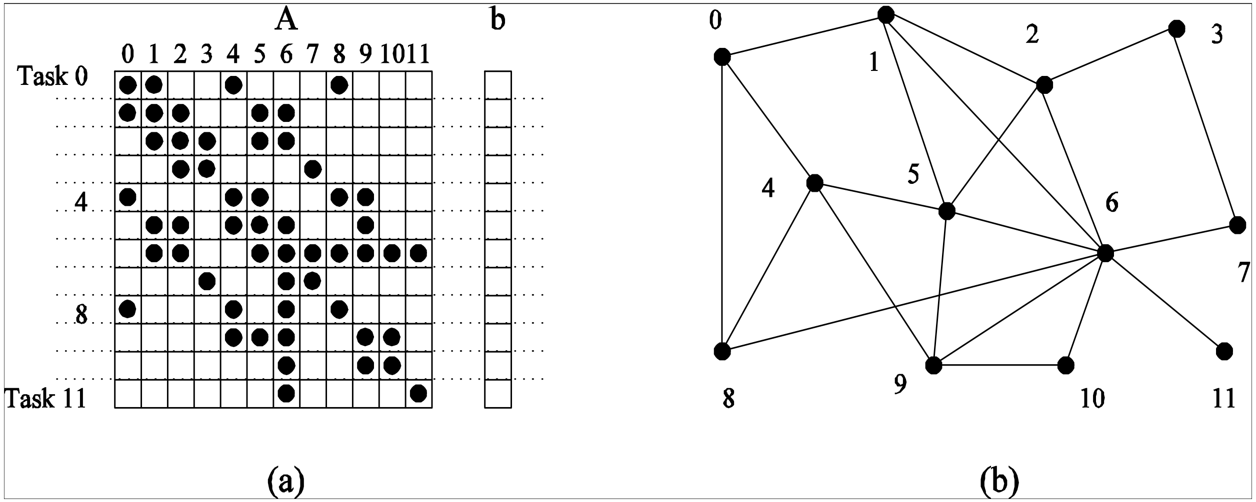
Task Interaction Graphs

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

Task Interaction Graphs: An Example

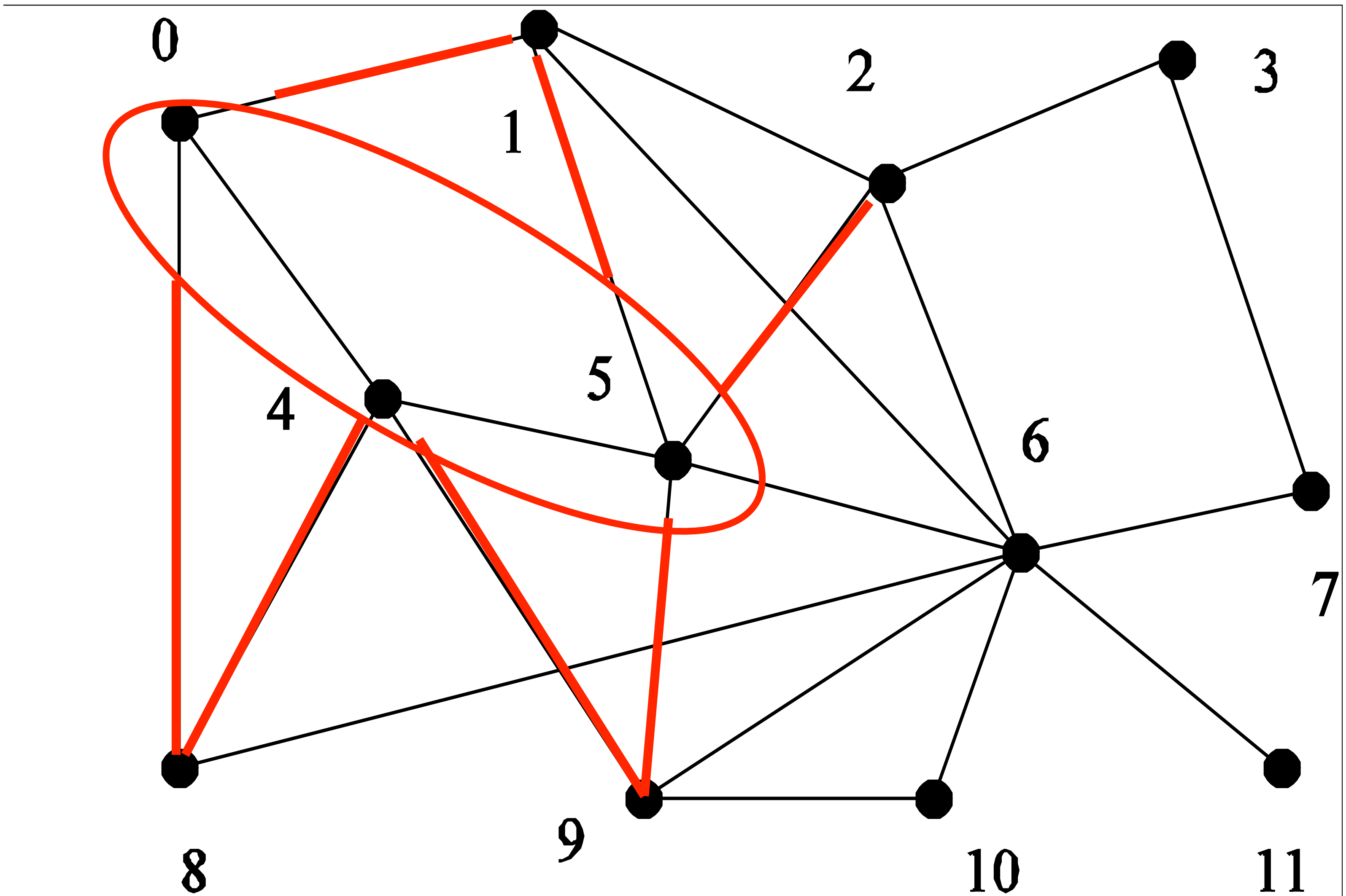
Consider the problem of multiplying a sparse matrix *A* with a vector *b*. The following observations can be made:

- 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.
- If, for memory optimality, we also partition ***b*** across tasks, 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).



Super Node Task Assignment (0,4,5 to Task)

Total # comm. edges = 7



Unique # comm. edges (count each destination node only once) = 4
(nodes 1,8,9,2)

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

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

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 four time units (four edges). Clearly, this is a more favorable ratio than the former case.

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 must also provide a mapping of tasks to processes.

Note: We refer to the mapping as being from tasks to processes, as opposed to processors.

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

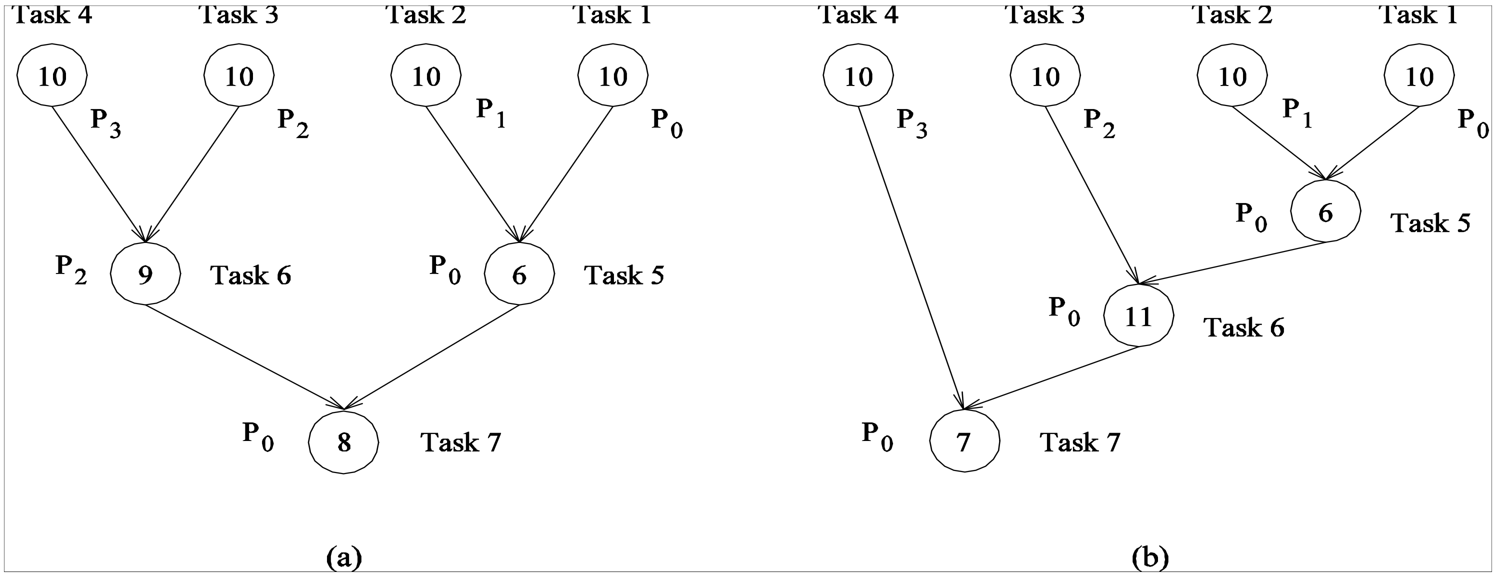
- Mapping independent tasks to different processes.

- 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.

Note: These criteria often conflict eith each other. For example, a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all!

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.

Decomposition Techniques

So how does one decompose a task into various subtasks?

These include:

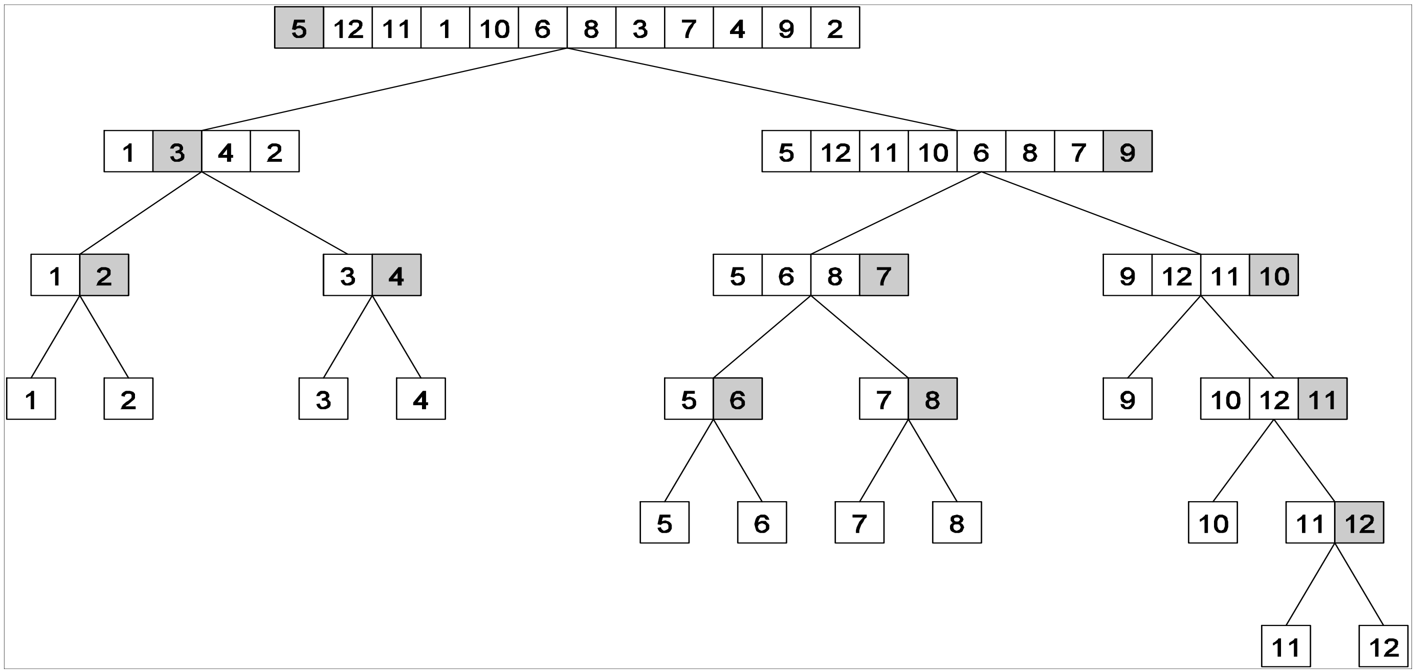
- **recursive decomposition**
- **data decomposition**
- **exploratory decomposition**
- **speculative decomposition**

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 (also applies to mergesort).



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 indeed any other associative operation such as sum, AND, etc.) can be fashioned as a divide-and-conquer algorithm. The following algorithm illustrates this.

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

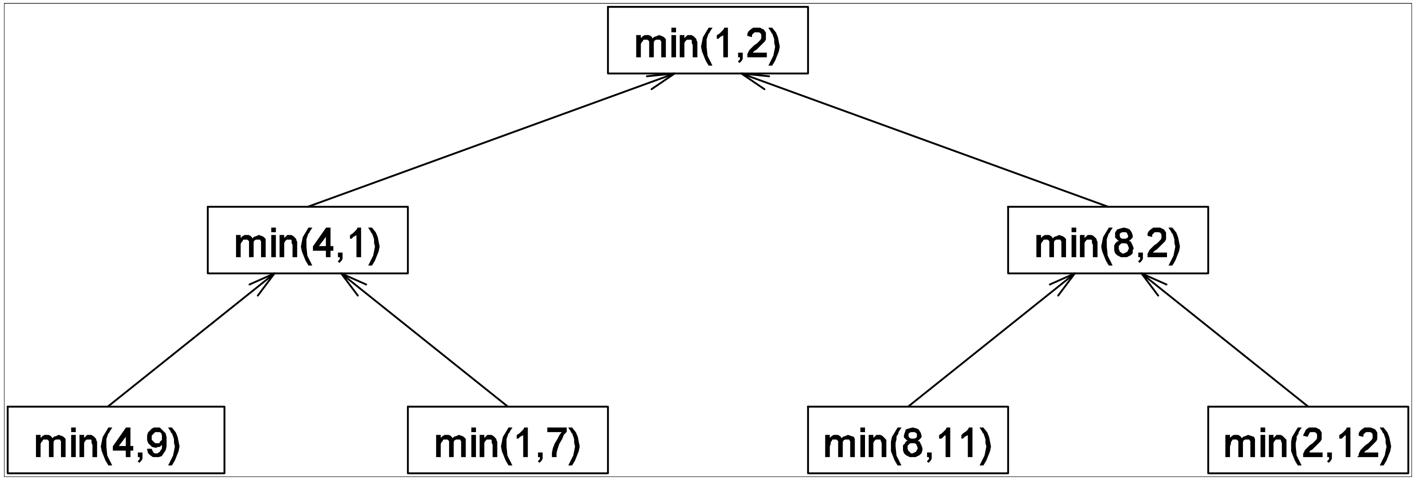
Recursive Decomposition: Example

We can rewrite the loop as follows:

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

Recursive Decomposition: Example

The code in the previous foil 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:



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

- 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.

Output Data Decomposition

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

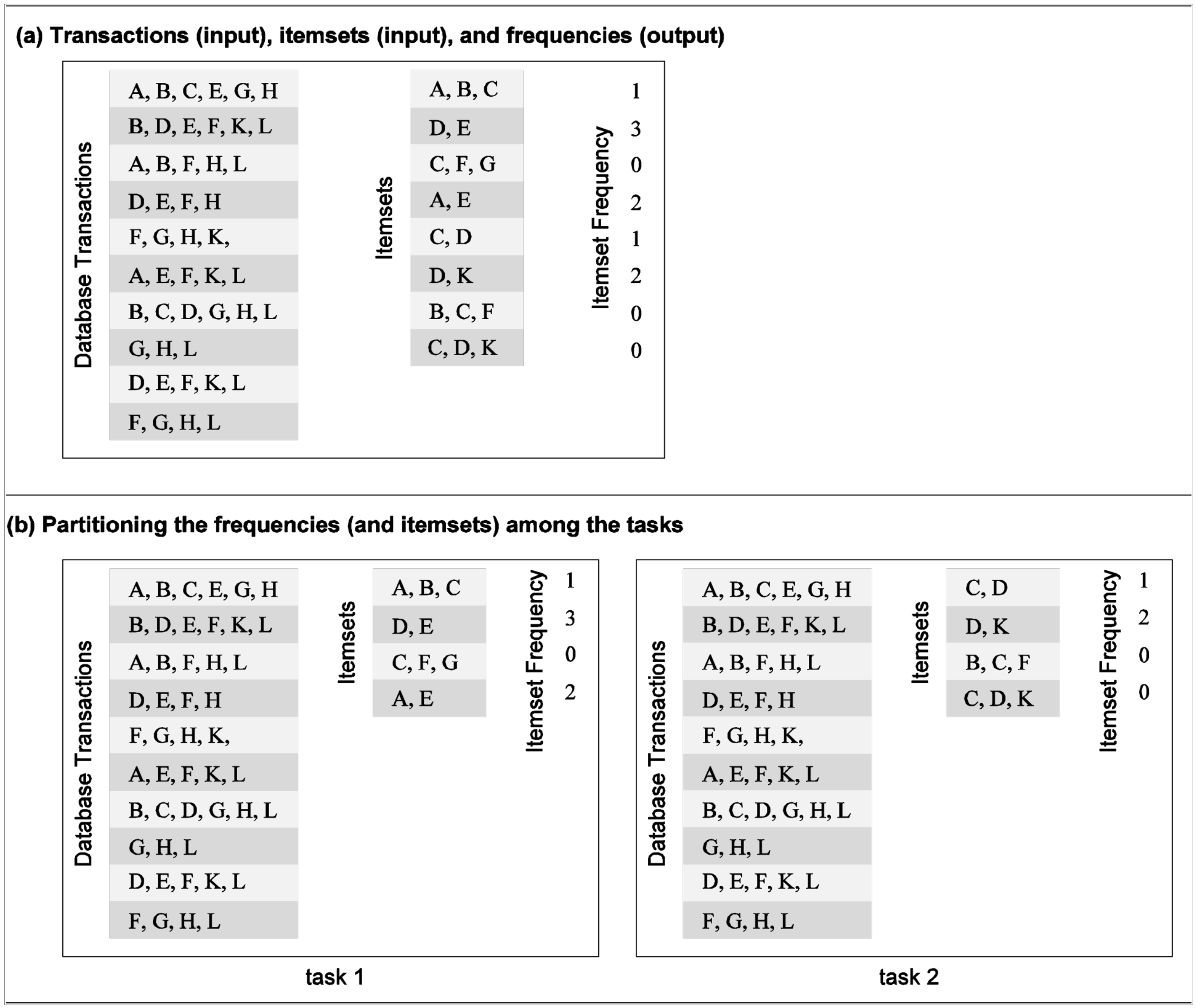
Output Data Decomposition: Example

A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previus foil, 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: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

Output Data Decomposition: Example

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



Output Data Decomposition: Example

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

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

Input Data 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.

Input Data Partitioning: Example

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

Partitioning the transactions among the tasks

task 1

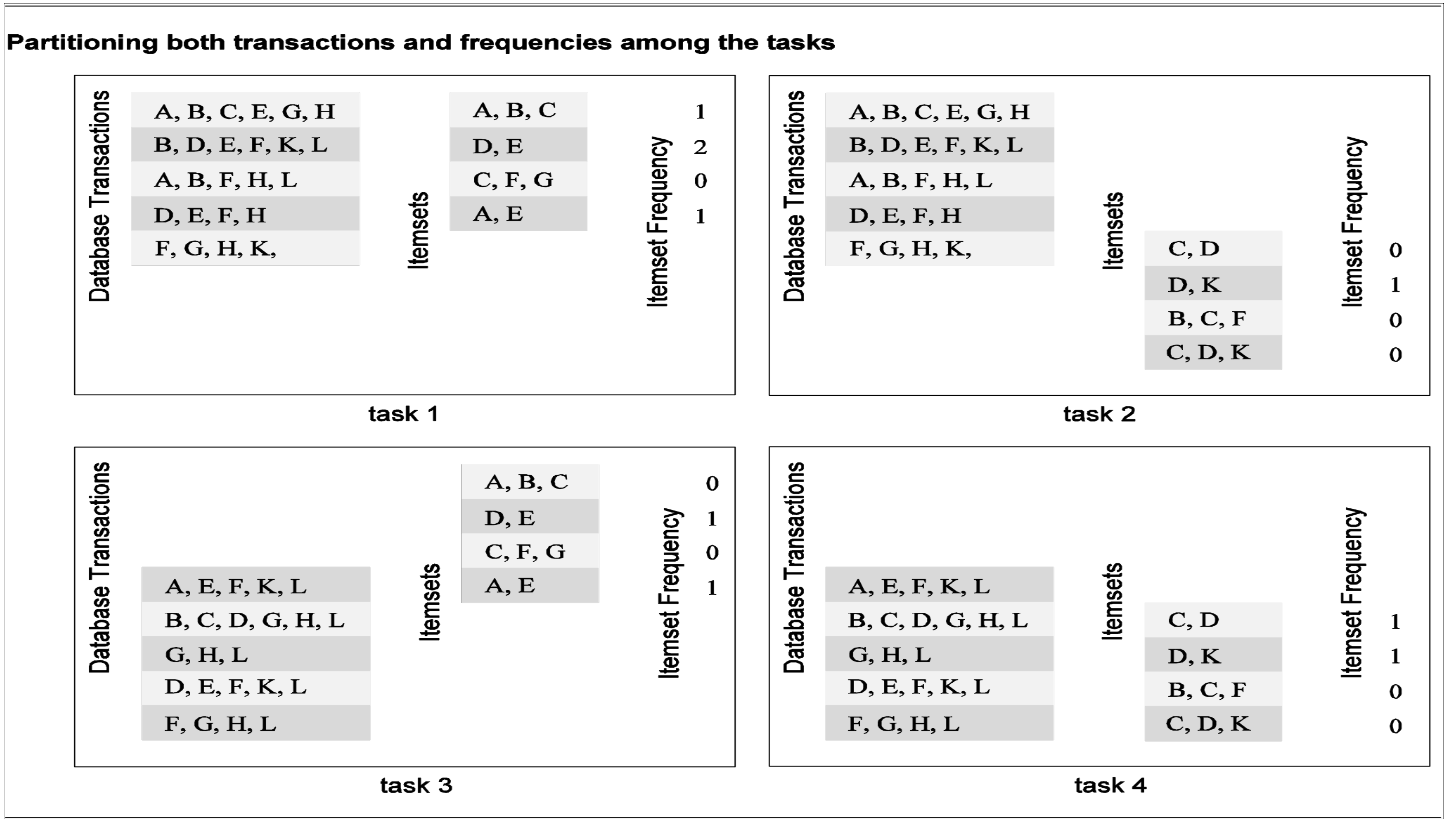
Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	2
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	1
F, G, H, K,	C, D	0
	D, K	1
	B, C, F	0
	C, D, K	0

task 2

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

Partitioning Input *and* Output Data

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



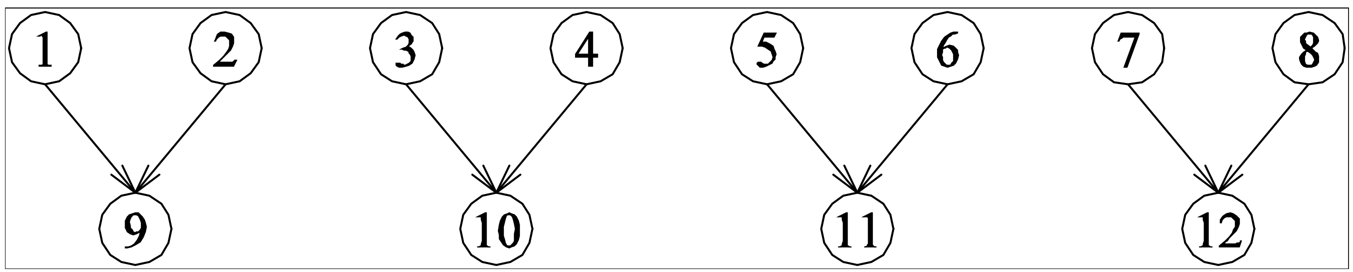
Intermediate Data Partitioning

- 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

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



The Owner Computes Rule

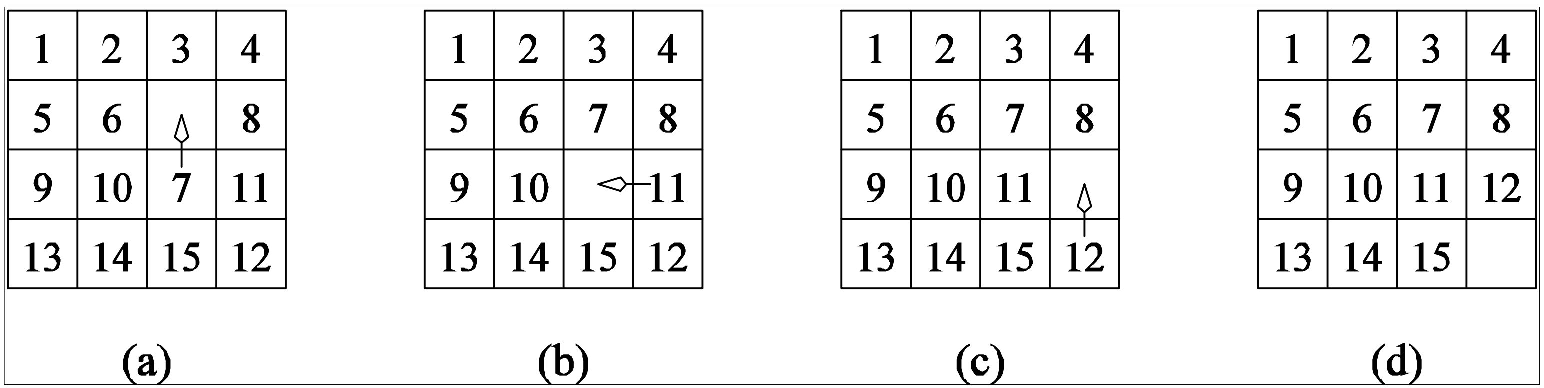
- Process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.
- In the case of output data decomposition, the owner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

Exploratory Decomposition

- 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 (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

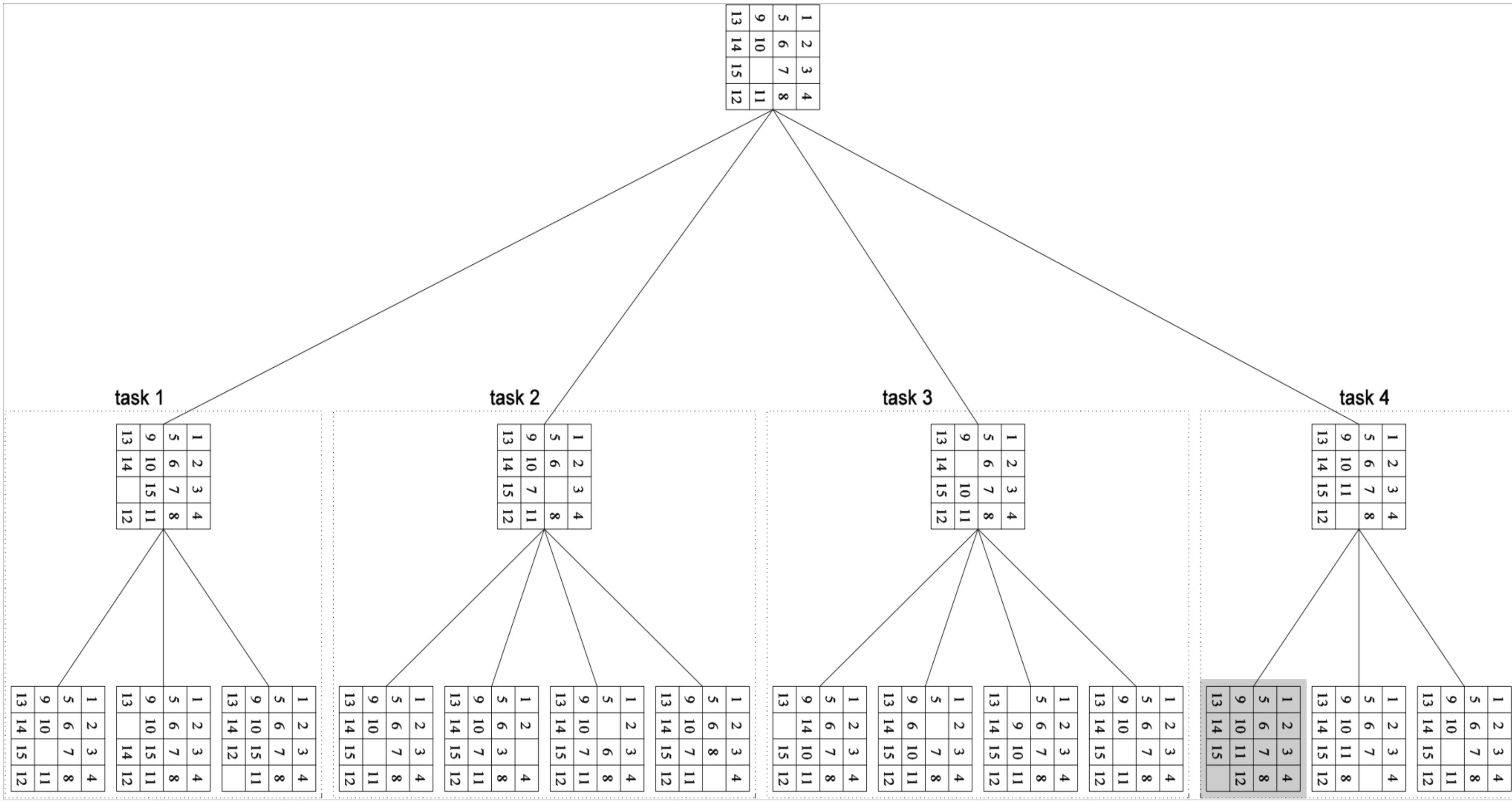
Exploratory Decomposition: Example

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).



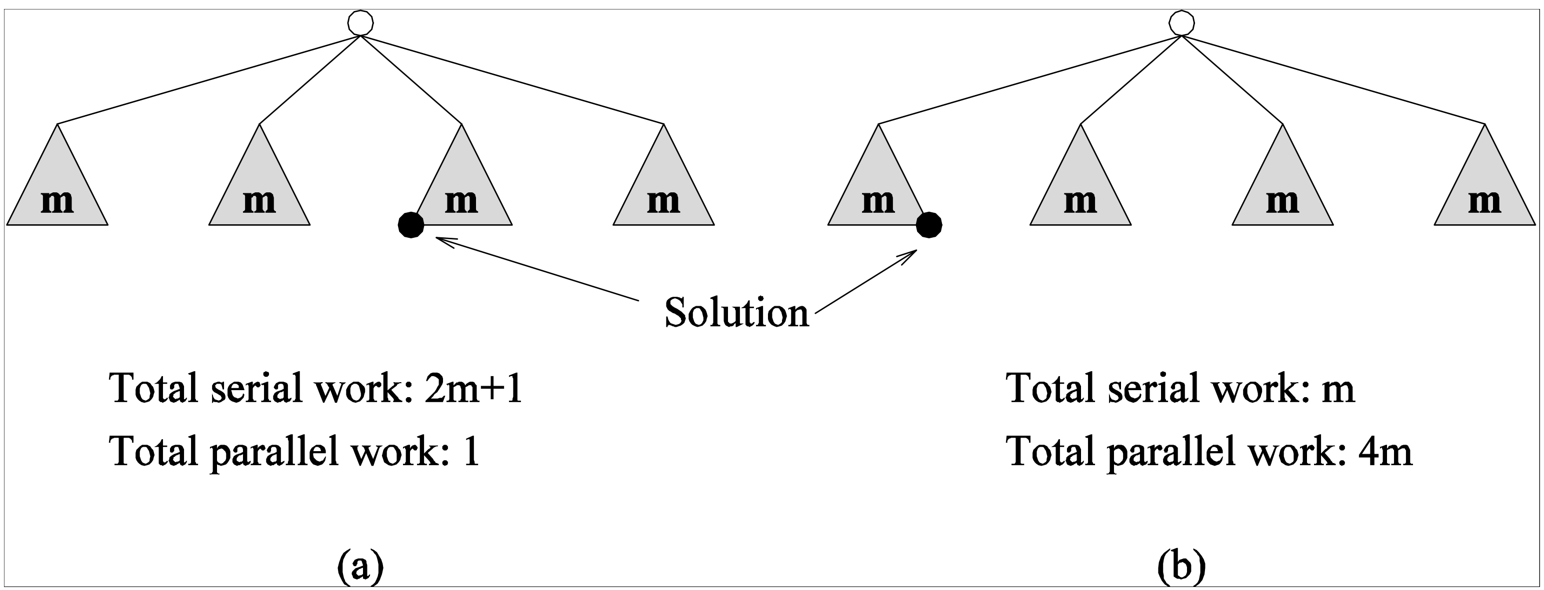
Of-course, the problem of computing the solution, in general, is much more difficult than in this simple example.

Exploratory Decomposition: Example



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.

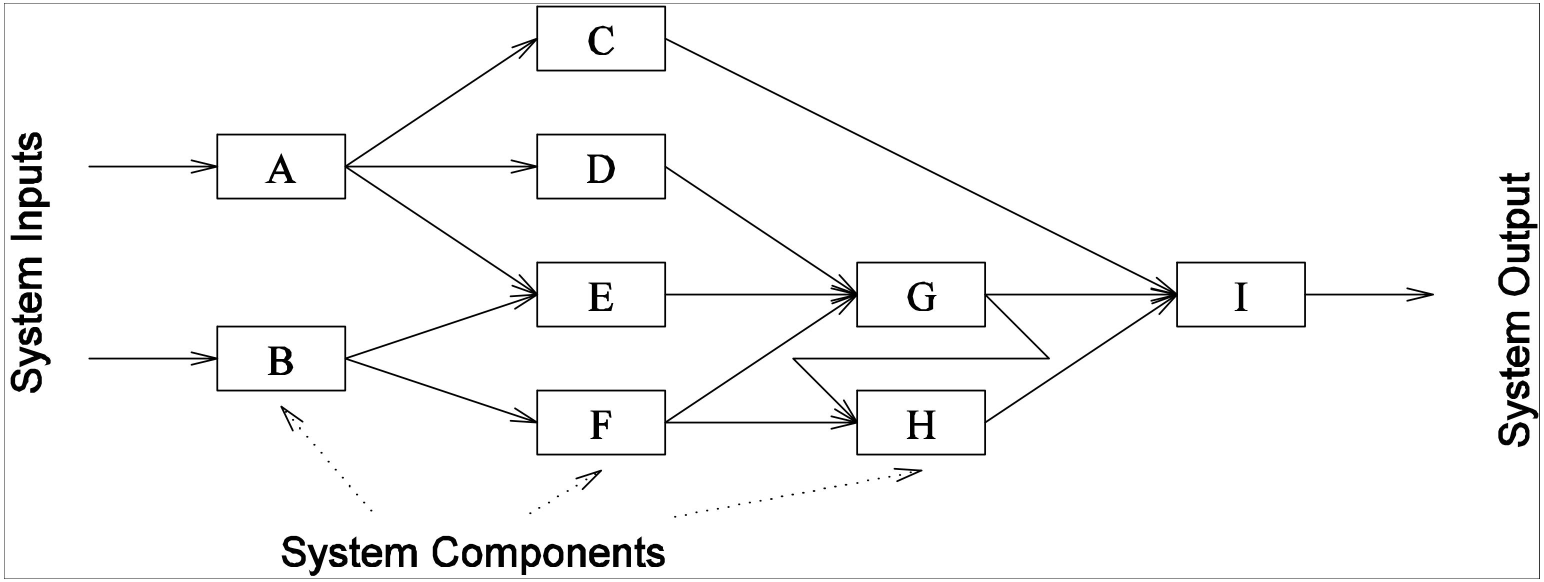


Speculative Decomposition

- 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 and optimistic
- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.

Speculative Decomposition: Example

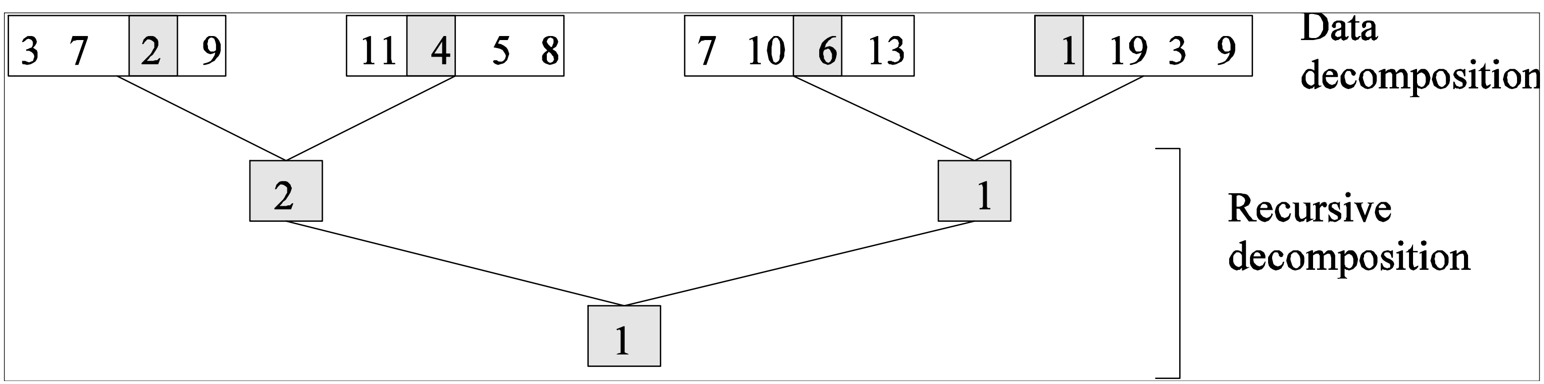
The task is to simulate the behavior of this network for various inputs and node delay parameters (note that networks may become unstable for certain values of service rates, queue sizes, etc.).



Hybrid Decompositions

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

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



Characteristics of Tasks

Once a problem has been decomposed into independent tasks, the characteristics of these tasks critically impact choice and performance of parallel algorithms. Relevant task characteristics include:

- **Task generation.**
- **Task sizes.**
- **Size of data associated with tasks.**

Task Generation

- **Static task generation:** Concurrent tasks can be identified a-priori. Typical matrix operations, graph algorithms, image processing applications

- **Dynamic task generation:** Tasks are generated as we perform computation e.g., Puzzles. These applications are typically decomposed using exploratory or speculative decompositions.

Task Sizes

- **Task sizes may be uniform**
  - (i.e., all tasks are the same size) or non-uniform.

- **Non-uniform task sizes**
  - may be such that they can be determined (or estimated) a-priori or not.

- **Examples**
  - discrete optimization problems, in which it is difficult to estimate the effective size of a state space.

Size of Data Associated with Tasks

- **A small context of a task**
  - implies that an algorithm can easily communicate this task to other processes dynamically (e.g., the 15 puzzle).

- **A large context of task**
  - an algorithm may attempt to reconstruct the context at another processes as opposed to communicating the context of the task (e.g., 0/1 integer programming).

Characteristics of Task Interactions

- **Tasks may communicate with each other in various ways. The associated dichotomy is:**

- **Static interactions:**
  - The tasks and their interactions are known a-priori. These are relatively simpler to code into programs.

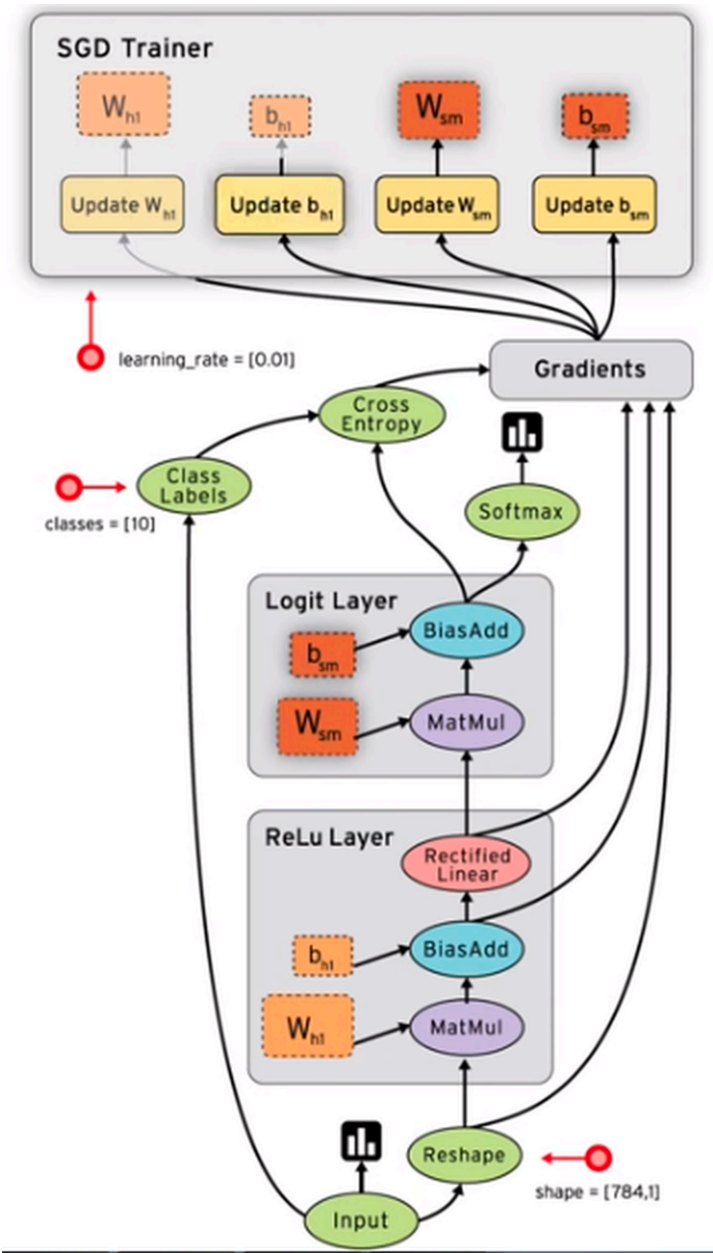
- **Dynamic interactions:**
  - The timing or interacting tasks cannot be determined a-priori. These interactions are harder to code, especitally, as we shall see, using message passing APIs.

Characteristics of Task Interactions

- **Regular interactions:**
  - There is a definite pattern (in the graph sense) to the interactions. These patterns can be exploited for efficient implementation.

- **Irregular interactions:**
  - Interactions lack well-defined topologies.

Characteristics of Task Interactions: Example



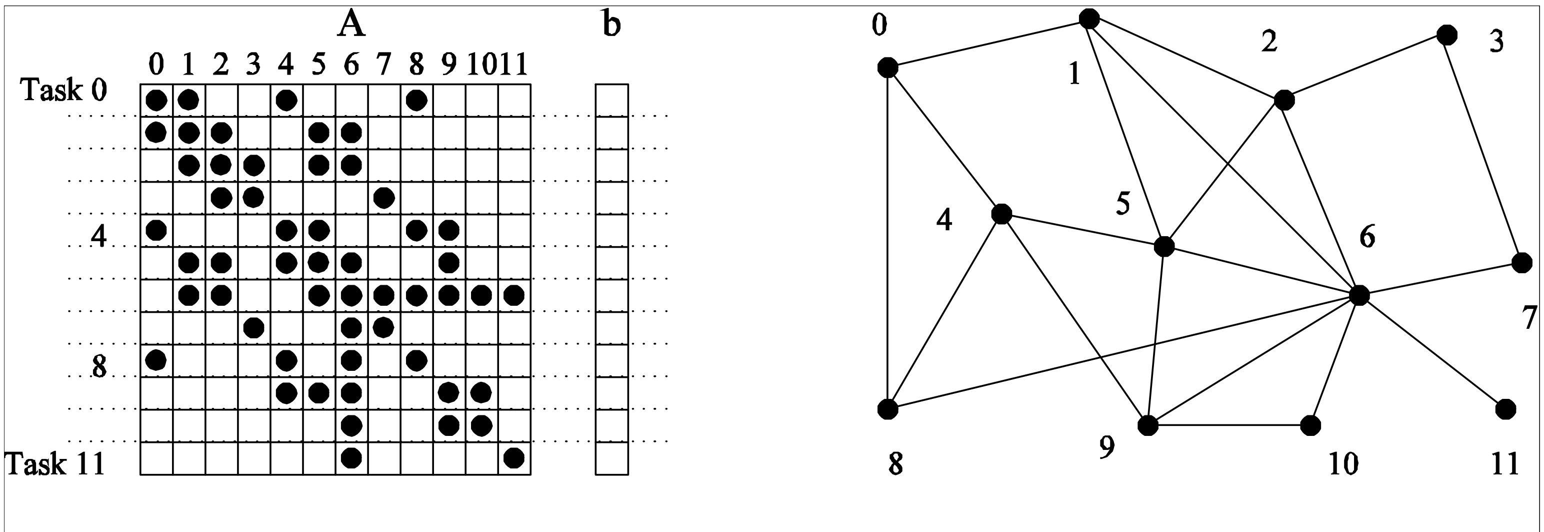

```
x = tf.placeholder(tf.float32, [None
, img_size_flat])
```

```
weights = tf.Variable(tf.zeros([i
mg_size_flat, num_classes]))
```

```
logits = tf.matmul(x, weights) + biases
```

Characteristics of Task Interactions:

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.



(a) (b)

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 modily data items associated with other tasks.
- In general, read-write interactions are harder to code, since they require additional synchronization.

Characteristics of Task Interactions

- Interactions may be one-way or two-way.
- A one-way interaction can be initiated and accomplished by one of the two interacting tasks.
- A two-way interaction requires participation from both tasks involved in an interaction.
- One way interactions are somewhat harder to code in message passing APIs.

Mapping Techniques

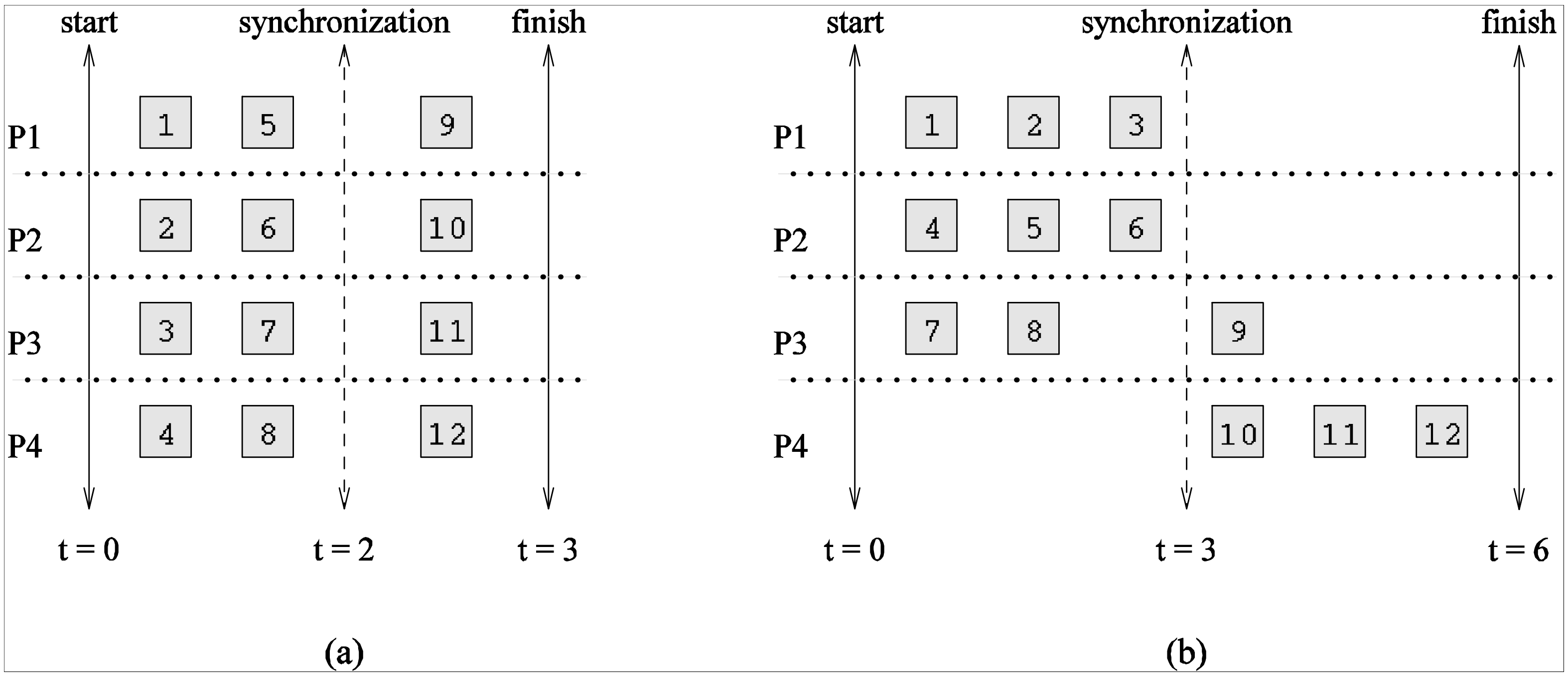
- **Once a problem has been decomposed into concurrent tasks, these must be mapped to processes (that can be executed on a parallel platform).**
  - Mappings must minimize overheads.
  - Primary overheads are communication and idling.

- **Minimizing these overheads often represents contradicting objectives.**

- **Assigning all work to one processor trivially minimizes communication at the expense of significant idling.**

Mapping Techniques for Minimum Idling

Mapping must simultaneously minimize idling and load balance. Merely balancing load does not minimize idling.



Mapping Techniques for Minimum Idling

Mapping techniques can be static or dynamic.

- **Static Mapping:** Tasks are mapped to processes a-priori. For this to work, we must have a good estimate of the size of each task. Even in these cases, the problem may be NP complete.
- **Dynamic Mapping:** Tasks are mapped to processes at runtime. This may be because the tasks are generated at runtime, or that their sizes are not known.

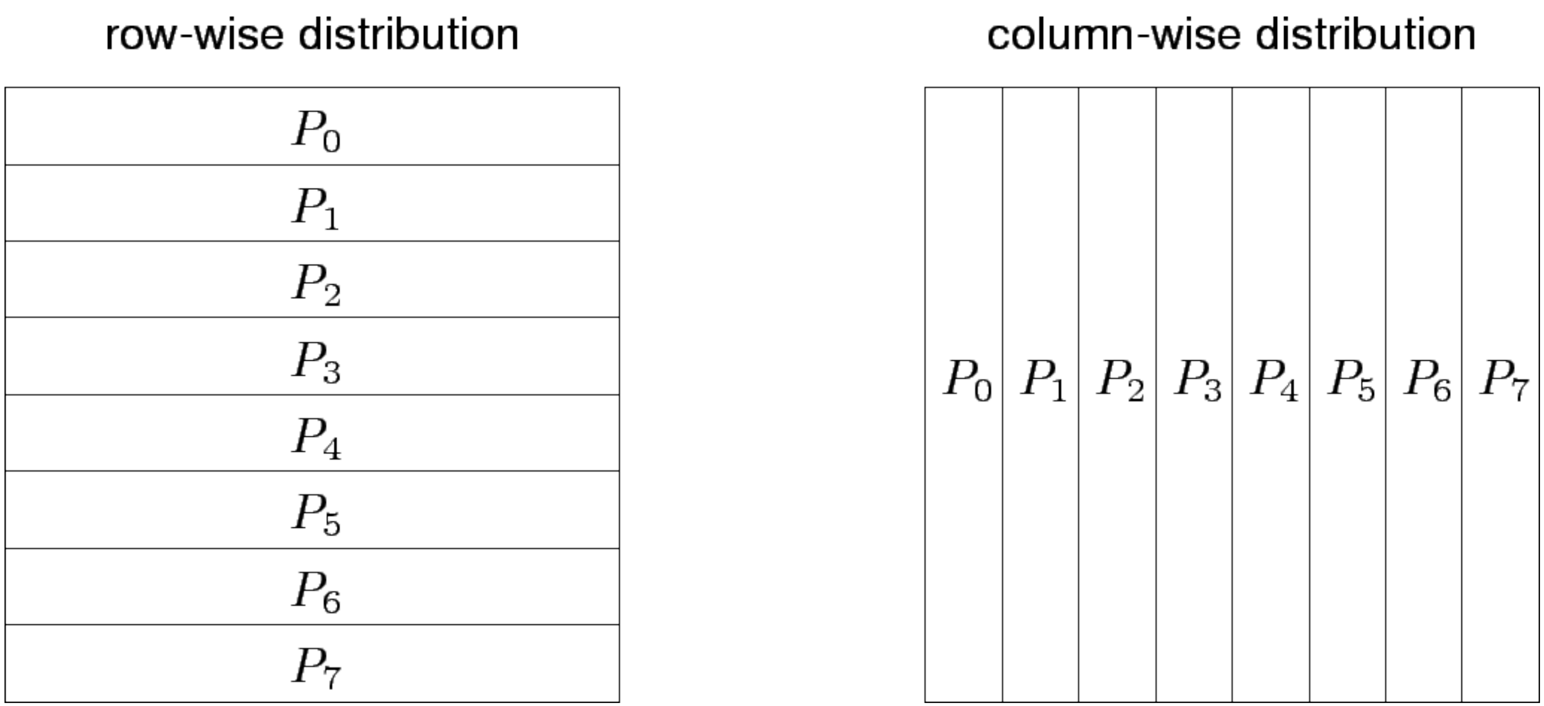
Other factors that determine the choice of techniques include the size of data associated with a task and the nature of underlying domain.

Schemes for Static Mapping

- **Mappings based on data partitioning.**
- **Mappings based on task graph partitioning.**
- **Hybrid mappings.**

Mappings Based on Data Partitioning

We can combine data partitioning with the ``owner-computes'' rule to partition the computation into subtasks. The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.



Block Array Distribution Schemes

Block distribution schemes can be generalized to higher dimensions as well.