# COMP 422, Lecture 4: Decomposition Techniques for Parallel Algorithms

(Sections 3.1 & 3.2 of textbook)

**Vivek Sarkar**

**Department of Computer Science**
**Rice University**

**vsarkar@rice.edu**

# Recap of Lecture 3

- **Interconnection Networks**
  - —Static (direct) vs. Dynamic (indirect) networks
  - —Metrics: diameter, bisection width, arc connectivity, cost
  - —Dynamic networks: Crossbar, Omega, (Fat) Tree
  - —Static networks: Mesh, Torus, Hypercube
- **Cache Coherence**
  - —Invalidate vs. Update protocols
  - —Bus Snooping vs. Directory-based
  - —False sharing
- **Communication Costs**
  - —Store-and-forward vs. Cut-through

# Lecture 3 Review Question

- Consider the following parallel computation
  - CPU 0: Update even-numbered rows

```
for ( j = 0 ; j < N ; j += 2 )
    for ( k = 0 ; k < N ; k++ )
        A[j,k] = f(j,k);
```

  - CPU 1: Update odd-numbered rows

```
for ( j = 1 ; j < N ; j += 2 )
    for ( k = 0 ; k < N ; k++ )
        A[j,k] = g(j,k);
```

- Assume that a cache line size of 32B, an invalidate protocol, an element size of 8B for A, and that A's first element is aligned on a cache line boundary. What is the maximum number of invalidate messages that will be sent between CPU 0 and 1 as a function of N?

# Acknowledgments for today's lecture

- Cilk lecture by Charles Leiserson and Bradley Kuszmaul (Lecture 1, Scheduling Theory)
  - http://supertech.csail.mit.edu/cilk/lecture-1.pdf
- Slides accompanying course textbook
  - http://www-users.cs.umn.edu/~karypis/parbook/

# Outline of Today's Lecture

- **Tasks, Dependence Graphs, Scheduling Theory**
- **Data and Computation Decompositions**

# Tasks and Dependency Graphs

- **The first step in developing a parallel algorithm is to decompose the problem into *tasks* that are candidates for parallel execution**
- **Task = indivisible sequential unit of computation**
- **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: 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 |

**Table 3.1** A database storing information about used vehicles.

# 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.**

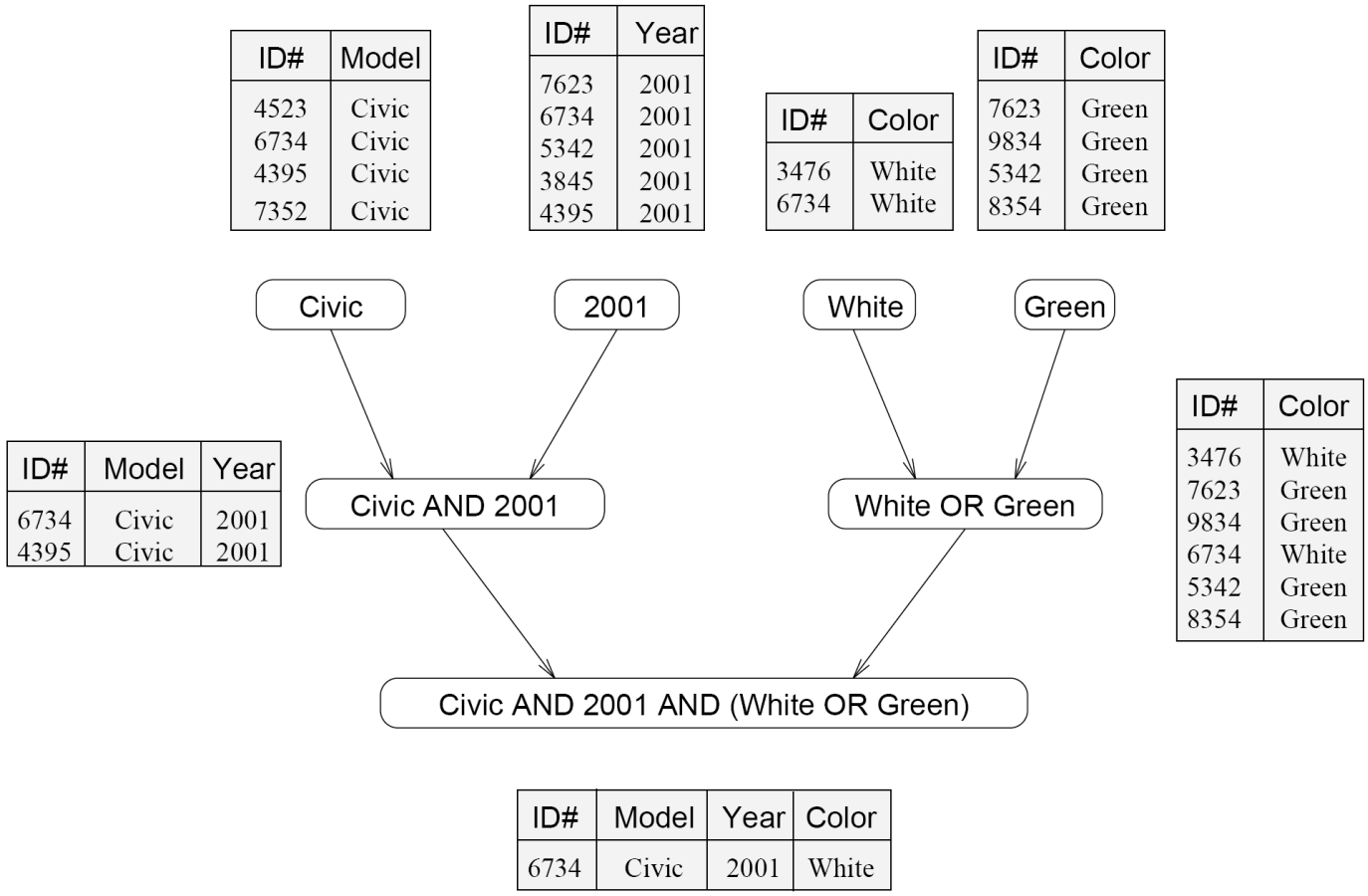



**Figure 3.2** The different tables and their dependencies in a query processing operation.

# An Alternate Task Decomposition and Dependency Graph

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

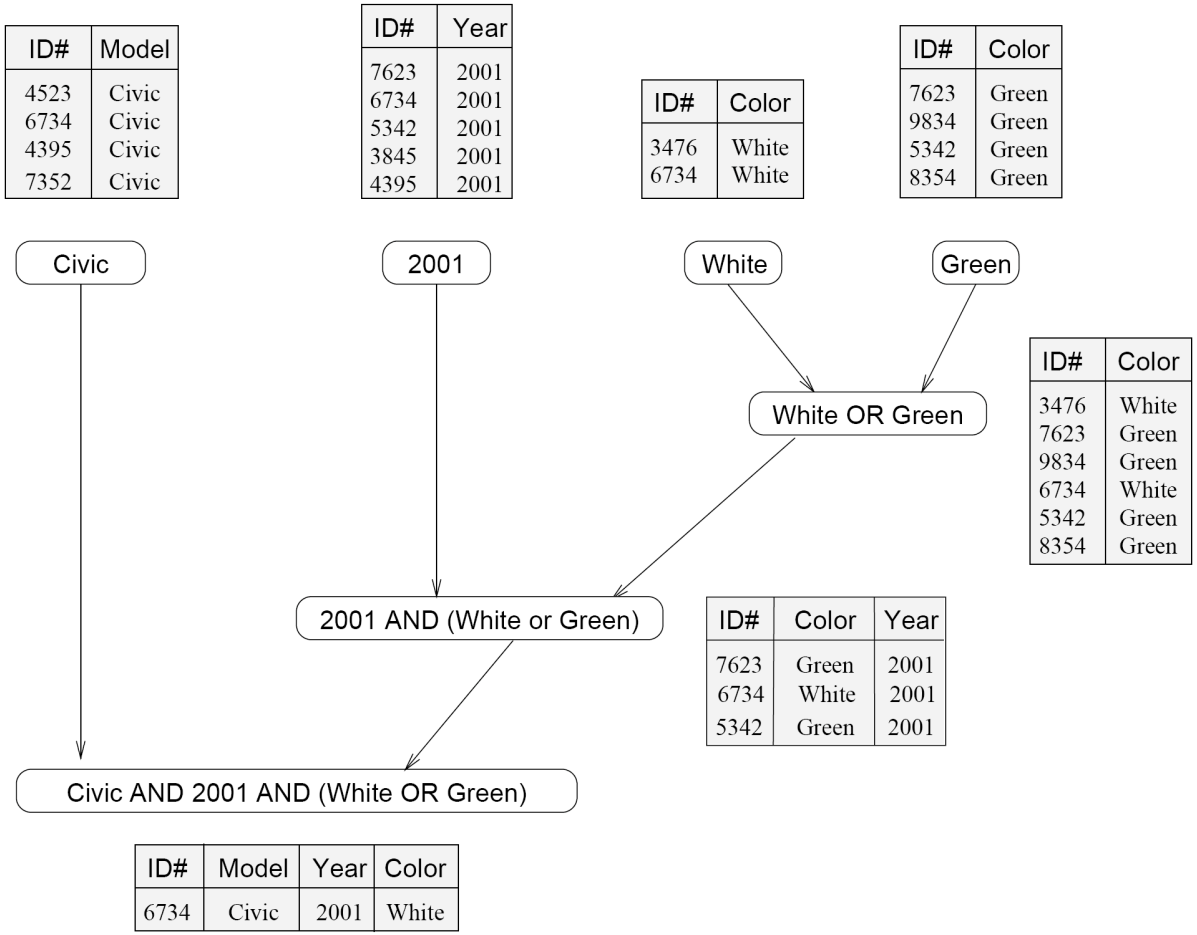



**Figure 3.3** An alternate data-dependency graph for the query processing operation.

# 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*.
- The ratio of the total amount of work to the critical path length is the *average degree of concurrency*.

# Examples of Critical Path Length

**Consider the task dependency graphs of the two database query decompositions:**

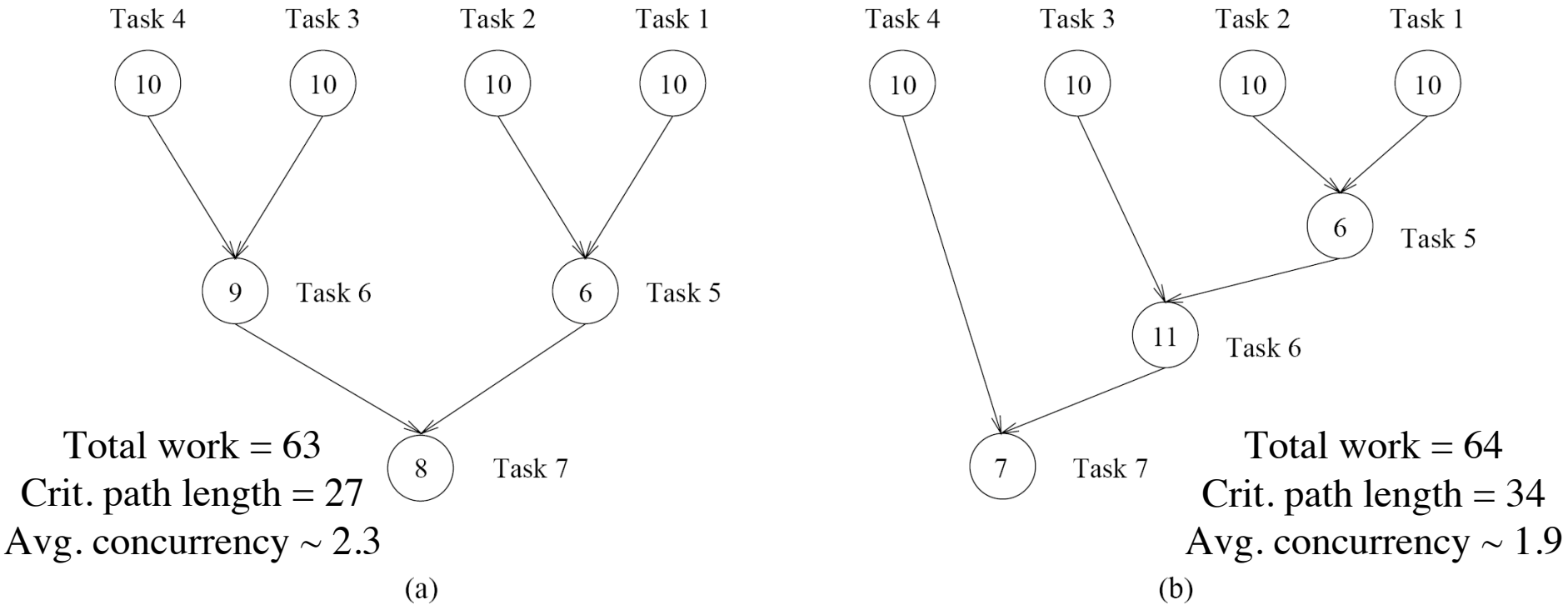
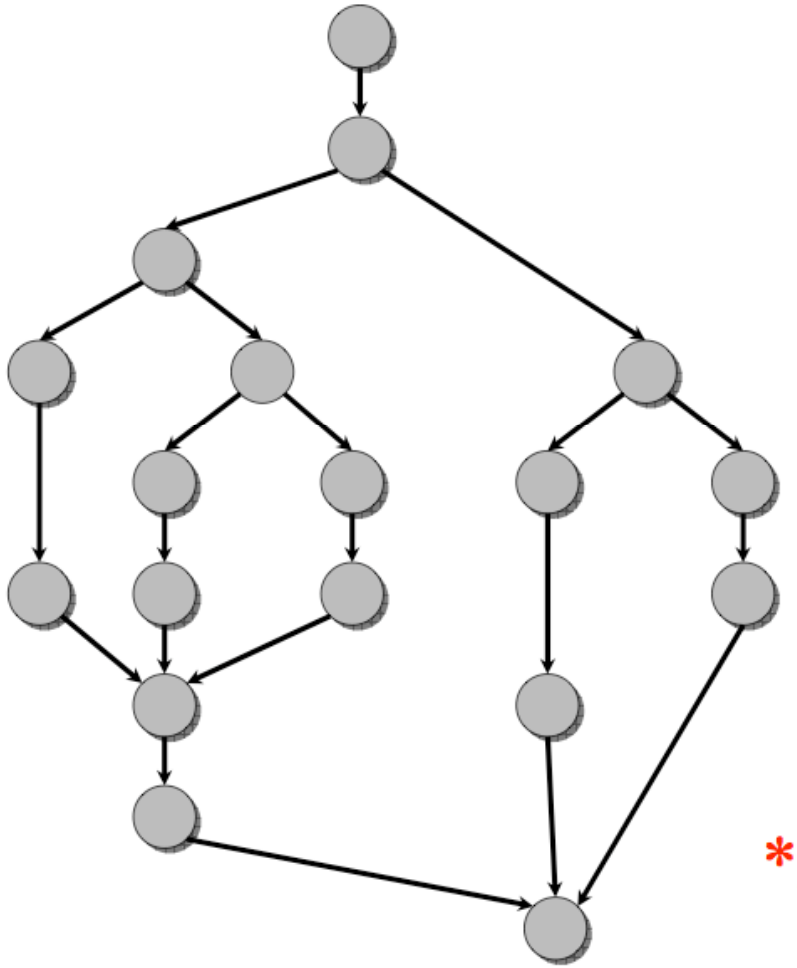



**Figure 3.5** Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.

# Algorithmic Complexity Measures (Ignoring Communication Overhead)

$T_P$ = execution time on $P$ processors



$T_1$ = ***work***

$T_\infty$ = ***span****

**LOWER BOUNDS**

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

*Also called ***critical-path length*** or ***computational depth***.

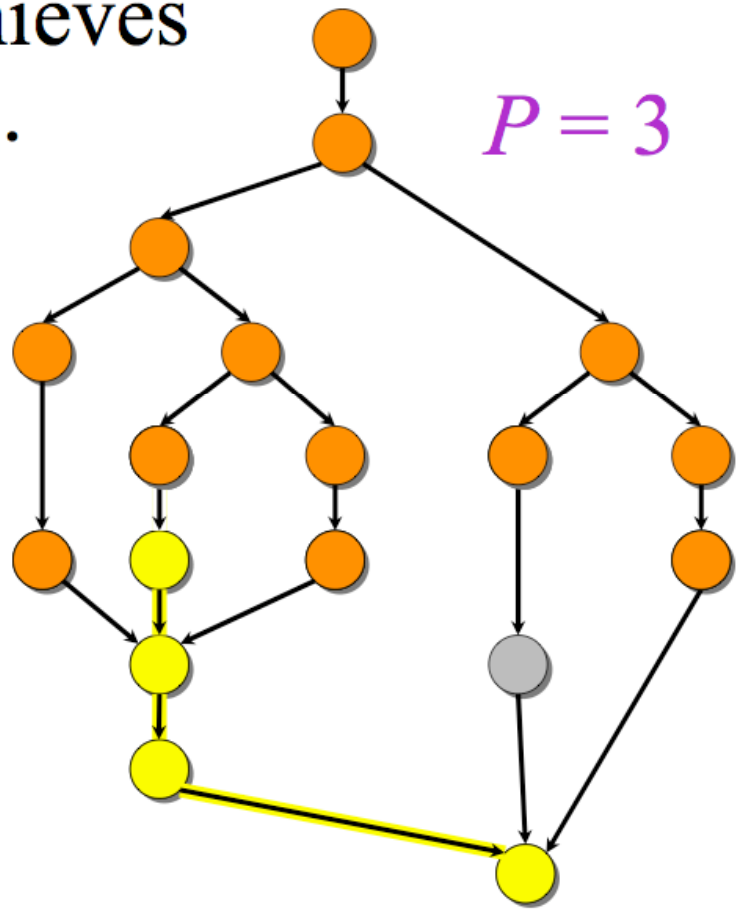
# Upper Bounds on $T_P$

***Theorem*** [Graham '68 & Brent '75]. Any greedy* scheduler achieves

$$T_P \leq T_1/P + T_\infty.$$

*Proof.*

- # complete steps $\leq T_1/P$, since each complete step performs $P$ work.
- # incomplete steps $\leq T_\infty$, since each incomplete step reduces the span of the unexecuted dag by 1. ■




* Greedy scheduler ==> no unenforced idleness

# Performance Bound for Greedy Algorithm

***Corollary.*** Any greedy scheduler achieves within a factor of 2 of optimal.

*Proof.* Let $T_P^*$ be the execution time produced by the optimal scheduler.

Since $T_P^* \geq \max\{T_1/P, T_\infty\}$ (lower bounds), we have

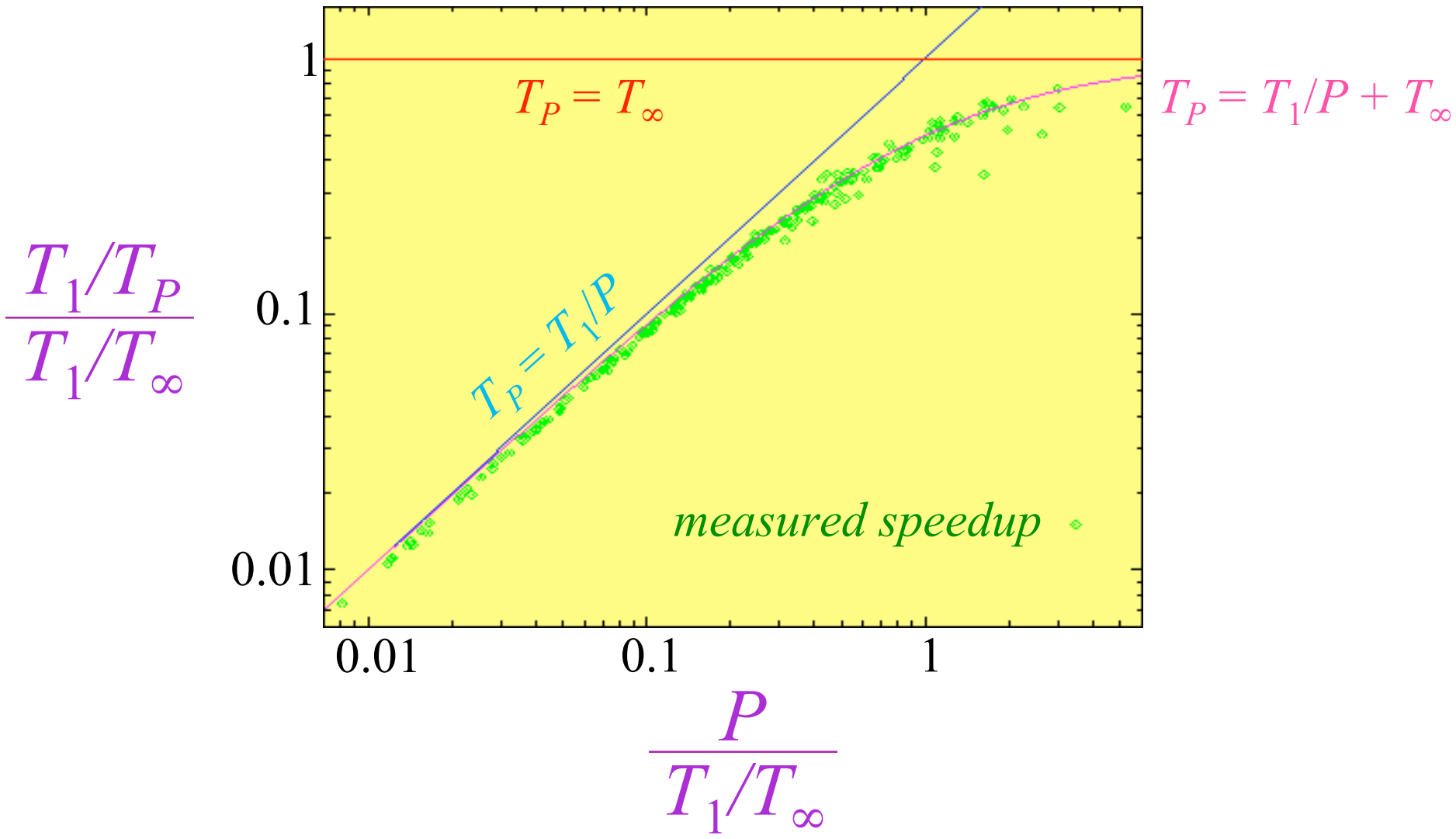
$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \cdot \max\{T_1/P, T_\infty\} \\ &\leq 2T_P^* \ . \ \blacksquare \end{aligned}$$

NOTE: performance bound approaches 1 (optimal) when one of the max terms dominates the other

# Case Study: Cilk Chess Programs

- ⋆*Socrates* placed 3rd in the 1994 International Computer Chess Championship running on NCSA's 512-node Connection Machine CM5.
- ⋆*Socrates 2.0* took 2nd place in the 1995 World Computer Chess Championship running on Sandia National Labs' 1824-node Intel Paragon.
- *Cilkchess* placed 1st in the 1996 Dutch Open running on a 12-processor Sun Enterprise 5000. It placed 2nd in 1997 and 1998 running on Boston University's 64-processor SGI Origin 2000.
- *Cilkchess* tied for 3rd in the 1999 WCCC running on NASA's 256-node SGI Origin 2000.

# ★Socrates Normalized Speedup



# Developing ★Socrates

- For the competition, ★Socrates was to run on a 512-processor Connection Machine Model CM5 supercomputer at the University of Illinois.
- The developers had easy access to a similar 32-processor CM5 at MIT.
- One of the developers proposed a change to the program that produced a speedup of over 20% on the MIT machine.
- After a back-of-the-envelope calculation, the proposed "improvement" was rejected!

# ★Socrates Speedup Paradox

| *Original program* | *Proposed program* |
|---|---|
| $T_{32} = 65$ seconds | $T'_{32} = 40$ seconds |

$$T_P \approx T_1/P + T_\infty$$

| | |
|---|---|
| $T_1 = 2048$ seconds | $T'_1 = 1024$ seconds |
| $T_\infty = 1$ second | $T'_\infty = 8$ seconds |
| $T_{32} = 2048/32 + 1$ $= 65$ seconds | $T'_{32} = 1024/32 + 8$ $= 40$ seconds |
| $T_{512} = 2048/512 + 1$ $= 5$ seconds | $T'_{512} = 1024/512 + 8$ $= 10$ seconds |

# Outline of Today's Lecture

- Tasks, Dependence Graphs, Scheduling Theory
- Data and Computation Decompositions

# Decomposition Techniques: Patterns for Parallel Algorithms

**So how does one decompose a task into various subtasks?**

**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:**

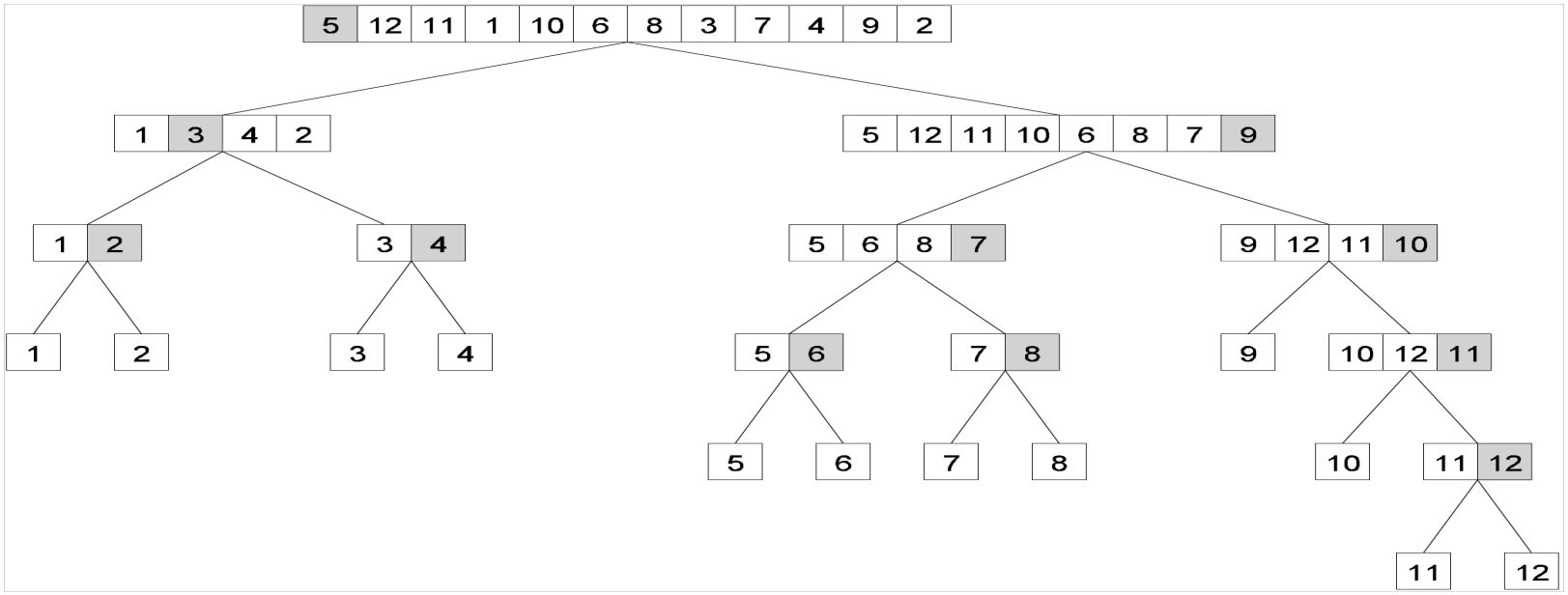
- **recursive decomposition**
- **data decomposition**
- **exploratory decomposition**
- **speculative decomposition**

# 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, a task represents the work of partitioning a (sub)array. Note that each subarray represents an independent subtask. This can be repeated recursively.**

# Data Decomposition

- **Identify the data on which computations are performed.**
- **Partition data into sub-units**
- **Data can be *input*, *output* or *intermediate* for different computations**
- **The data partitioning induces one or more decompositions of the computation into tasks e.g., by using the *owner computes rule***

# 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 submatrices as follows:**

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

**(a)**

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

**(b)**

**Figure 3.10** (a) Partitioning of input and output matrices into $2 \times 2$ submatrices.

# Output Data Decomposition: Example

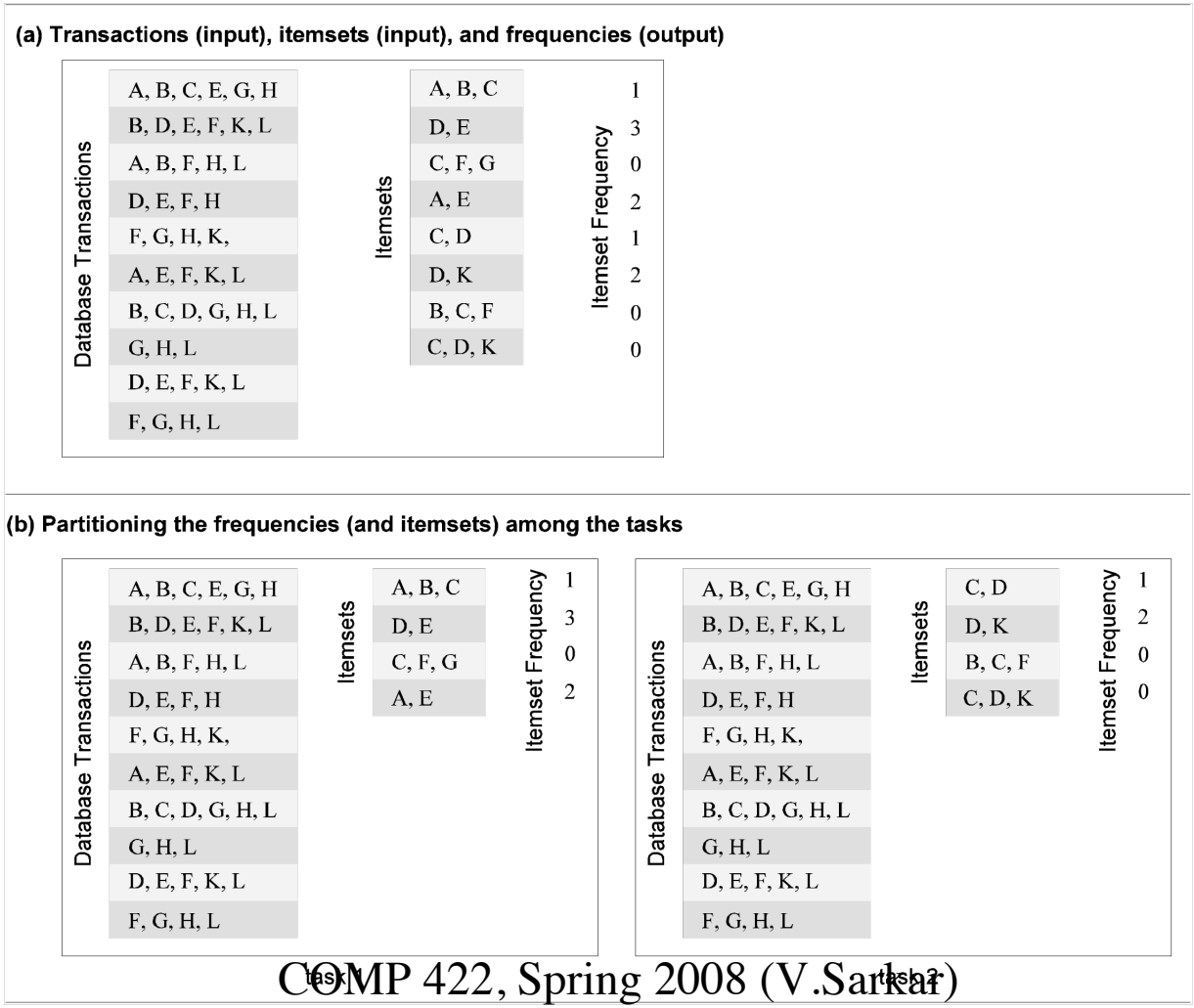
**A partitioning of output data does not result in a unique decomposition into tasks. Here are two possible task decompositions for the output data decomposition from the previous slide:**

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

**Figure 3.11** Two examples of decomposition of matrix multiplication into eight tasks.

# 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.**

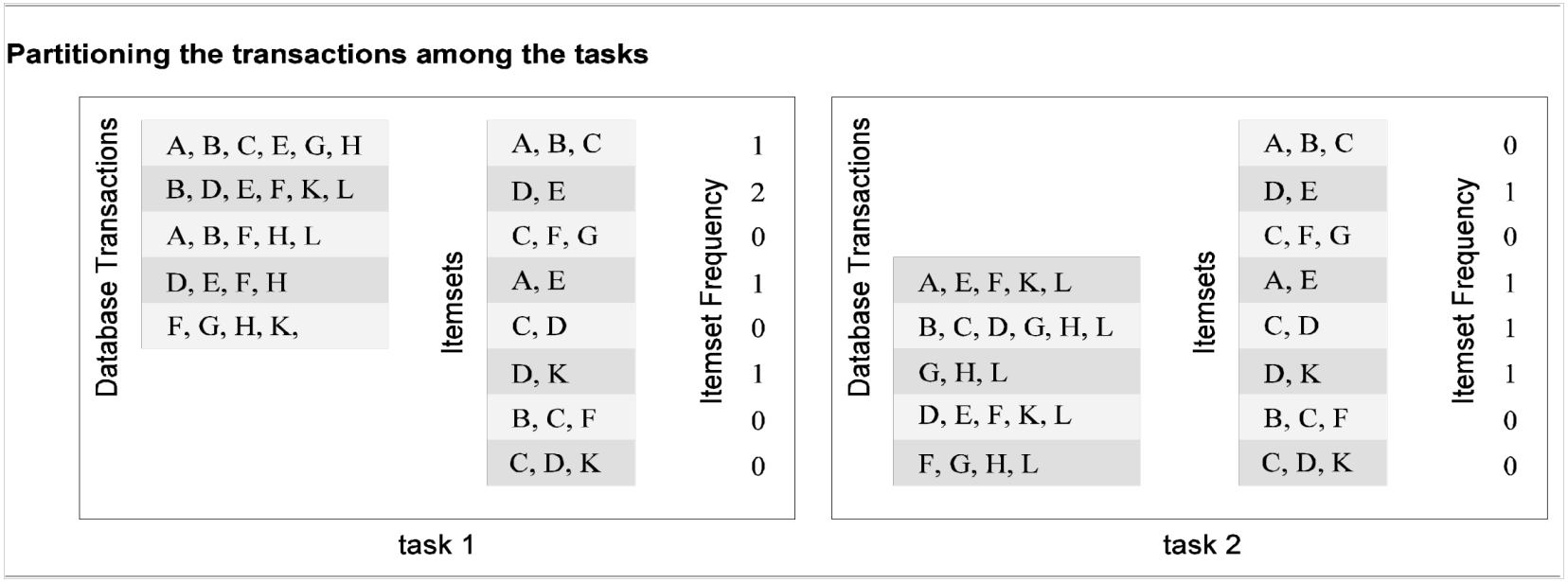


# Input Data Decomposition

- 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.

# Input Data Decomposition: 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.**



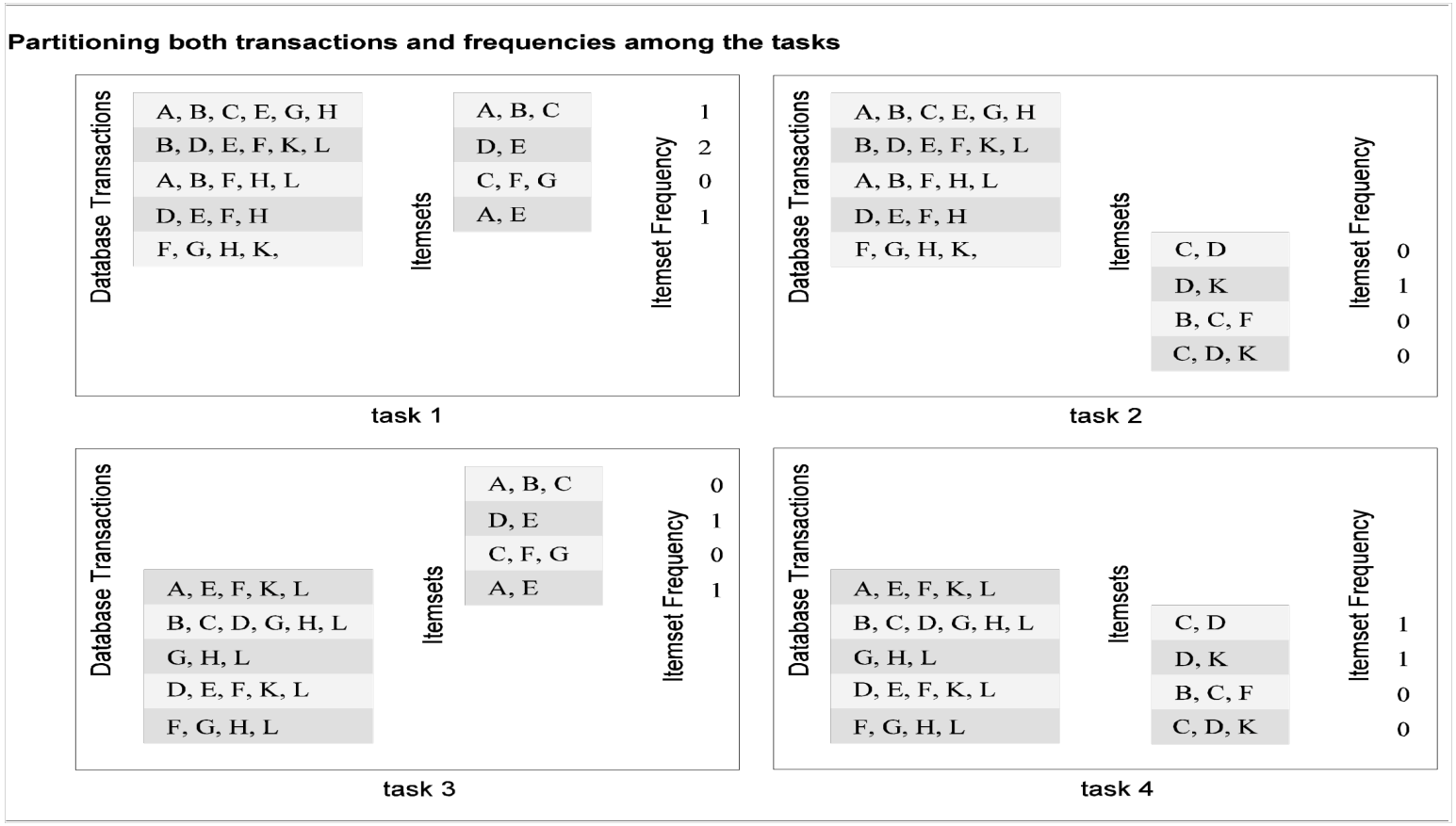
# Output vs. Input Data Decompositions

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

- **If only the output is decomposed and the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.**
- **If the input database is also partitioned (for scalability), it induces a computation mapping in which each task computes partial counts, and additional tasks are used to aggregate the counts.**

# Combining Input *and* Output Data Decompositions

**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:**

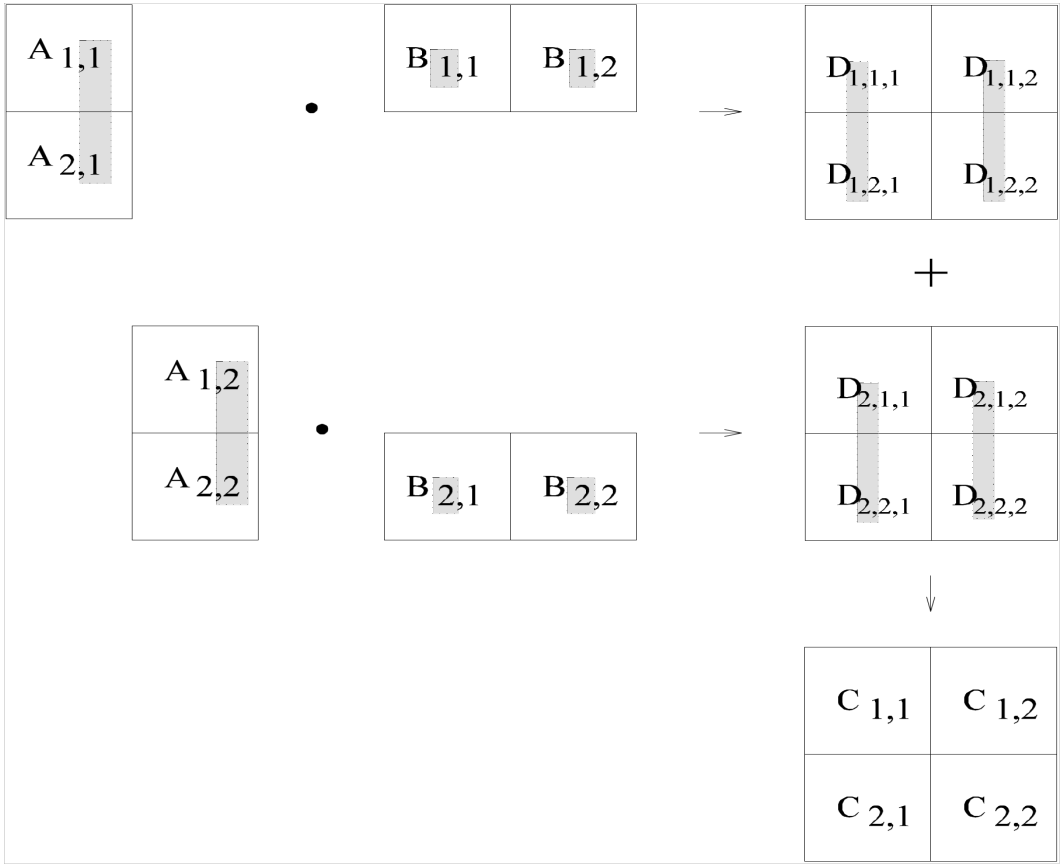


# Intermediate Data Decomposition

- **Computation can often be viewed as a sequence of transformation from the input to the output data.**
- **In these cases, it is sometimes beneficial to use one of the intermediate stages as a basis for decomposition.**

# Intermediate Data Partitioning: Example

**Consider the intermediate submatrices that can be created in dense matrix multiplication.**



# Intermediate Data Partitioning: Example

Stage I

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

Stage II

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

A decomposition induced by a partitioning of $D$

$$\begin{aligned}
\text{Task 01:} &\quad D_{1,1,1} = A_{1,1}B_{1,1} \\
\text{Task 02:} &\quad D_{2,1,1} = A_{1,2}B_{2,1} \\
\text{Task 03:} &\quad D_{1,1,2} = A_{1,1}B_{1,2} \\
\text{Task 04:} &\quad D_{2,1,2} = A_{1,2}B_{2,2} \\
\text{Task 05:} &\quad D_{1,2,1} = A_{2,1}B_{1,1} \\
\text{Task 06:} &\quad D_{2,2,1} = A_{2,2}B_{2,1} \\
\text{Task 07:} &\quad D_{1,2,2} = A_{2,1}B_{1,2} \\
\text{Task 08:} &\quad D_{2,2,2} = A_{2,2}B_{2,2} \\
\text{Task 09:} &\quad C_{1,1} = D_{1,1,1} + D_{2,1,1} \\
\text{Task 10:} &\quad C_{1,2} = D_{1,1,2} + D_{2,1,2} \\
\text{Task 11:} &\quad C_{2,1} = D_{1,2,1} + D_{2,2,1} \\
\text{Task 12:} &\quad C_{2,2} = D_{1,2,2} + D_{2,2,2}
\end{aligned}$$

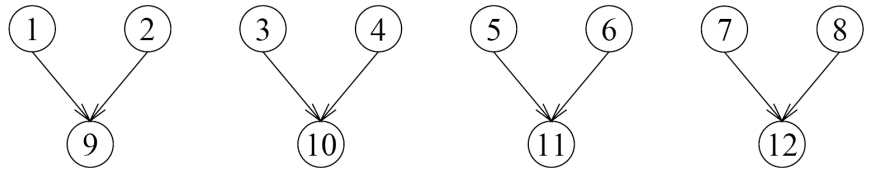



**Figure 3.16** The task-dependency graph of the decomposition shown in Figure 3.15.

**Figure 3.15** A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

# From Data Decompositions to Task Mappings: Owner Computes Rule

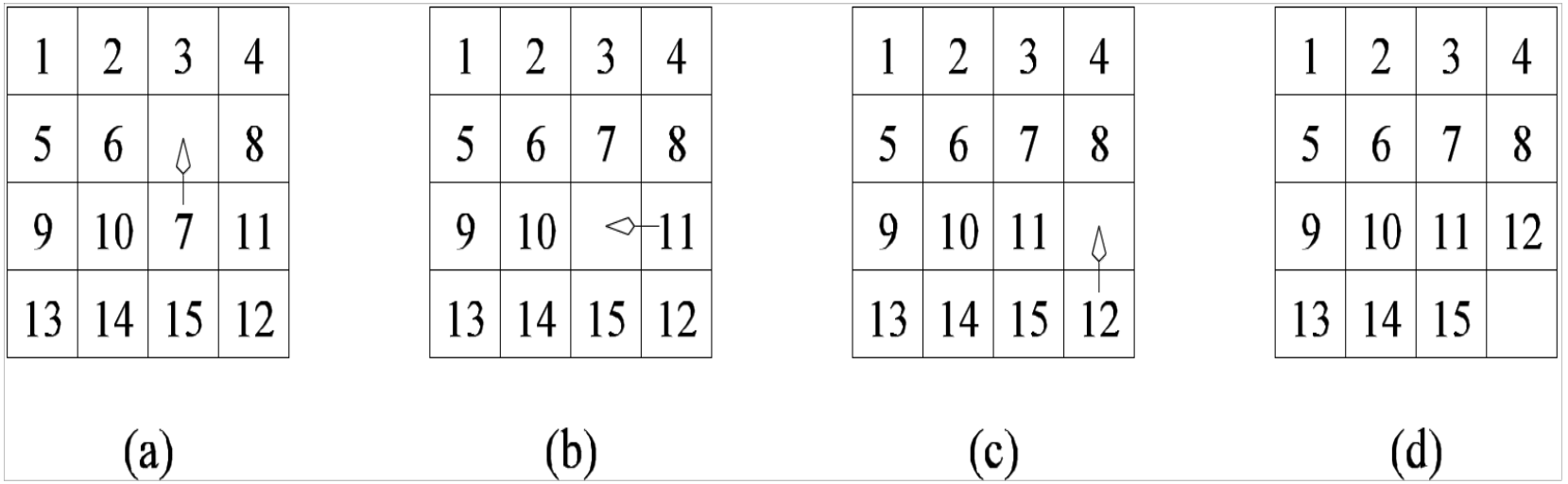
- **The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.**
- **In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.**
- **In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.**
- **Likewise for intermediate data decompositions**

# 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

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

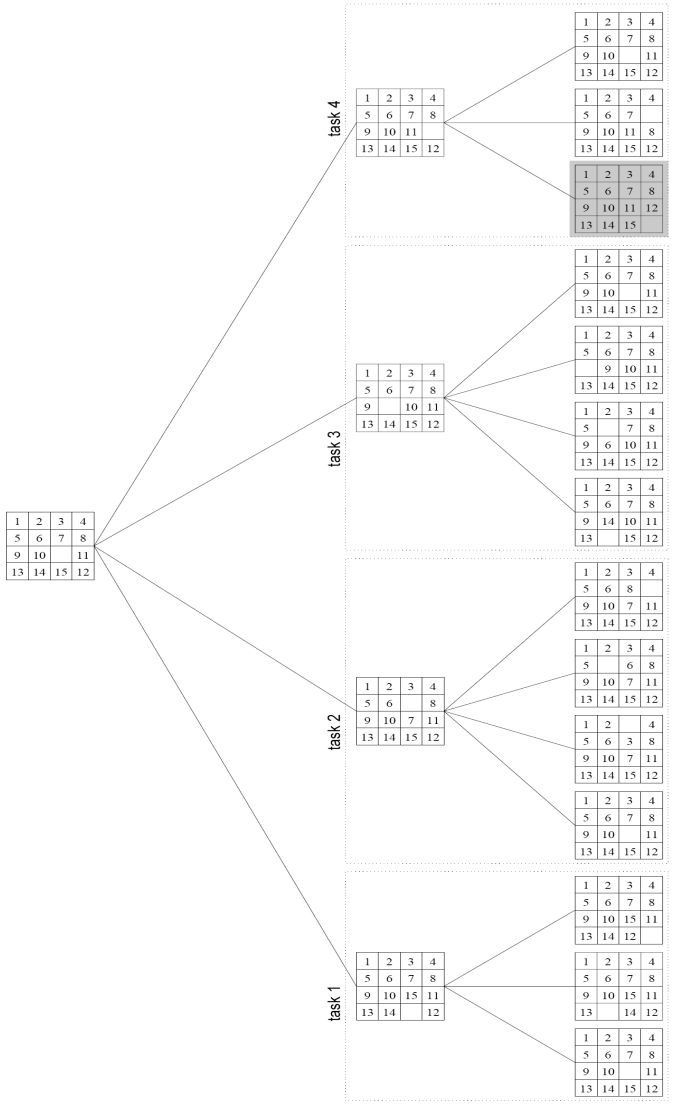
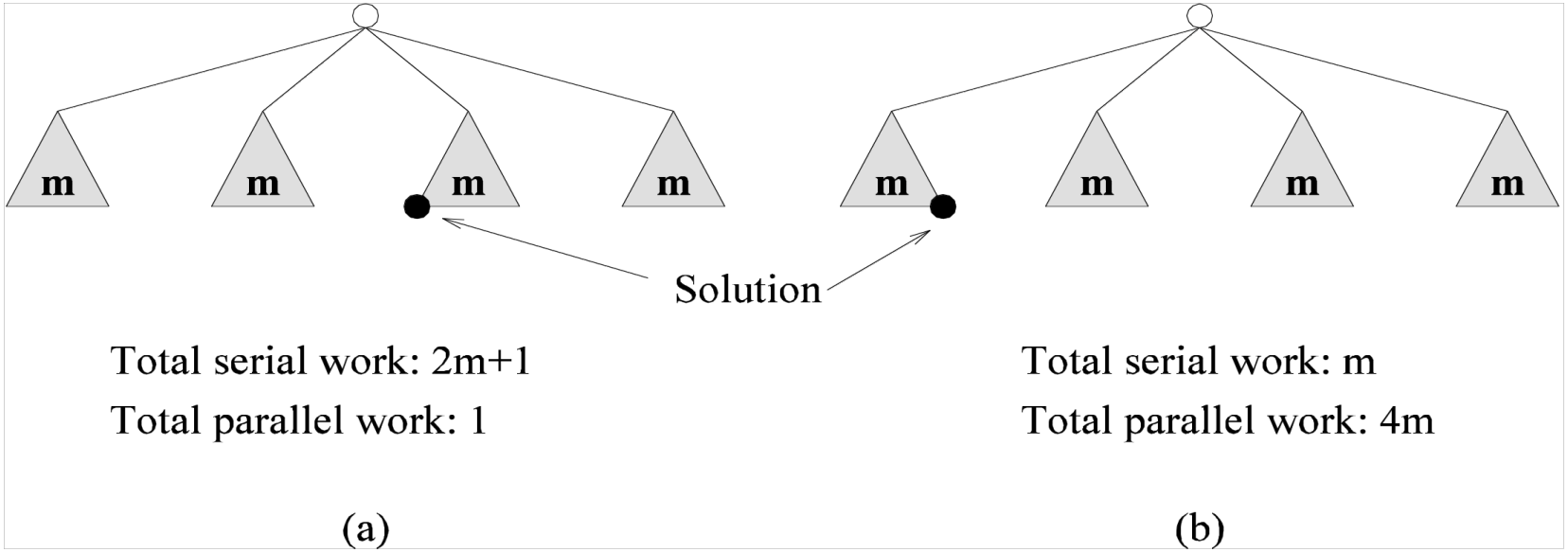



**Figure 3.18** The states generated by an instance of the 15-puzzle problem.

# Exploratory Decomposition: Anomalous Speedups

- In many instances of parallel exploratory decomposition, unfinished tasks can be terminated when the first solution is found
- This can result in "anomalous" super- or sub-linear speedups relative to serial execution.



# 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, 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.
- Parallel Discrete Event Simulation (Example 3.8) is a motivating example for optimistic approaches

# Hybrid Decompositions

**Often, a mix of decomposition techniques is necessary for decomposing a problem e.g.,**

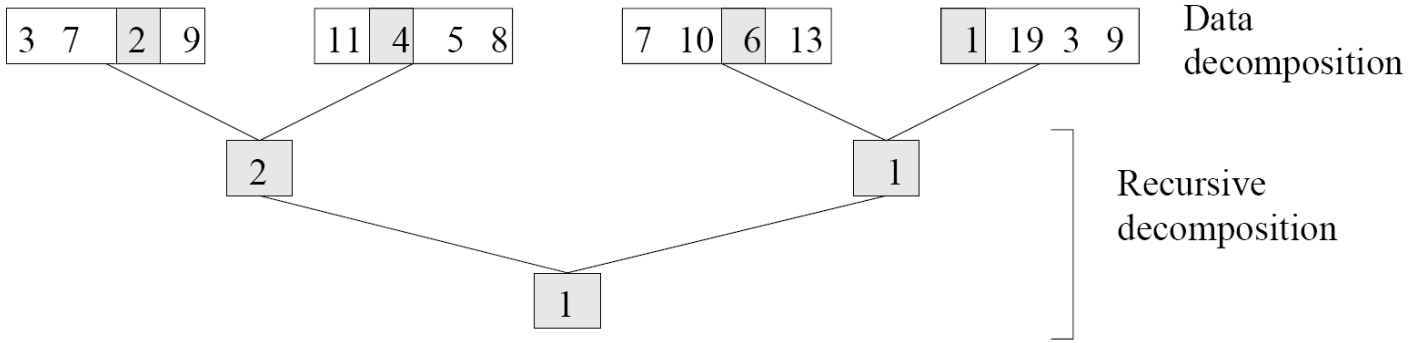



**Figure 3.21** Hybrid decomposition for finding the minimum of an array of size 16 using four tasks.

# Summary of Today's Lecture

- Tasks, Dependence Graphs, Scheduling Theory
- Data and Computation Decompositions

Reading List for Next Lecture (Jan 22nd)

- Sections 7.1, 7.2, 7.3, 7.4 of textbook
- Pages 4 - 17 (Sections 2.1 - 2.5) of Cilk Reference Manual
  —http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf