



# Introduction to Parallel Computing

George Karypis

Principles of Parallel Algorithm  
Design



# Outline

- Overview of some Serial Algorithms
- Parallel Algorithm vs Parallel Formulation
- Elements of a Parallel Algorithm/Formulation
- Common Decomposition Methods
  - concurrency extractor!
- Common Mapping Methods
  - parallel overhead reducer!



# Some Serial Algorithms

## Working Examples

- Dense Matrix-Matrix & Matrix-Vector Multiplication
- Sparse Matrix-Vector Multiplication
- Gaussian Elimination
- Floyd's All-pairs Shortest Path
- Quicksort
- Minimum/Maximum Finding
- Heuristic Search—15-puzzle problem



# Dense Matrix-Vector Multiplication

---

```
1.  procedure MAT_VECT ( $A, x, y$ )
2.  begin
3.    for  $i := 0$  to  $n - 1$  do
4.    begin
5.       $y[i] := 0$ ;
6.      for  $j := 0$  to  $n - 1$  do
7.         $y[i] := y[i] + A[i, j] \times x[j]$ ;
8.      endfor;
9.    end MAT_VECT
```

---

**Algorithm 8.1** A serial algorithm for multiplying an  $n \times n$  matrix  $A$  with an  $n \times 1$  vector  $x$  to yield an  $n \times 1$  product vector  $y$ .



# Dense Matrix-Matrix Multiplication

---

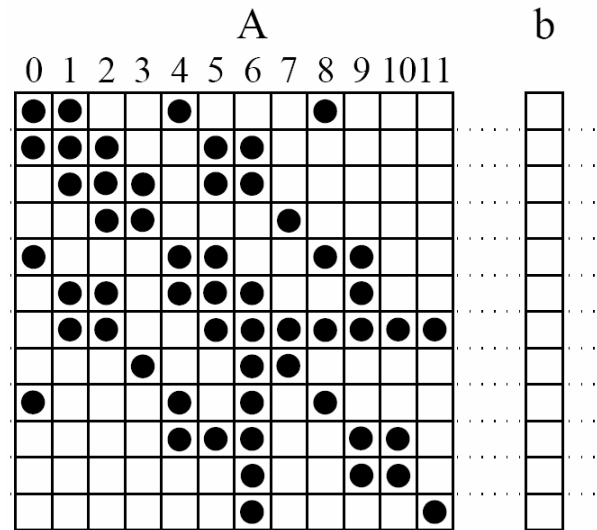
```
1.  procedure MAT_MULT ( $A, B, C$ )
2.  begin
3.    for  $i := 0$  to  $n - 1$  do
4.      for  $j := 0$  to  $n - 1$  do
5.        begin
6.           $C[i, j] := 0$ ;
7.          for  $k := 0$  to  $n - 1$  do
8.             $C[i, j] := C[i, j] + A[i, k] \times B[k, j]$ ;
9.          endfor;
10. end MAT_MULT
```

---

**Algorithm 8.2** The conventional serial algorithm for multiplication of two  $n \times n$  matrices.

# Sparse Matrix-Vector Multiplication

$$y = Ab$$



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

# Gaussian Elimination

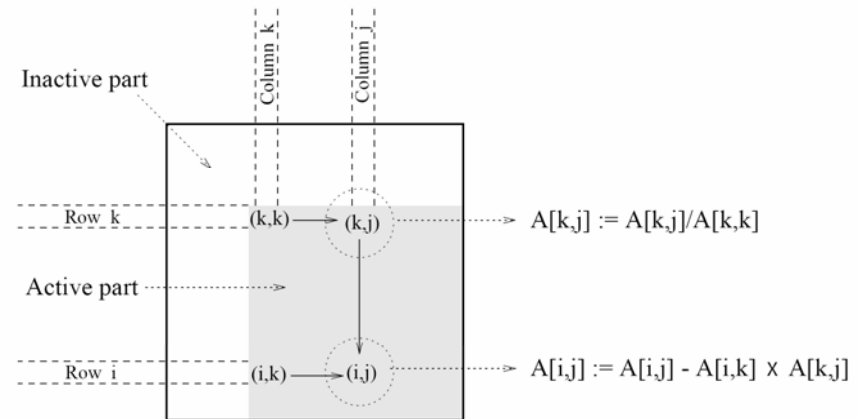
$$\begin{array}{cccccc}
 a_{0,0}x_0 & + & a_{0,1}x_1 & + & \cdots & + & a_{0,n-1}x_{n-1} & = & b_0, \\
 a_{1,0}x_0 & + & a_{1,1}x_1 & + & \cdots & + & a_{1,n-1}x_{n-1} & = & b_1, \\
 \vdots & & \vdots & & & & \vdots & & \vdots \\
 a_{n-1,0}x_0 & + & a_{n-1,1}x_1 & + & \cdots & + & a_{n-1,n-1}x_{n-1} & = & b_{n-1}.
 \end{array}$$

```

1. procedure GAUSSIAN_ELIMINATION (A, b, y)
2. begin
3.   for k := 0 to n - 1 do           /* Outer loop */
4.     begin
5.       for j := k + 1 to n - 1 do
6.         A[k, j] := A[k, j]/A[k, k]; /* Division step */
7.         y[k] := b[k]/A[k, k];
8.         A[k, k] := 1;
9.         for i := k + 1 to n - 1 do
10.          begin
11.            for j := k + 1 to n - 1 do
12.              A[i, j] := A[i, j] - A[i, k] × A[k, j]; /* Elimination step */
13.              b[i] := b[i] - A[i, k] × y[k];
14.              A[i, k] := 0;
15.            endfor; /* Line 9 */
16.          endfor; /* Line 3 */
17.        end GAUSSIAN_ELIMINATION

```

**Algorithm 8.4** A serial Gaussian elimination algorithm that converts the system of linear equations  $Ax = b$  to a unit upper-triangular system  $Ux = y$ . The matrix  $U$  occupies the upper-triangular locations of  $A$ . This algorithm assumes that  $A[k, k] \neq 0$  when it is used as a divisor on lines 6 and 7.



**Figure 3.28** A typical computation in Gaussian elimination and the active part of the coefficient matrix during the  $k$ th iteration of the outer loop.

# Floyd's All-Pairs Shortest Path

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min \{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \} & \text{if } k \geq 1 \end{cases}$$

---

```
1. procedure FLOYD_ALL_PAIRS_SP(A)
2. begin
3.    $D^{(0)} = A$ ;
4.   for  $k := 1$  to  $n$  do
5.     for  $i := 1$  to  $n$  do
6.       for  $j := 1$  to  $n$  do
7.          $d_{i,j}^{(k)} := \min (d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$ ;
8.   end FLOYD_ALL_PAIRS_SP
```

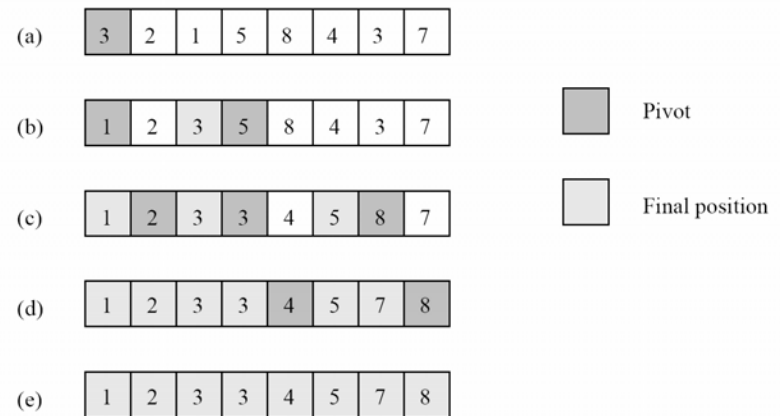
---

**Algorithm 10.3** Floyd's all-pairs shortest paths algorithm. This program computes the all-pairs shortest paths of the graph  $G = (V, E)$  with adjacency matrix  $A$ .



# Quicksort

```
1. procedure QUICKSORT ( $A, q, r$ )
2. begin
3.   if  $q < r$  then
4.     begin
5.        $x := A[q]$ ;
6.        $s := q$ ;
7.       for  $i := q + 1$  to  $r$  do
8.         if  $A[i] \leq x$  then
9.           begin
10.             $s := s + 1$ ;
11.            swap( $A[s], A[i]$ );
12.          end if
13.        swap( $A[q], A[s]$ );
14.        QUICKSORT ( $A, q, s$ );
15.        QUICKSORT ( $A, s + 1, r$ );
16.      end if
17.    end QUICKSORT
```



**Figure 9.15** Example of the quicksort algorithm sorting a sequence of size  $n = 8$ .

**Algorithm 9.5** The sequential quicksort algorithm.



# Minimum Finding

---

```
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
```

---

**Algorithm 3.1** A serial program for finding the minimum in an array of numbers  $A$  of length  $n$ .

# 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)

**Figure 3.17** A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.



# Parallel Algorithm vs Parallel Formulation

- *Parallel Formulation*

- Refers to a *parallelization* of a serial algorithm.

- *Parallel Algorithm*

- May represent an entirely different algorithm than the one used serially.

- We primarily focus on “*Parallel Formulations*”

- Our goal today is to primarily discuss how to develop such parallel formulations.
- Of course, there will always be examples of “parallel algorithms” that were not derived from serial algorithms.



# Elements of a Parallel Algorithm/Formulation

- Pieces of work that can be done concurrently
  - tasks
- Mapping of the tasks onto multiple processors
  - processes vs processors
- Distribution of input/output & intermediate data across the different processors
- Management the access of shared data
  - either input or intermediate
- Synchronization of the processors at various points of the parallel execution

Holy Grail:

Maximize concurrency and reduce overheads due to parallelization!

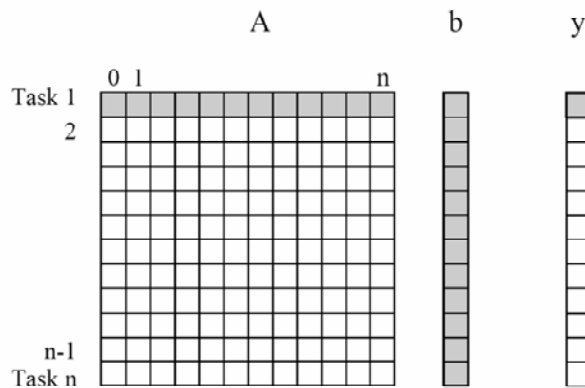
Maximize potential speedup!



# Finding Concurrent Pieces of Work

- Decomposition:
  - The process of dividing the computation into smaller pieces of work i.e., *tasks*
- Tasks are programmer defined and are considered to be indivisible

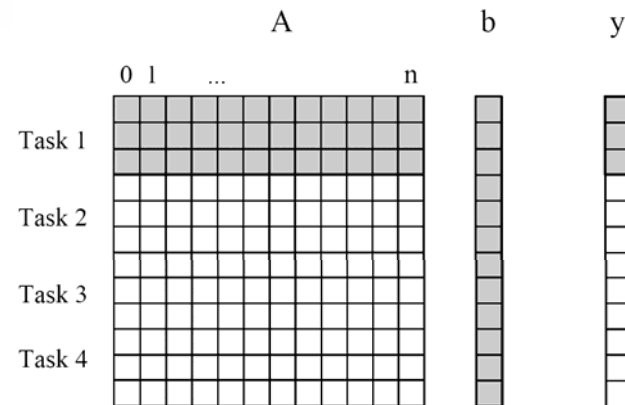
# Example: Dense Matrix-Vector Multiplication



**Figure 3.1** Decomposition of dense matrix-vector multiplication into  $n$  tasks, where  $n$  is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.

Tasks can be of different size.

- *granularity of a task*



**Figure 3.4** Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.



# Example: Query Processing

| 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 |

Query: MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")



# Example: Query Processing

## ■ Finding concurrent tasks...

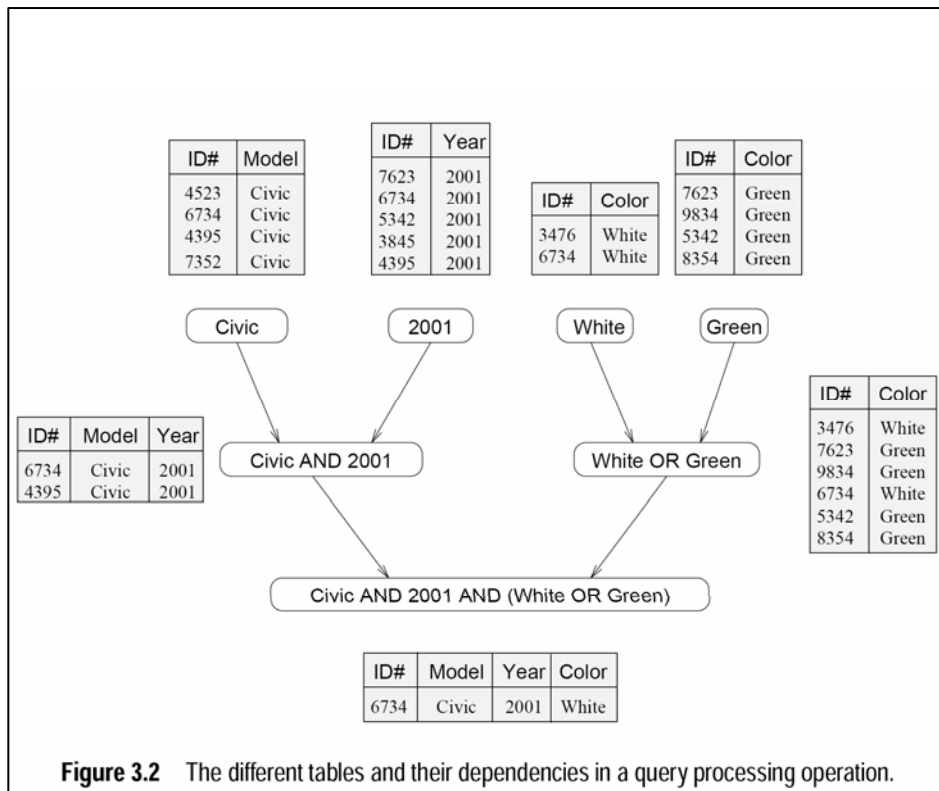


Figure 3.2 The different tables and their dependencies in a query processing operation.

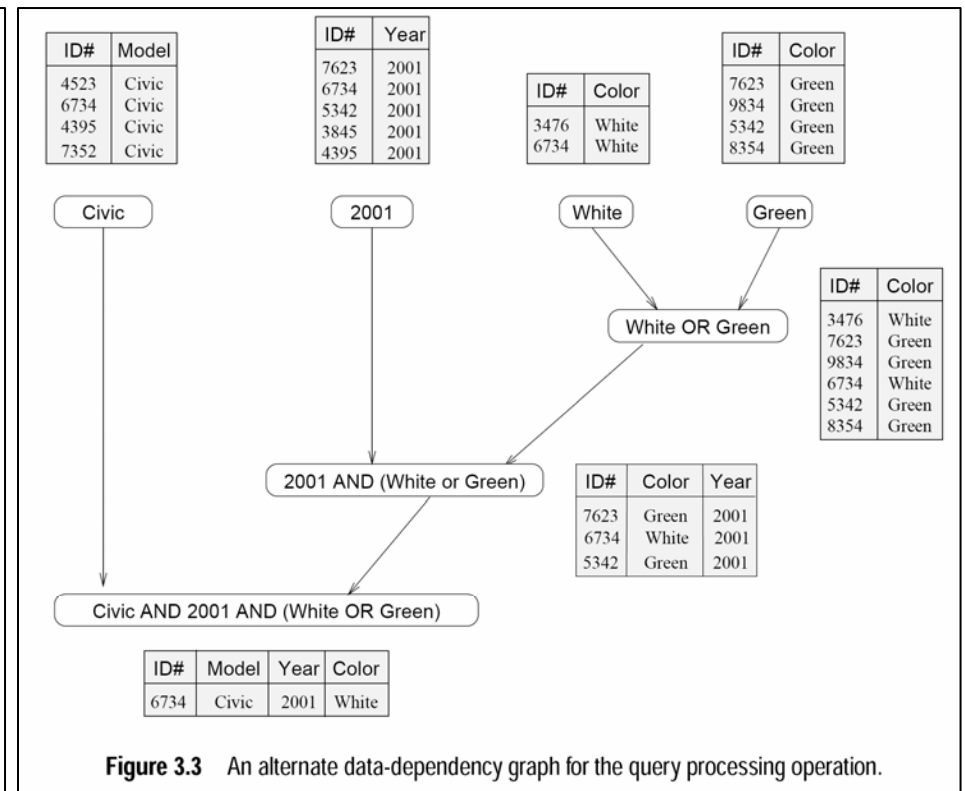
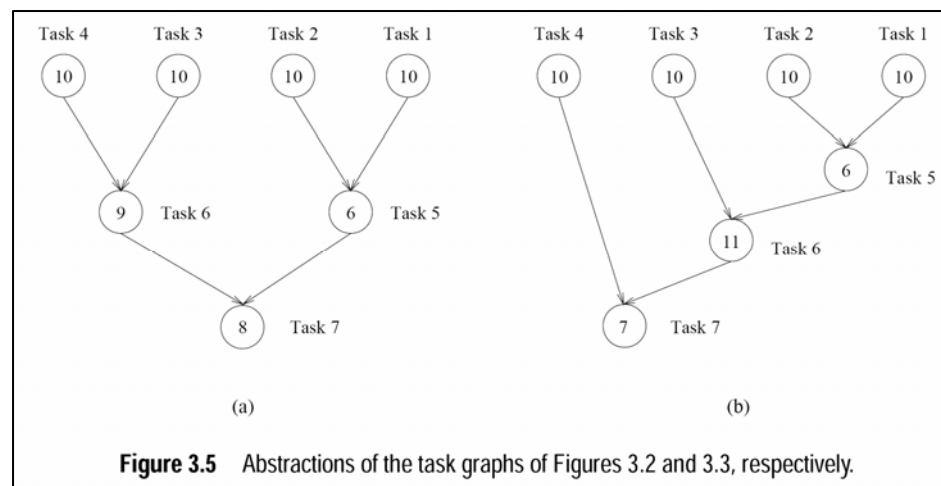


Figure 3.3 An alternate data-dependency graph for the query processing operation.

# Task-Dependency Graph

- In most cases, there are dependencies between the different tasks
  - certain task(s) can only start once some other task(s) have finished
    - e.g., producer-consumer relationships
- These dependencies are represented using a DAG called *task-dependency graph*



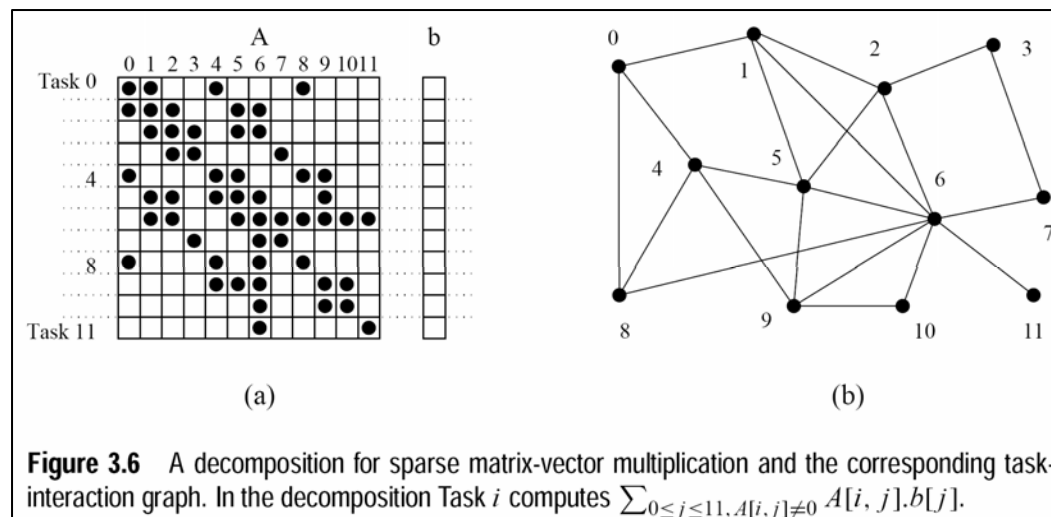


# Task-Dependency Graph (cont)

- Key Concepts Derived from the Task-Dependency Graph
  - Degree of Concurrency
    - The number of tasks that can be concurrently executed
      - we usually care about the *average* degree of concurrency
  - Critical Path
    - The longest vertex-weighted path in the graph
      - The weights represent task size
  - Task granularity affects both of the above characteristics

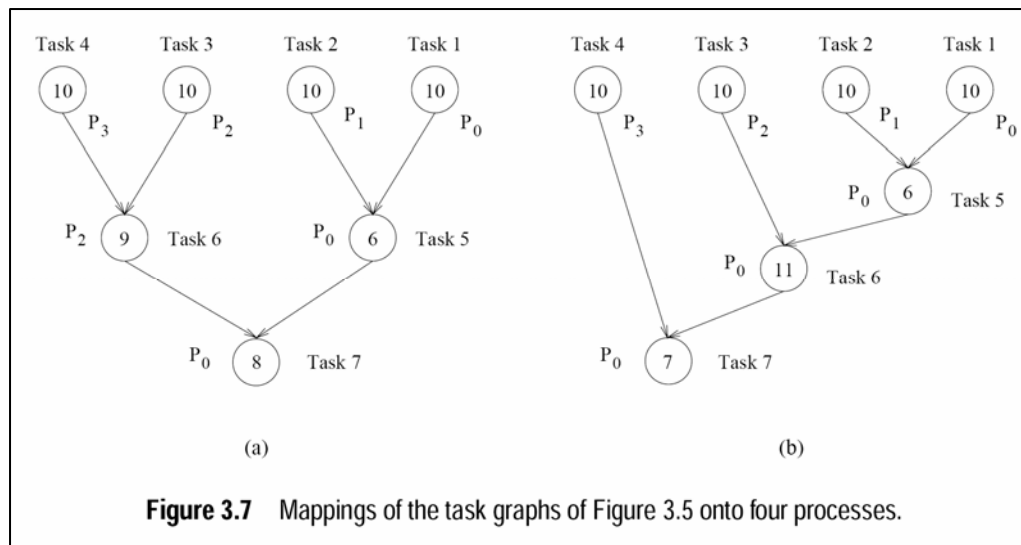
# Task-Interaction Graph

- Captures the pattern of interaction between tasks
  - This graph usually contains the task-dependency graph as a *subgraph*
    - i.e., there may be interactions between tasks even if there are no dependencies
      - these interactions usually occur due to accesses on shared data



# Task Dependency/Interaction Graphs

- These graphs are important in developing effectively mapping the tasks onto the different processors
  - Maximize concurrency and minimize overheads

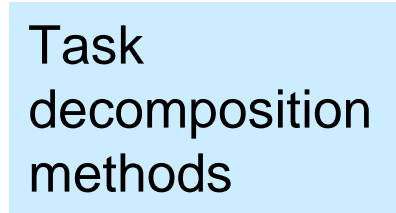


- More on this later...



# Common Decomposition Methods

- Data Decomposition
- Recursive Decomposition
- Exploratory Decomposition
- Speculative Decomposition
- Hybrid Decomposition



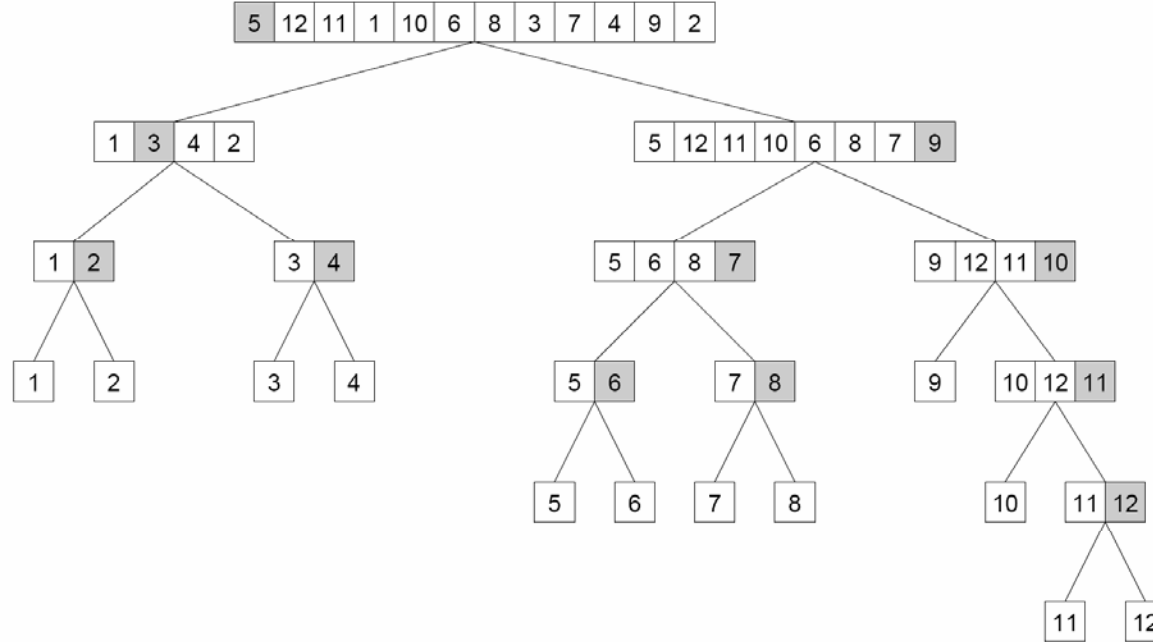
Task  
decomposition  
methods



# Recursive Decomposition

- Suitable for problems that can be solved using the divide-and-conquer paradigm
- Each of the *subproblems* generated by the *divide* step becomes a task

# Example: Quicksort



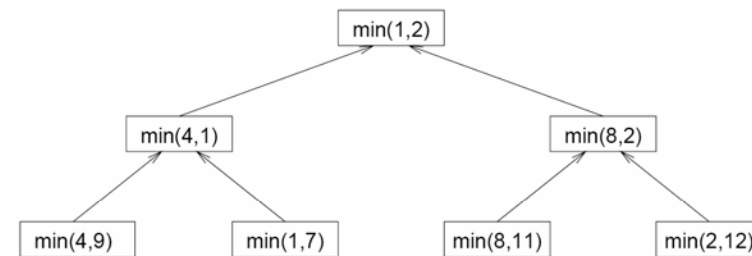
**Figure 3.8** The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.



# Example: Finding the Minimum

- Note that we can obtain divide-and-conquer algorithms for problems that are traditionally solved using non-divide-and-conquer approaches

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



**Figure 3.9** The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.

**Algorithm 3.2** A recursive program for finding the minimum in an array of numbers *A* of length *n*.



# Recursive Decomposition

- How good are the decompositions that it produces?
  - average concurrency?
  - critical path?
- How do the quicksort and min-finding decompositions measure-up?



# Data Decomposition

- Used to derive concurrency for problems that operate on large amounts of data
- The idea is to derive the tasks by focusing on the multiplicity of data
- Data decomposition is often performed in two steps
  - Step 1: Partition the data
  - Step 2: Induce a computational partitioning from the data partitioning
- Which data should we partition?
  - Input/Output/Intermediate?
    - Well... all of the above—leading to different data decomposition methods
- How do induce a computational partitioning?
  - Owner-computes rule

# Example: Matrix-Matrix Multiplication

## ■ Partitioning the output data

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

(a)

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

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

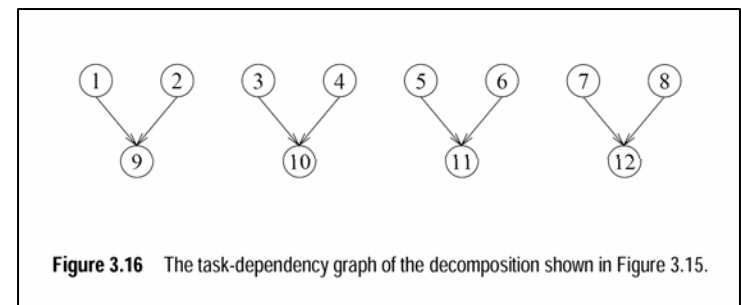
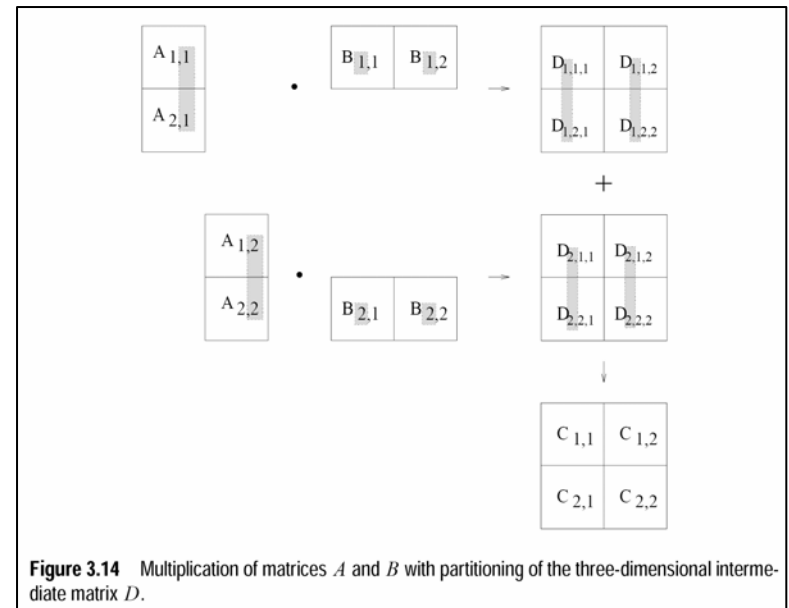
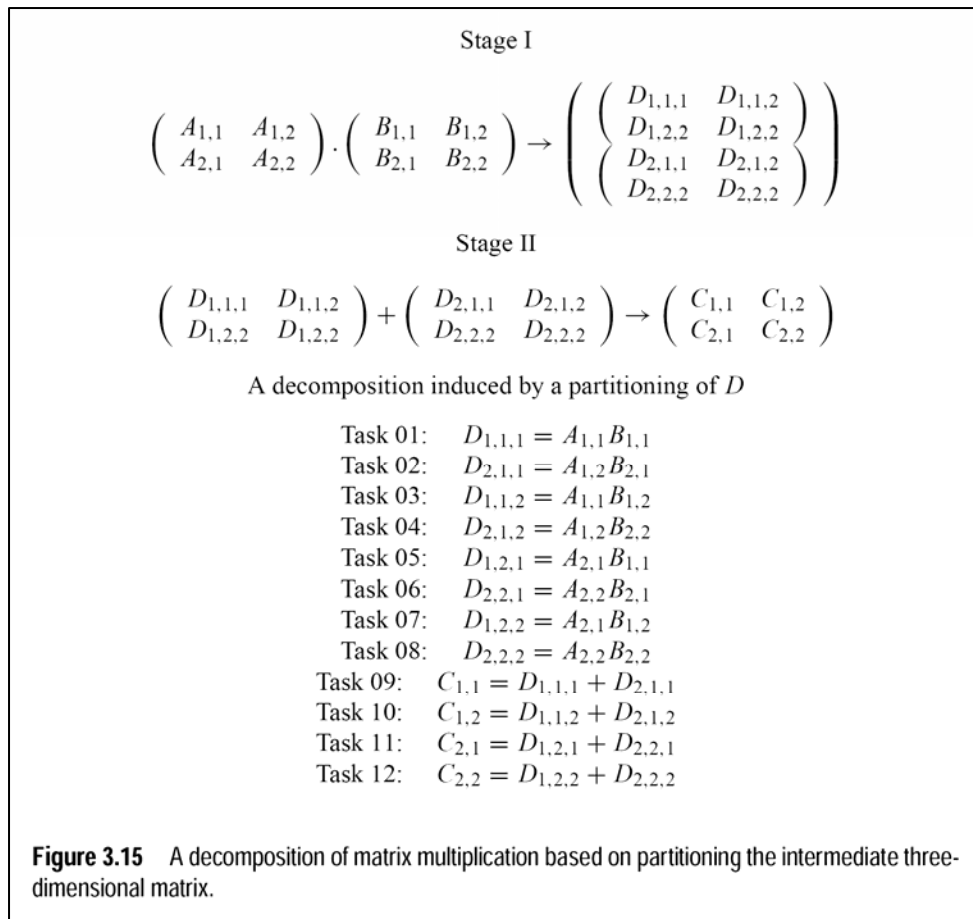
$$\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

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

(b)

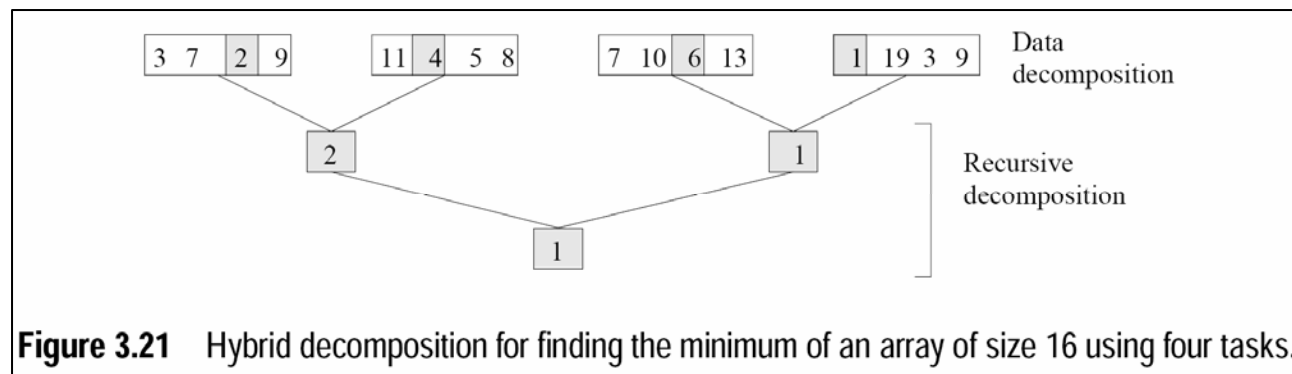
# Example: Matrix-Matrix Multiplication

## ■ Partitioning the intermediate data



# Data Decomposition

- Is the most widely-used decomposition technique
  - after all parallel processing is often applied to problems that have a lot of data
  - splitting the work based on this data is the natural way to extract high-degree of concurrency
- It is used by itself or in conjunction with other decomposition methods
  - Hybrid decomposition



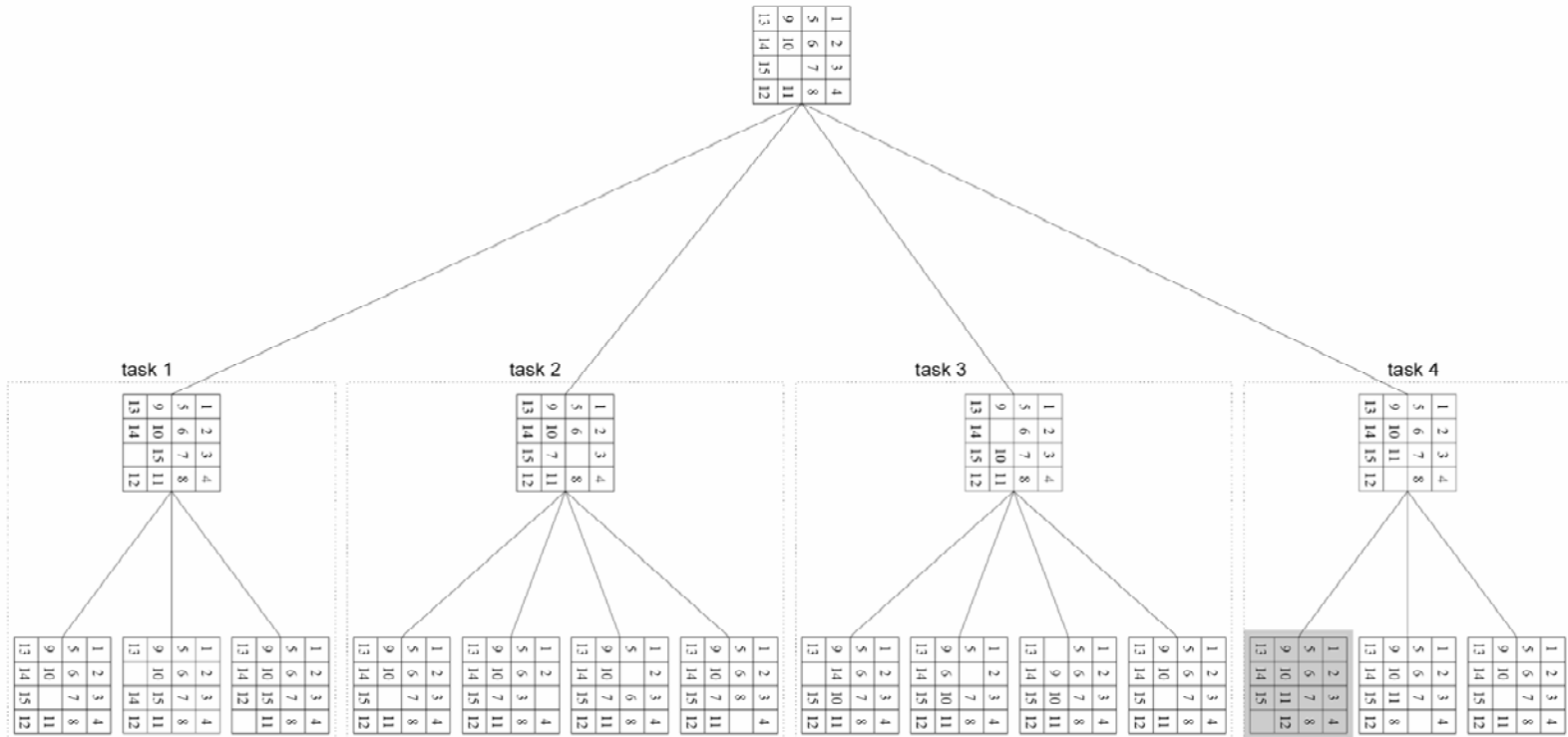
**Figure 3.21** Hybrid decomposition for finding the minimum of an array of size 16 using four tasks.



# Exploratory Decomposition

- Used to decompose computations that correspond to a search of a space of solutions

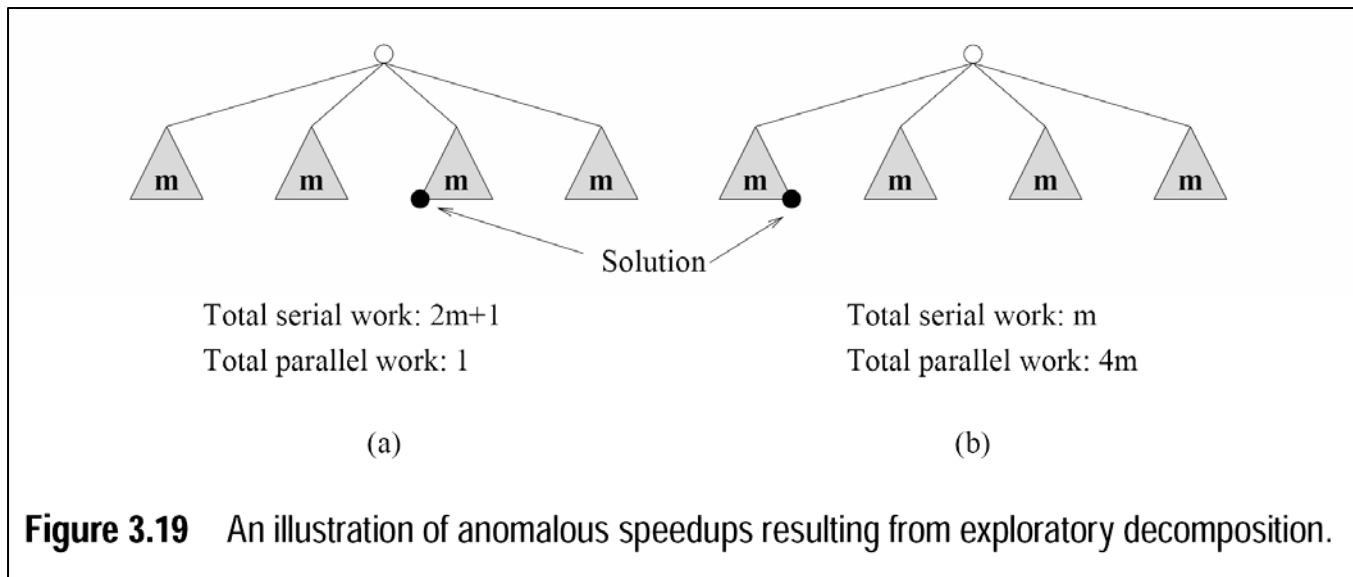
# Example: 15-puzzle Problem





# Exploratory Decomposition

- It is not as general purpose
- It can result in speedup anomalies
  - *engineered* slow-down or superlinear speedup

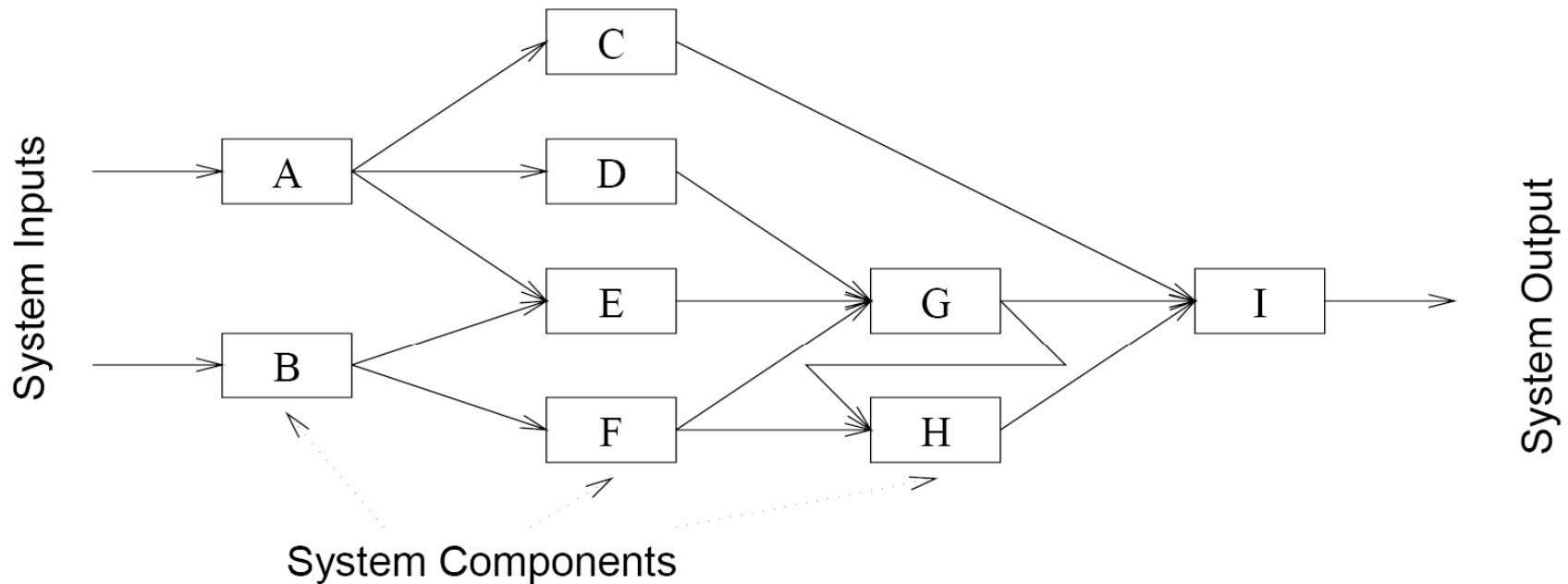




# Speculative Decomposition

- Used to extract concurrency in problems in which the *next step* is one of many possible actions that can only be determined when the current tasks finishes
- This decomposition assumes a certain *outcome* of the currently executed task and *executes* some of the next steps
  - Just like speculative execution at the microprocessor level

# Example: Discrete Event Simulation



**Figure 3.20** A simple network for discrete event simulation.



# Speculative Execution

- If predictions are wrong...
  - work is wasted
  - work may need to be *undone*
    - state-restoring overhead
      - memory/computations
- However, it may be the only way to extract concurrency!

# Mapping the Tasks

- Why do we care about task mapping?
  - Can I just randomly assign them to the available processors?
- Proper mapping is critical as it needs to minimize the parallel processing overheads
  - If  $T_p$  is the parallel runtime on  $p$  processors and  $T_s$  is the serial runtime, then the *total overhead*  $T_o$  is  $p * T_p - T_s$ 
    - The work done by the parallel system beyond that required by the serial system

- Overhead sources:

they can be at odds with each other

- Load imbalance
- Inter-process communication
  - coordination/synchronization/data-sharing

remember the holy grail...

# Why Mapping can be Complicated?

- Proper mapping needs to take into account the task-dependency and interaction graphs

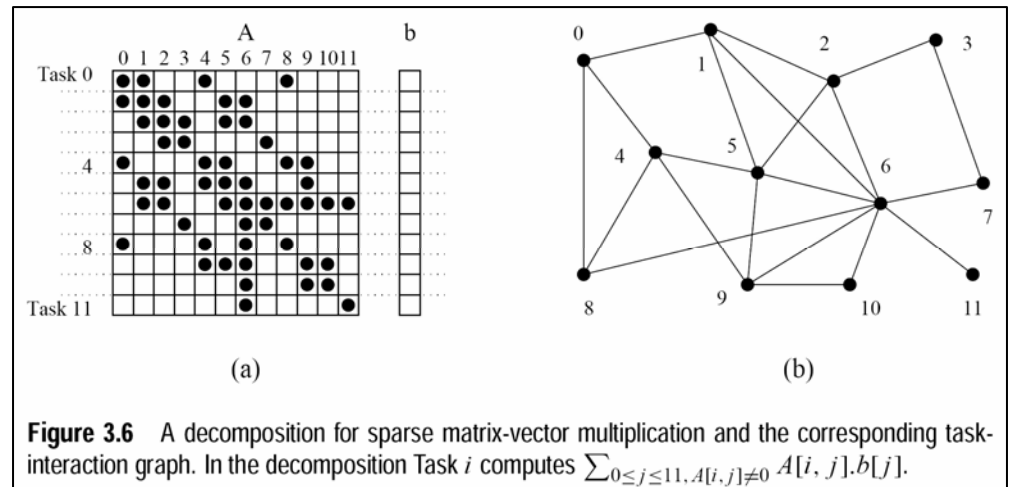
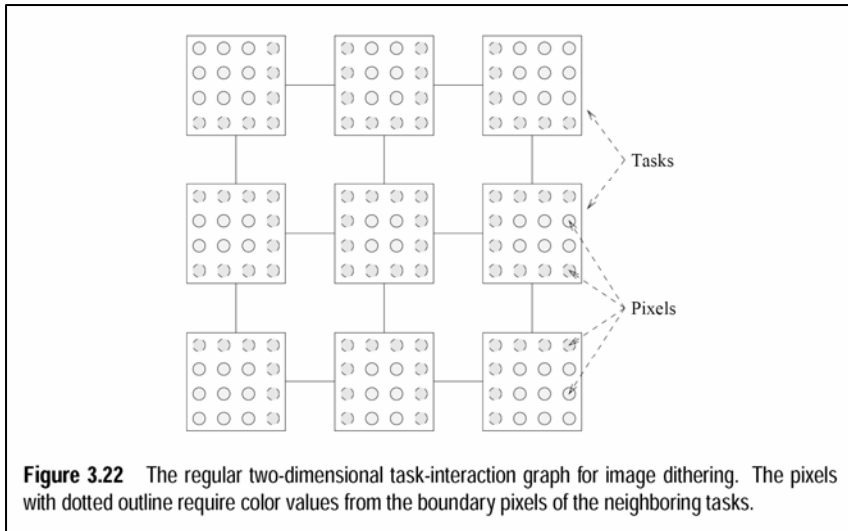
- Are the tasks available a priori?
  - Static vs dynamic task generation
- How about their computational requirements?
  - Are they uniform or non-uniform?
  - Do we know them a priori?
- How much data is associated with each task?
- How about the interaction patterns between the tasks?
  - Are they static or dynamic?
  - Do we know them a priori?
  - Are they data instance dependent?
  - Are they regular or irregular?
  - Are they read-only or read-write?

Task  
dependency  
graph

Task  
interaction  
graph

- Depending on the above characteristics different mapping techniques are required of different complexity and cost

# Example: Simple & Complex Task Interaction

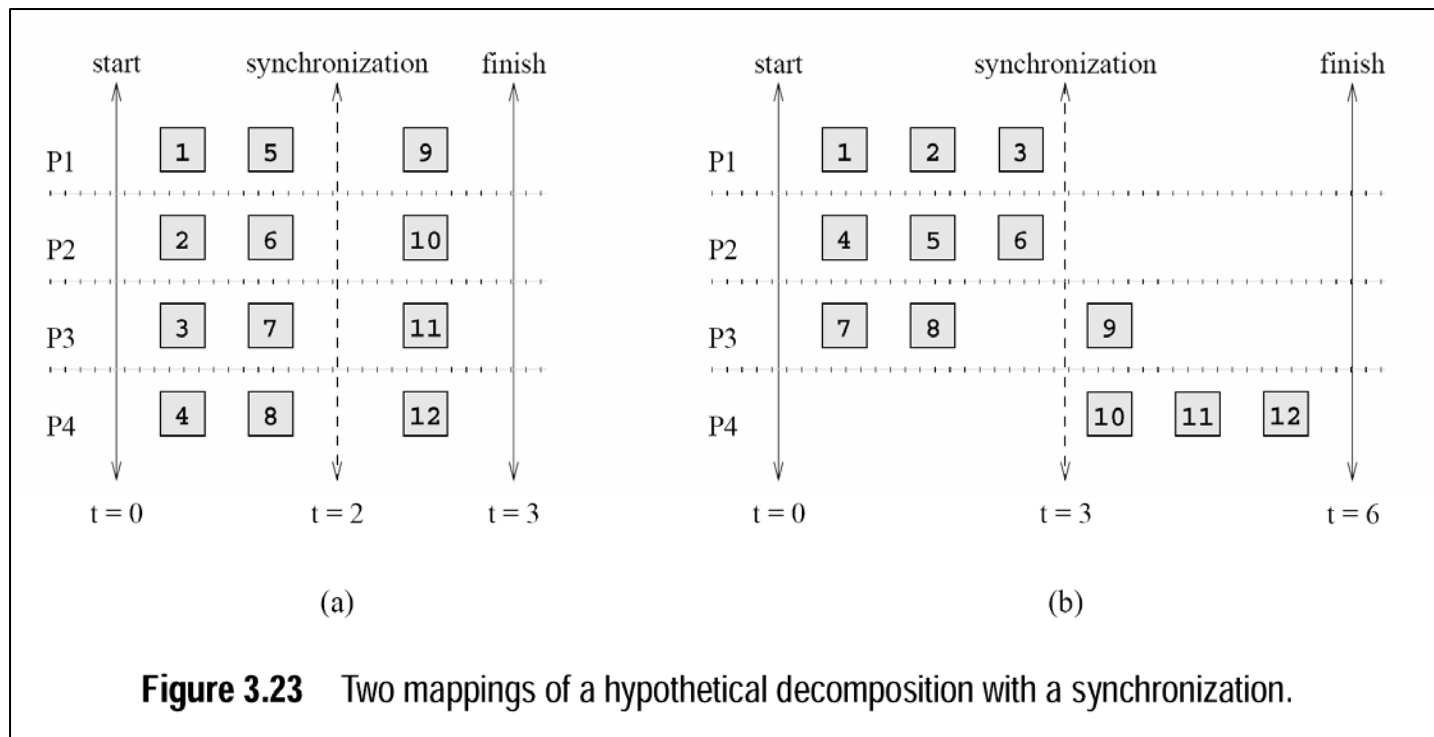


# Mapping Techniques for Load Balancing

## ■ Be aware...

- The assignment of tasks whose aggregate computational requirements are the same does not automatically ensure load balance.

Each processor is assigned three tasks but (a) is better than (b)!







# Load Balancing Techniques

## ■ Static

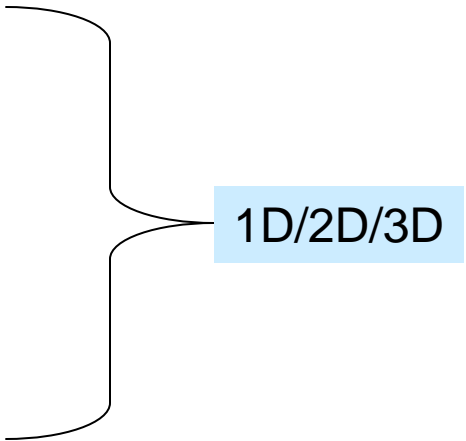
- The tasks are distributed among the processors prior to the execution
- Applicable for tasks that are
  - generated statically
  - known and/or uniform computational requirements

## ■ Dynamic

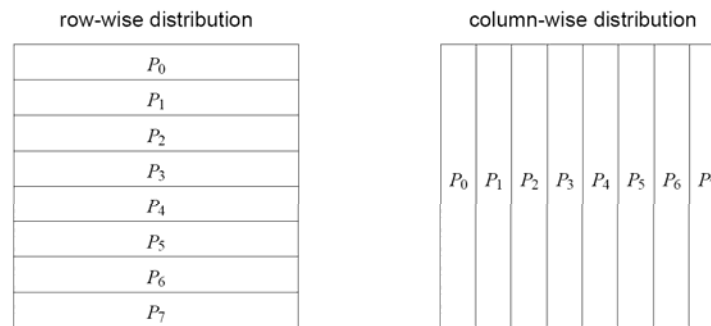
- The tasks are distributed among the processors during the execution of the algorithm
  - i.e., tasks & data are migrated
- Applicable for tasks that are
  - generated dynamically
  - unknown computational requirements



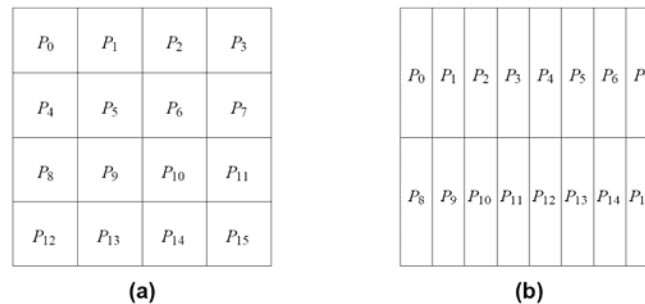
# Static Mapping—Array Distribution

- Suitable for algorithms that
    - use data decomposition
    - their underlying input/output/intermediate data are in the form of arrays
  - Block Distribution
  - Cyclic Distribution
  - Block-Cyclic Distribution
  - Randomized Block Distributions
- 
- 1D/2D/3D

# Examples: Block Distributions

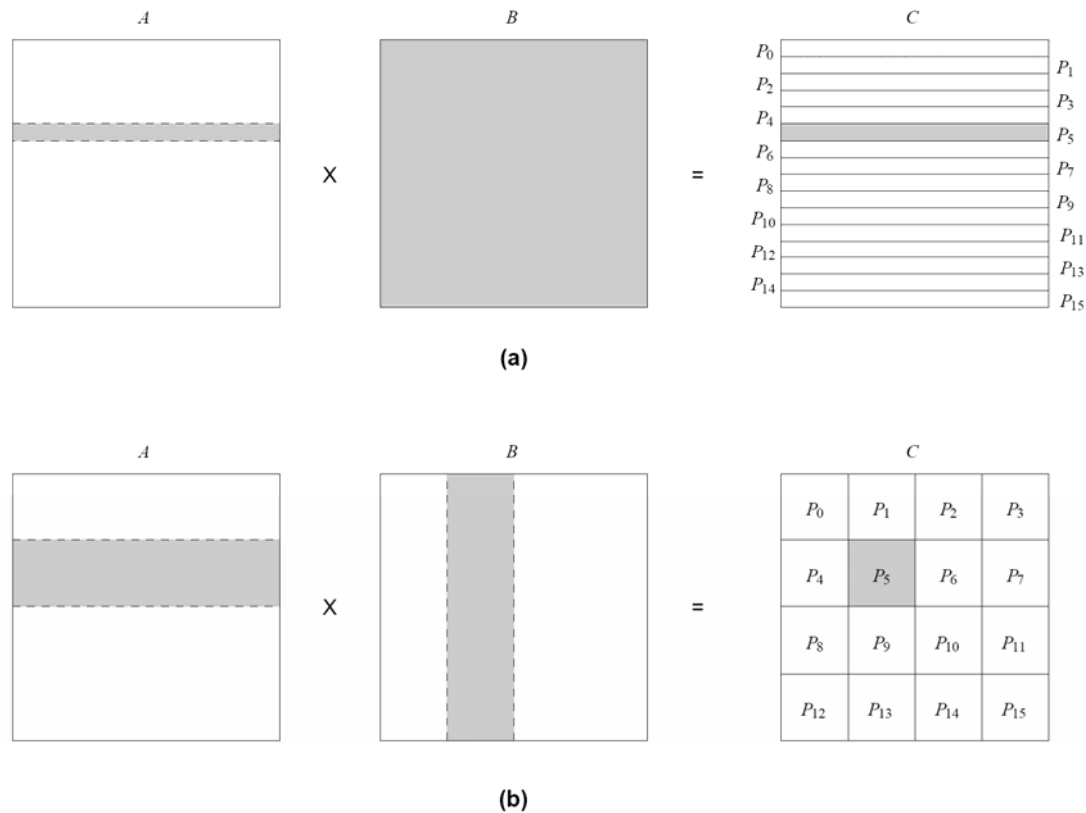


**Figure 3.24** Examples of one-dimensional partitioning of an array among eight processes.



**Figure 3.25** Examples of two-dimensional distributions of an array, (a) on a  $4 \times 4$  process grid, and (b) on a  $2 \times 8$  process grid.

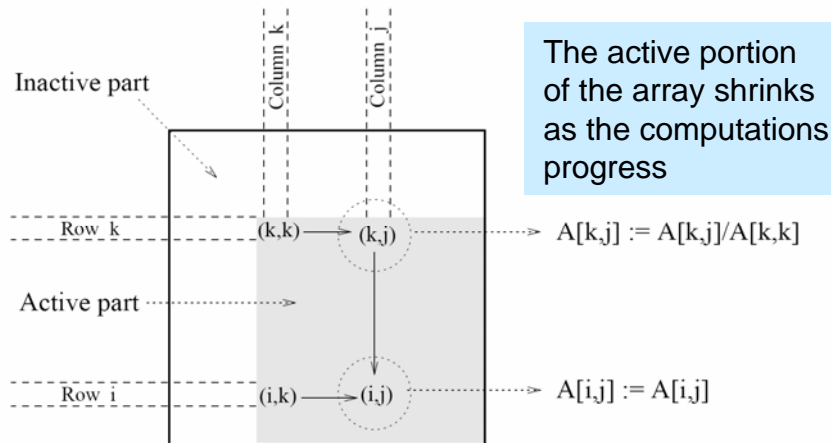
# Examples: Block Distributions



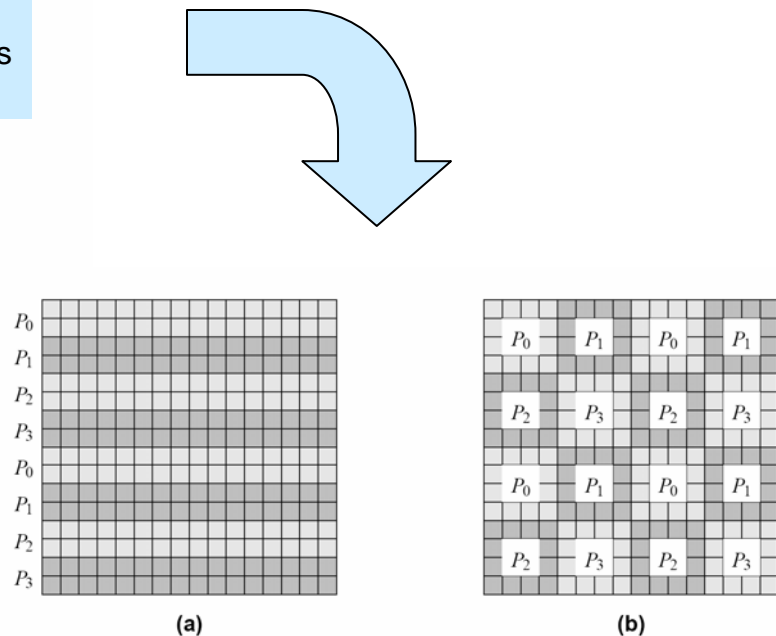
**Figure 3.26** Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices  $A$  and  $B$  are required by the process that computes the shaded portion of the output matrix  $C$ .

# Example: Block-Cyclic Distributions

## ■ Gaussian Elimination



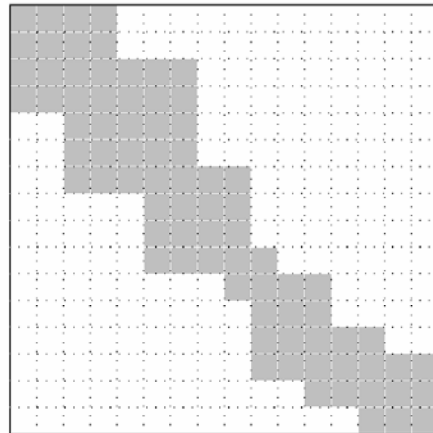
The active portion of the array shrinks as the computations progress



**Figure 3.30** Examples of one- and two-dimensional block-cyclic distributions among four processes. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wraparound fashion. (b) The matrix is blocked into 16 blocks each of size  $4 \times 4$ , and it is mapped onto a  $2 \times 2$  grid of processes in a wraparound fashion.

# Random Block Distributions

- Sometimes the computations are performed only at certain portions of an array
  - sparse matrix-matrix multiplication



(a)

|          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|
| $P_0$    | $P_1$    | $P_2$    | $P_3$    | $P_0$    | $P_1$    | $P_2$    | $P_3$    |
| $P_4$    | $P_5$    | $P_6$    | $P_7$    | $P_4$    | $P_5$    | $P_6$    | $P_7$    |
| $P_8$    | $P_9$    | $P_{10}$ | $P_{11}$ | $P_8$    | $P_9$    | $P_{10}$ | $P_{11}$ |
| $P_{12}$ | $P_{13}$ | $P_{14}$ | $P_{15}$ | $P_{12}$ | $P_{13}$ | $P_{14}$ | $P_{15}$ |
| $P_0$    | $P_1$    | $P_2$    | $P_3$    | $P_0$    | $P_1$    | $P_2$    | $P_3$    |
| $P_4$    | $P_5$    | $P_6$    | $P_7$    | $P_4$    | $P_5$    | $P_6$    | $P_7$    |
| $P_8$    | $P_9$    | $P_{10}$ | $P_{11}$ | $P_8$    | $P_9$    | $P_{10}$ | $P_{11}$ |
| $P_{12}$ | $P_{13}$ | $P_{14}$ | $P_{15}$ | $P_{12}$ | $P_{13}$ | $P_{14}$ | $P_{15}$ |

(b)

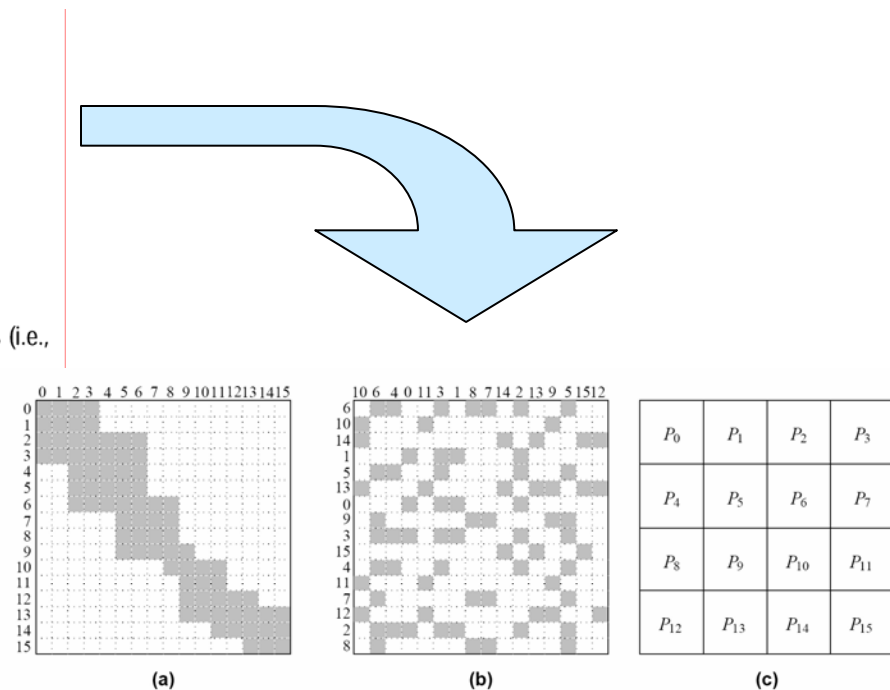
**Figure 3.31** Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.

# Random Block Distributions

- Better load balance can be achieved via a random block distribution

$V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$   
 $\text{random}(V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]$   
 $\text{mapping} = \begin{array}{cccc} 8 & 2 & 6 & 0 & 3 & 7 & 11 & 1 & 9 & 5 & 4 & 10 \\ \hline & \underbrace{\hspace{1.5em}} & \underbrace{\hspace{1.5em}} & \underbrace{\hspace{1.5em}} & \underbrace{\hspace{1.5em}} & & & & & & & \\ & P_0 & P_1 & P_2 & P_3 & & & & & & & \end{array}$

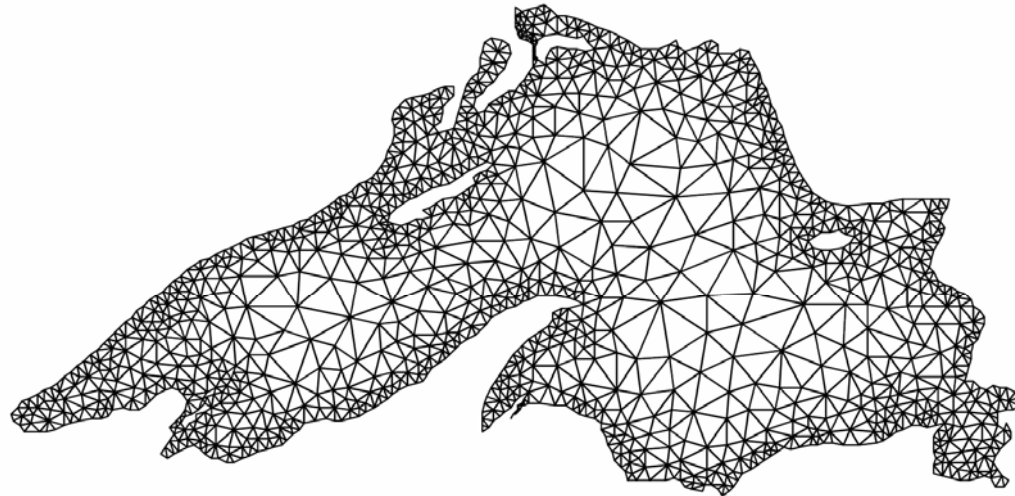
**Figure 3.32** A one-dimensional randomized block mapping of 12 blocks onto four process (i.e.,  $\alpha = 3$ ).



**Figure 3.33** Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c).

# Graph Partitioning

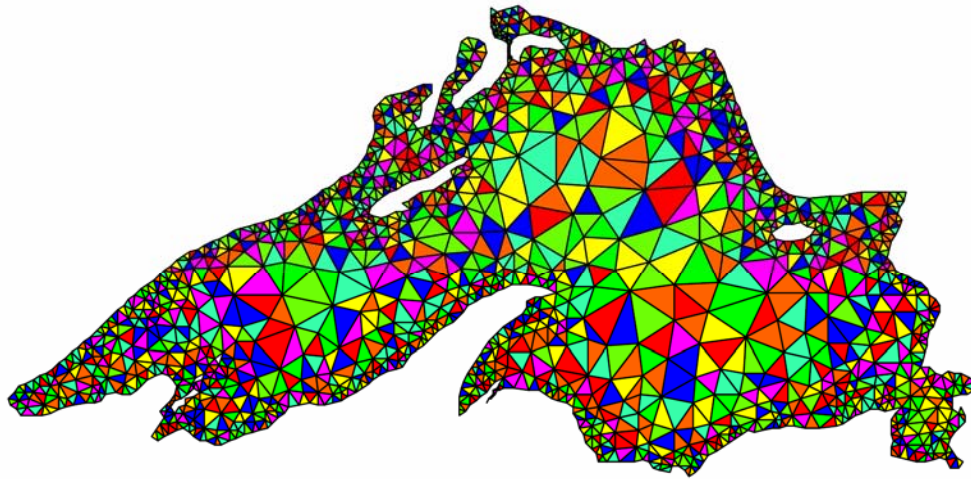
- A mapping can be achieved by directly partitioning the task interaction graph.
  - EG: Finite element mesh-based computations



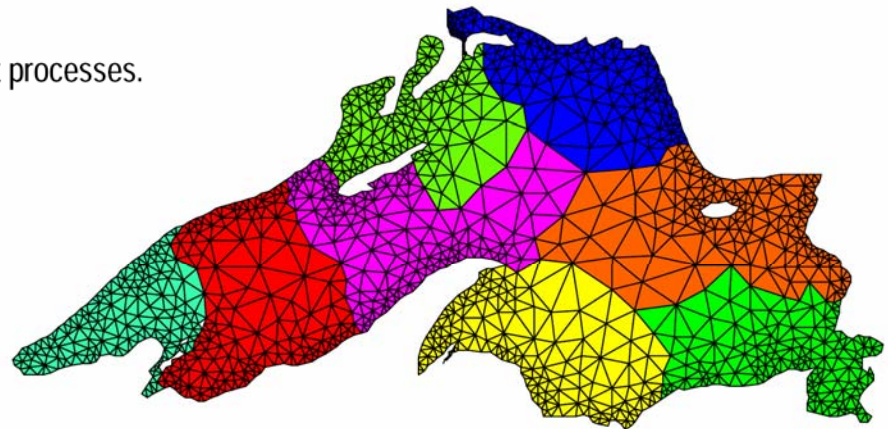
**Figure 3.34** A mesh used to model Lake Superior.



# Directly partitioning this graph



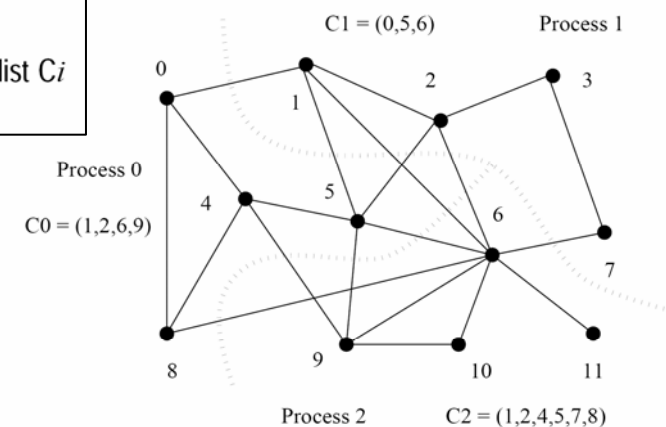
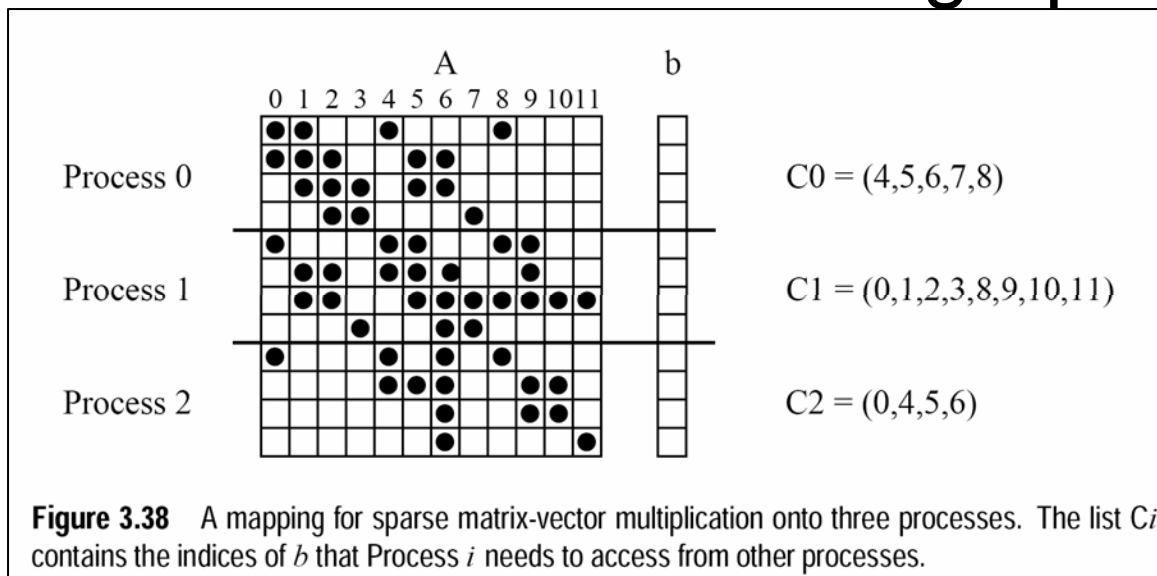
**Figure 3.35** A random distribution of the mesh elements to eight processes.



**Figure 3.36** A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm.

# Example: Sparse Matrix-Vector

## ■ Another *instance* of graph partitioning



**Figure 3.39** Reducing interaction overhead in sparse matrix-vector multiplication by partitioning the task-interaction graph.



# Dynamic Load Balancing Schemes

- There is a huge body of research
  - Centralized Schemes
    - A certain processors is responsible for giving out work
      - master-slave paradigm
    - Issue:
      - task granularity
  - Distributed Schemes
    - Work can be transferred between any pairs of processors.
    - Issues:
      - How do the processors get paired?
      - Who initiates the work transfer? push vs pull
      - How much work is transferred?



# Mapping to Minimize Interaction Overheads

- Maximize data locality
- Minimize volume of data-exchange
- Minimize frequency of interactions
- Minimize contention and hot spots
- Overlap computation with interactions
- Selective data and computation replication

Achieving the above is usually an interplay of decomposition and mapping and is usually done iteratively