Lecture 4: Principles of Parallel Algorithm Design

Constructing a Parallel Algorithm

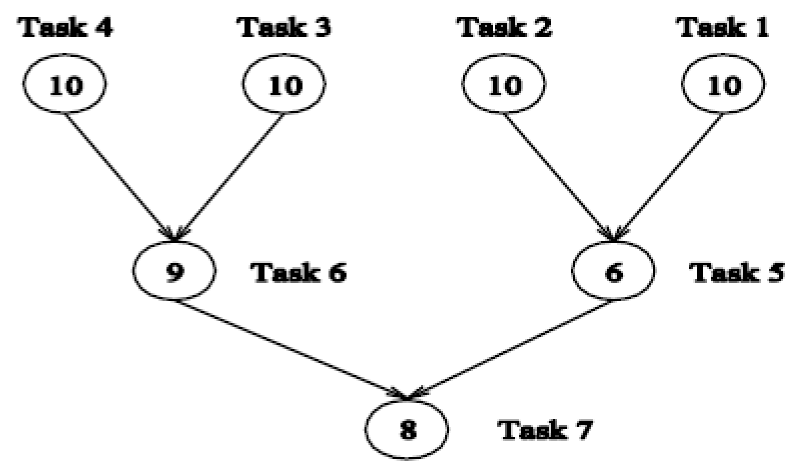
- *identify portions of work that can be performed concurrently*
- *map concurrent portions of work onto multiple processes running in parallel*
- distribute a program's input, output, and intermediate data
- manage accesses to shared data: avoid conflicts
- synchronize the processes at stages of the parallel program execution

Task Decomposition and Dependency Graphs

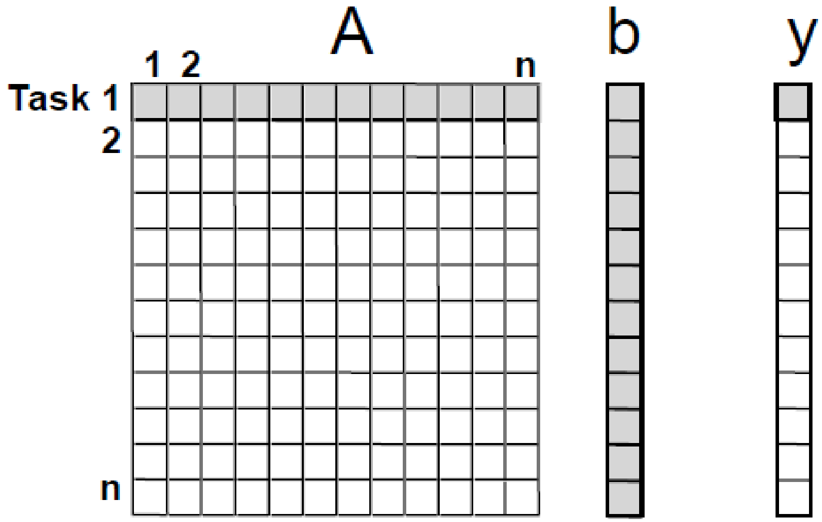
Decomposition: divide a computation into smaller parts, which can be executed concurrently

Task: programmer-defined units of computation.

Task-dependency graph:
Node represent s task.
Directed edge represents control dependence.



Example 1: Dense Matrix-Vector Multiplication



- Computing y[i] only use ith row of *A* and *b* – treat computing y[i] as a task.
- Remark:
  - Task size is uniform
  - No dependence between tasks
  - All tasks need *b*

Example 2: Database Query Processing

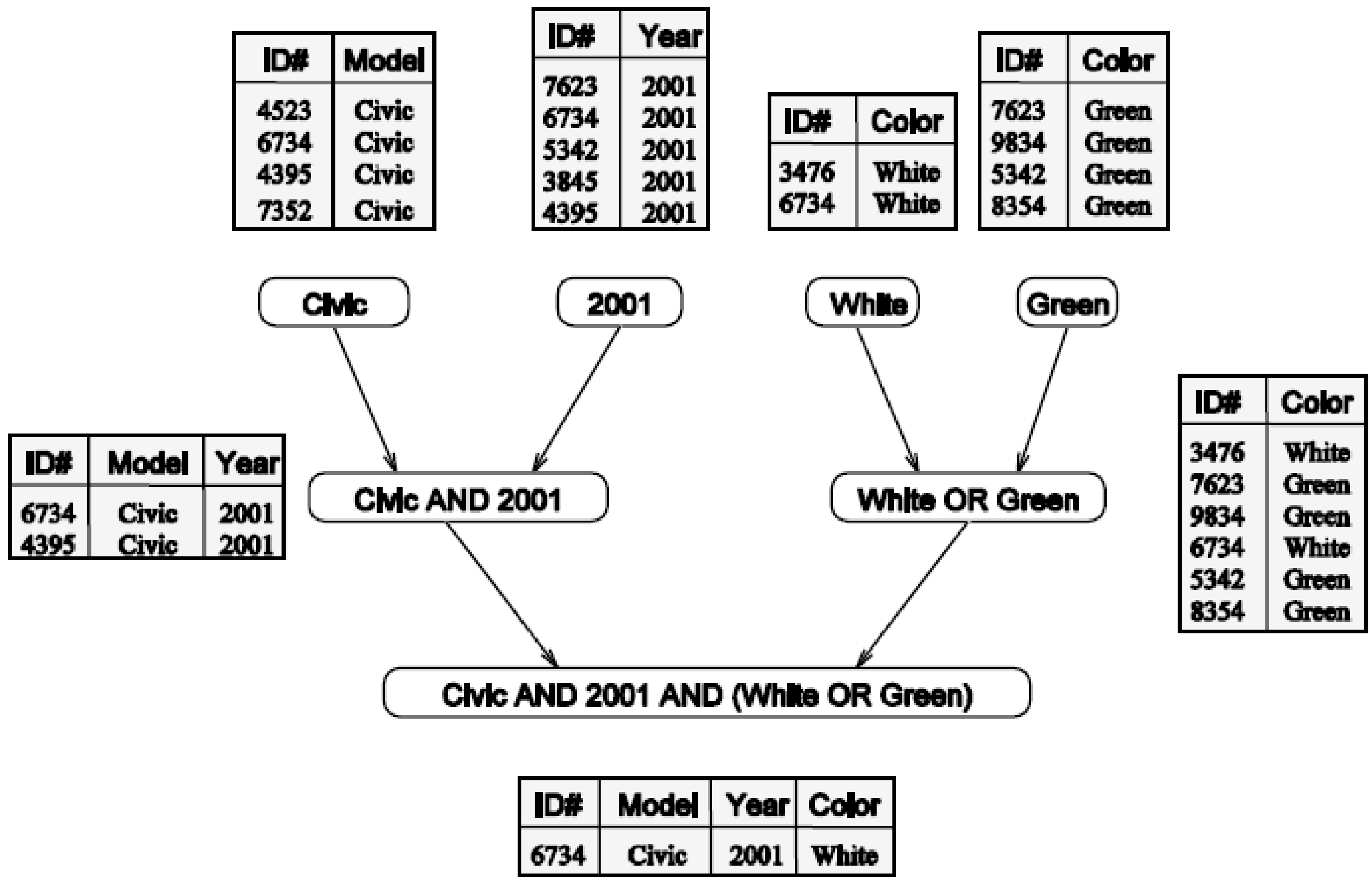
- Executing the query:

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

- **Task:** create sets of elements that satisfy a (or several) criteria.
- **Edge:** output of one task serves as input to the next



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

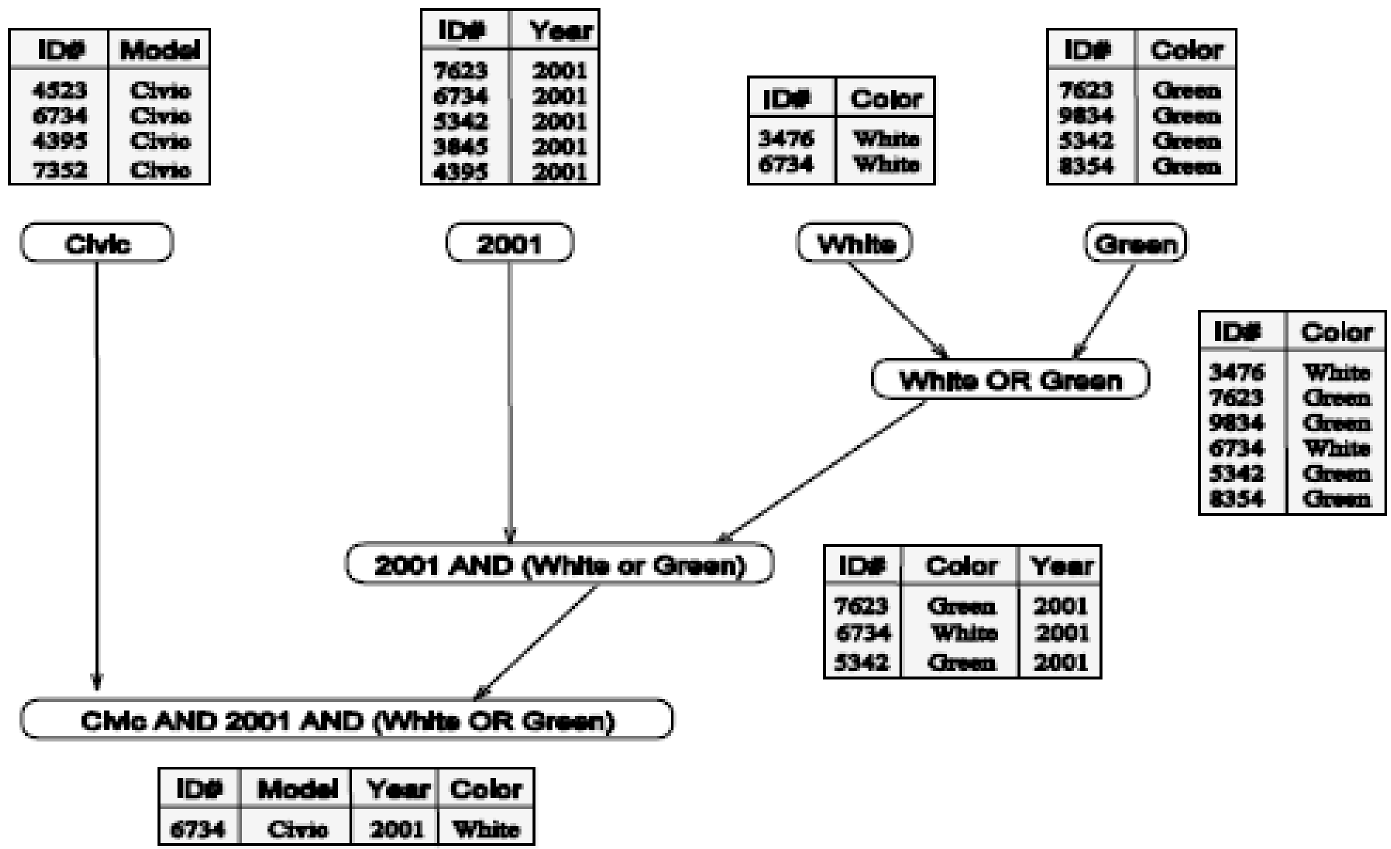
ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

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

ID#	Model	Year	Color
6734	Civic	2001	White

- An alternate task-dependency graph for query



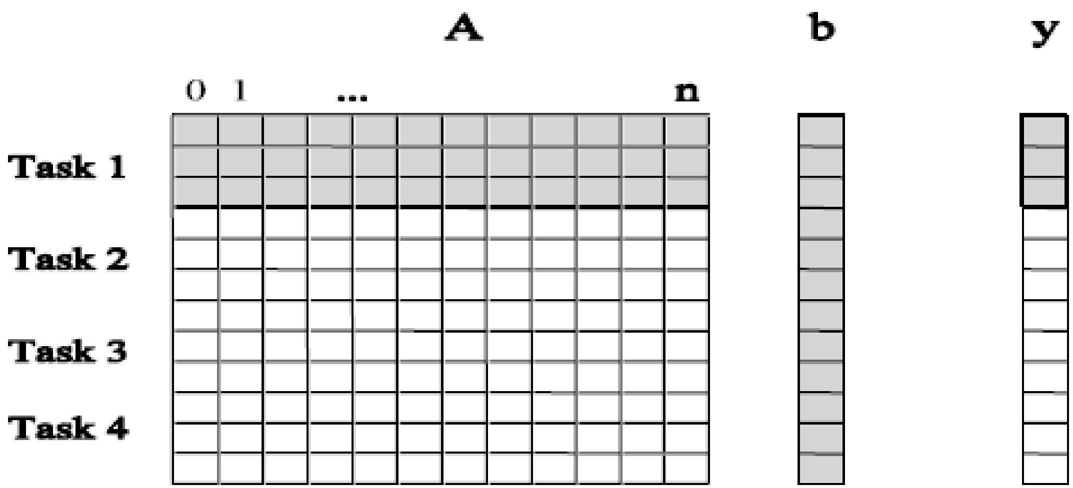
- Different task decomposition leads to different parallelism

Granularity of Task Decomposition

- **Fine-grained** decomposition: large number of small tasks
- **Coarse-grained** decomposition: small number of large tasks

Matrix-vector multiplication example

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



Degree of Concurrency

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

  -- **maximum degree of concurrency:** largest # of concurrent tasks at any point of the execution

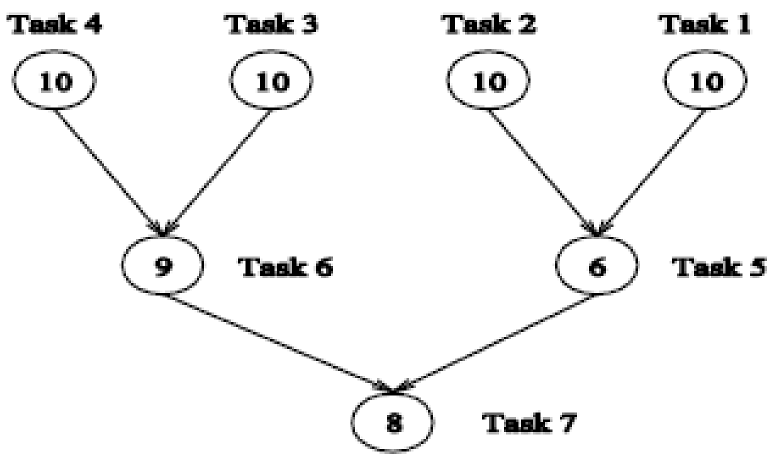
  -- **average degree of concurrency:** average # of tasks that can be executed concurrently
- Degree of Concurrency vs. Task Granularity
  - Inverse relation

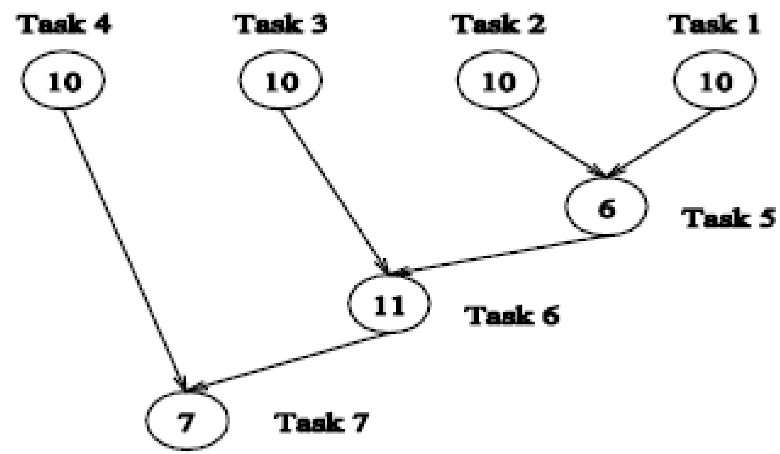
Critical Path of Task Graph

- **Critical path**: The longest directed path between any pair of *start node* (node with no incoming edge) and *finish node* (node with on 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

Example: Critical Path Length

Task-dependency graphs of query processing operation





Left graph:

Critical path length = 27

Average degree of concurrency = 63/27 = 2.33

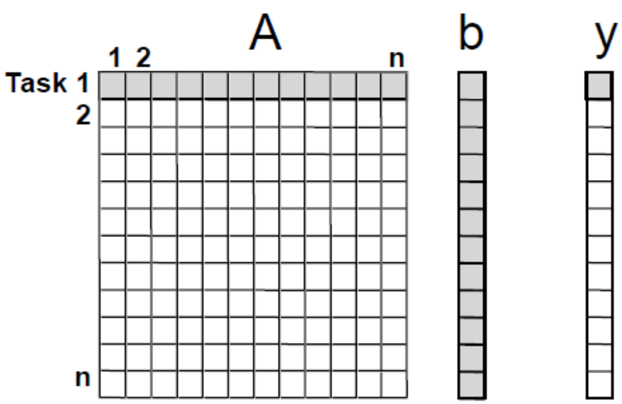
Right graph:

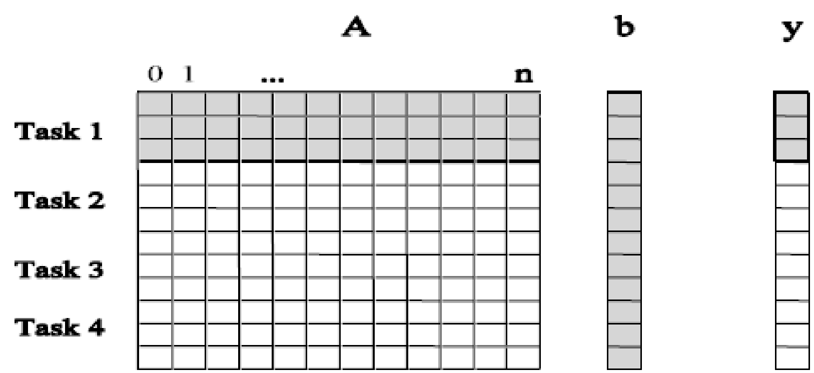
Critical path length = 34

Average degree of concurrency = 64/34 = 1.88

Limits on Parallelization

- Facts bounds on parallel execution
  - Maximum task granularity is finite
    - Matrix-vector multiplication $O(n^2)$
  - Interactions between tasks
    - Tasks often share input, output, or intermediate data, which may lead to interactions not shown in task-dependency graph.





Ex. For the matrix-vector multiplication problem, all tasks are independent, and all need access to the entire input vector b.

- **Speedup** = sequential execution time/parallel execution time
- **Parallel efficiency** = sequential execution time/(parallel execution time × processors used)

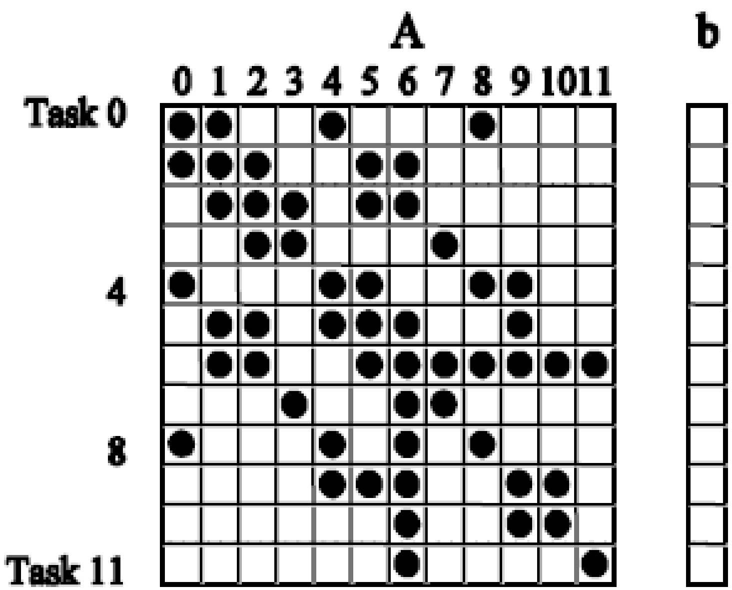
Task Interaction Graphs

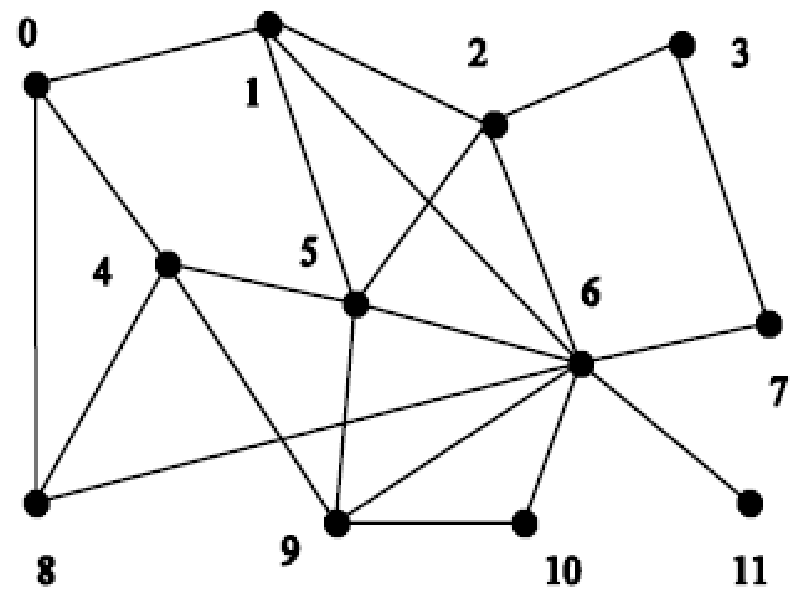
- Tasks generally share input, output or intermediate data
  - Ex. Matrix-vector multiplication: originally there is only one copy of *b*, tasks will have to communicate *b*.
- **Task-interaction graph**
  - To capture interactions among tasks
  - Node = task
  - Edge(undirected/directed) = interaction or data exchange
- Task-dependency graph vs. task-interaction graph
  - Task-dependency graph represents *control dependency*
  - Task-interaction graph represents *data dependency*
  - The edge-set of a task-interaction graph is usually a superset of the edge-set of the task-dependency graph

Example: Task-Interaction Graph

Sparse matrix-vector multiplication

- **Tasks**: each task computes an entry of y[]
- Assign *i*th row of A to Task *i*. Also assign b[i] to Task *i*.





Processes and Mapping

- **Mapping**: the mechanism by which tasks are assigned to processes for execution.
- **Process**: a logic computing agent that performs tasks, which is an abstract entity that uses the code and data corresponding to a task to produce the output of that task.
- Why use processes rather than processors?
  - We rely on OS to map processes to physical processors.
  - We can aggregate tasks into a process

Criteria of Mapping

1. Maximize the use of concurrency by mapping independent tasks onto different processes
2. Minimize the total completion time by making sure that processes are available to execute the tasks on critical path as soon as such tasks become executable
3. Minimize interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process.

Basis for Choosing Mapping

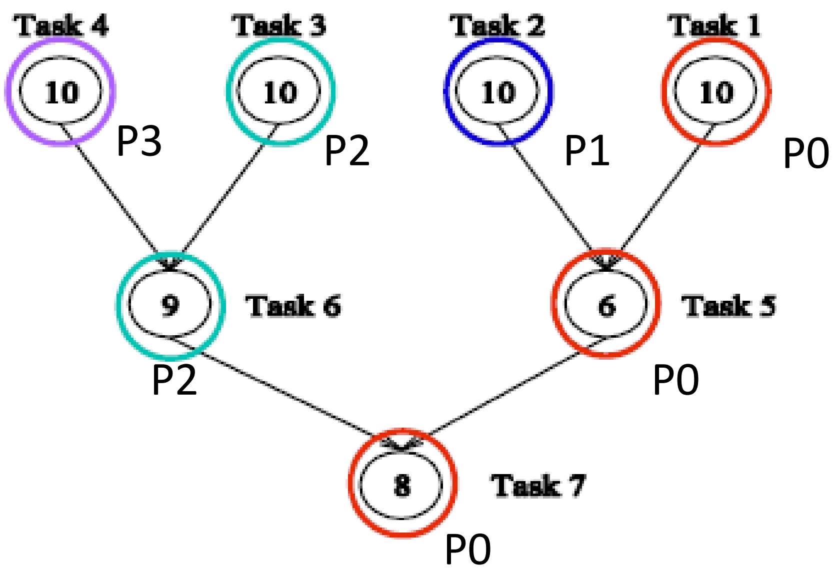
Task-dependency graph

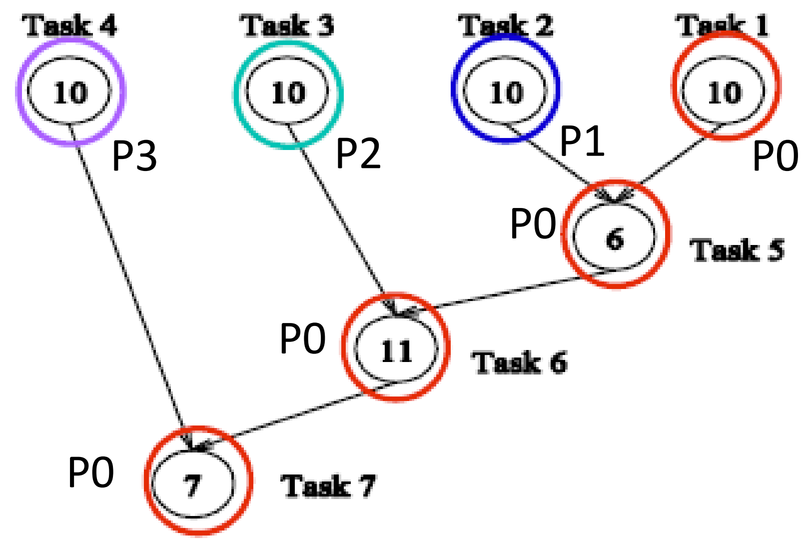
  Makes sure the max. concurrency

Task-interaction graph

  Minimum communication.

Example: Mapping Database Query to Processes





- 4 processes can be used in total since the max. concurrency is 4.
- Assign all tasks within a level to different processes.

Decomposition Techniques

How to decompose a computation into a set of tasks?

✓ Recursive decomposition

✓ Data decomposition

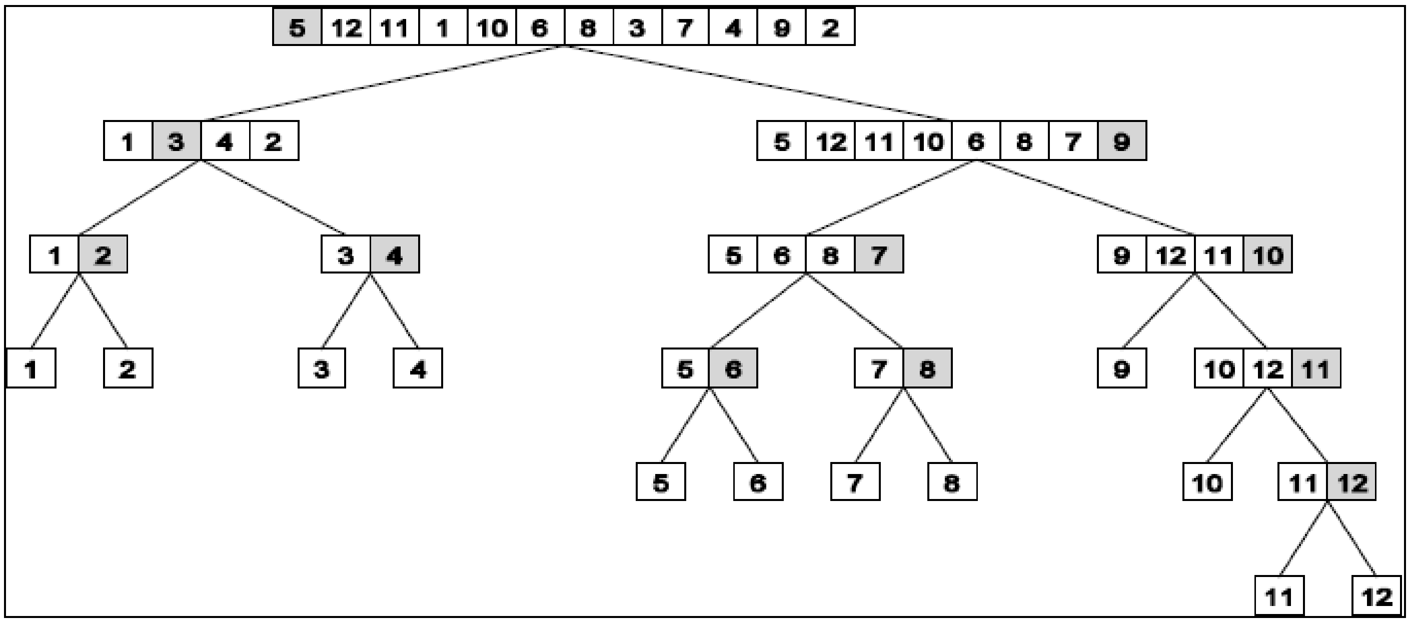
- Exploratory decomposition
- Speculative decomposition

Recursive Decomposition

- *Ideal for problems to be solved by divide-and-conquer method.*
- **Steps**
  1. Decompose a problem into a set of independent sub-problems
  2. Recursively decompose each sub-problem
  3. Stop decomposition when minimum desired granularity is achieved or (partial) result is obtained

Quicksort Example

Sort a sequence A of *n* elements in the increasing order.



- Select a pivot
- Partition the sequence around the pivot
- Recursively sort each sub-sequence

Task: the work of partitioning a given sub-sequence

Recursive Decomposition for Finding Min

Find the minimum in an array of numbers A of length *n*

```
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
```

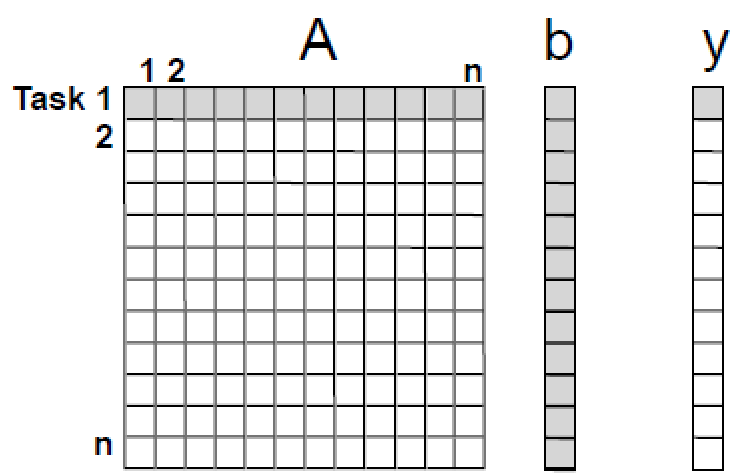
```
procedure  Recursive_MIN(A,n)
begin
    if (n == 1) then
        min := A[0];
    else
        lmin := Recursive_MIN(A,n/2);
        rmin := Recursive_MIN(&[A/2],n-n/2);
        if( lmin < rmin) then
            min := lmin;
        else
            min := rmin;
        endelse;
    endelse;
   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

Data Decomposition

- *Ideal for problems that operate on large data structures*
- **Steps**
  1. The data on which the computations are performed are partitioned
  2. Data partition is used to induce a partitioning of the computations into tasks.
- Data Partitioning
  - Partition output data
  - Partition input data
  - Partition input + output data
  - Partition intermediate data

Data Decomposition Based on Partitioning Output Data

- If each element of the output can be computed independently of others as a function of the input.
- Partitioning computations into tasks is natural. Each task is assigned with the work of computing a portion of the output.
- **Example**. Dense matrix-vector multiplication.



Example: Output Data Decomposition

Matrix-matrix multiplication: $C = A \times B$

- Partition matrix C into 2× 2 submatrices
- Computation of C then can be partitioned into four tasks.

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

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

Remark: data-decomposition is different from task decomposition.
Same data decomposition can have different task 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: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

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

Data Decomposition Based on Partitioning Input Data

- Ideal if output is a single unknown value or the individual elements of the output can not be efficiently determined in isolation.
  - Example. Finding the minimum, maximum, or sum of a set of numbers.
  - Example. Sorting a set.
- Partitioning the input data and associating a task with each partition of the input data.

Data Decomposition Based on Partitioning Intermediate Data

- Applicable for problems which can be solved by multi-stage computations such that the output of one stage is the input to the subsequent stage.
- Partitioning can be based on input or output of an intermediate stage.

Example: Intermediate Data Decomposition

Dense matrix-matrix multiplication

- Original output data decomposition yields a maximum degree of concurrency of 4.

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

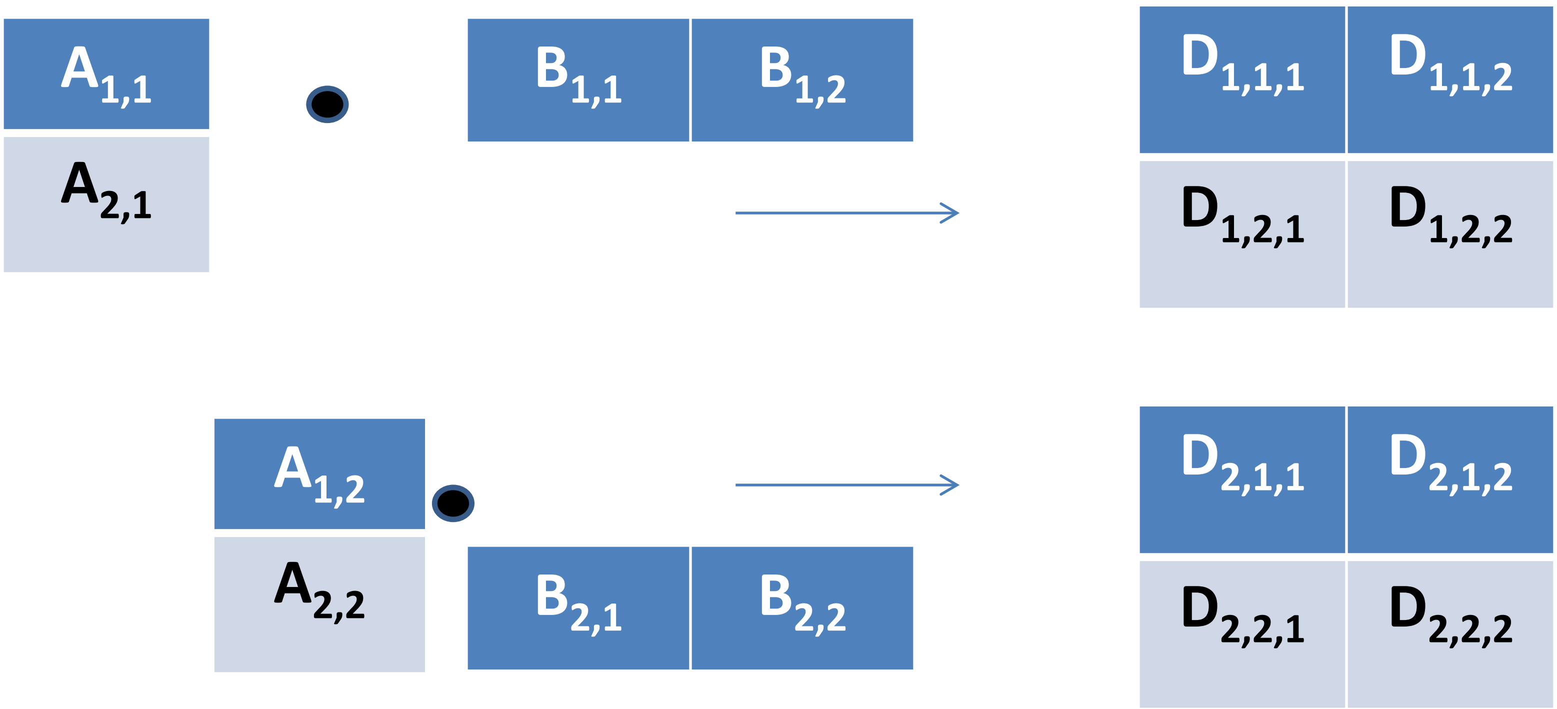
Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

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

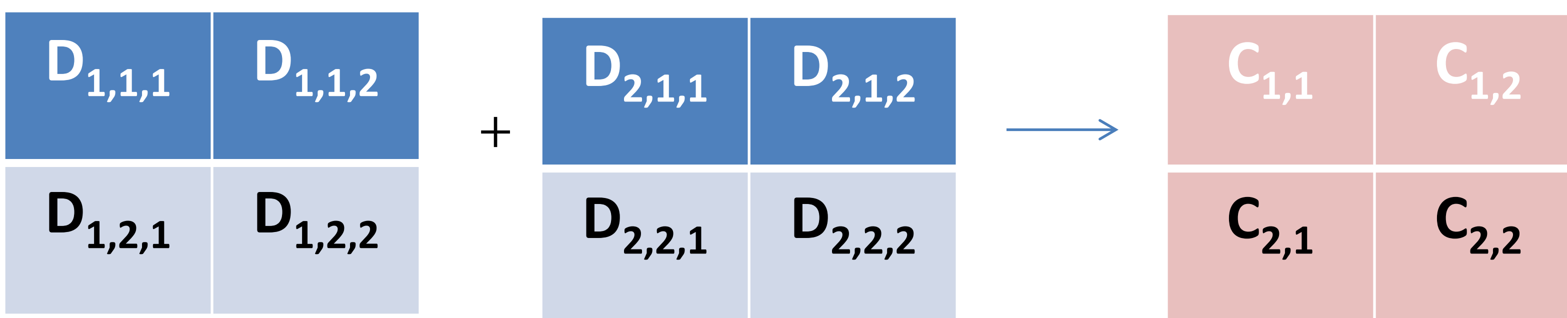
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

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

Stage 1: $D_{k,i,j} = A_{i,k} B_{k,j}$



Stage 2: $C_{i,j} = D_{1,i,j} + D_{2,i,j}$



Let $D_{k,i,j} = A_{i,k} \cdot B_{k,j}$

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: $D_{2,2,2} = A_{2,2}\, B_{2,2}$

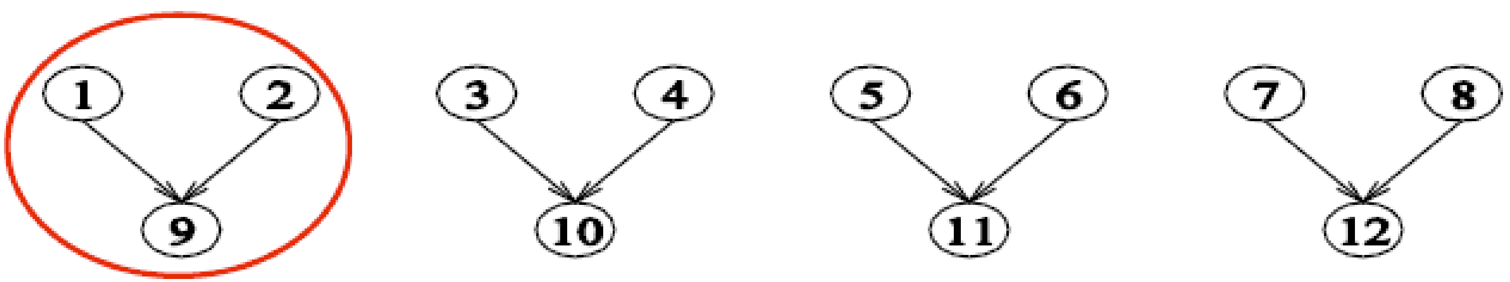
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

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: $C_{2..2} = D_{1.2.2} + D_{2.2.2}$

Task-dependency graph



Owner-Computes Rule

- Decomposition based on partitioning input/output data is referred to as the **owner-computes** rule.
  - Each partition performs all the computations involving data that it owns.
- Input data decomposition
  - A task performs all the computations that can be done using these input data.
- Output data decomposition
  - A task computes all the results in the partition assigned to it.

Characteristics of Tasks

Key characteristics of tasks influencing choice of mapping and performance of parallel algorithm:

1. Task generation
   - Static or dynamic generation
     - *Static:* all tasks are known before the algorithm starts execution. Data or recursive decomposition often leads to static task generation.

       Ex. Matrix-multiplication. Recursive decomposition in finding min. of a set of numbers.
     - *Dynamic:* the actual tasks and the task-dependency graph are not explicitly available *a priori*. Recursive, exploratory decomposition can generate tasks dynamically.

       Ex. Recursive decomposition in Quicksort, in which tasks are generated dynamically.
2. Task sizes
   - Amount of time required to compute it: *uniform, non-uniform*
3. Knowledge of task sizes
4. Size of data associated with tasks
   - Data associated with the task must be available to the process performing the task. The size and location of data may determine the data-movement overheads.

Characteristics of Task Interactions

1) Static versus dynamic
   – Static: interactions are known prior to execution.
2) Regular versus irregular
   – Regular: interaction pattern can be exploited for efficient implementation.
3) Read-only versus read-write
4) One-way versus two-way

Static vs. Dynamic Interactions

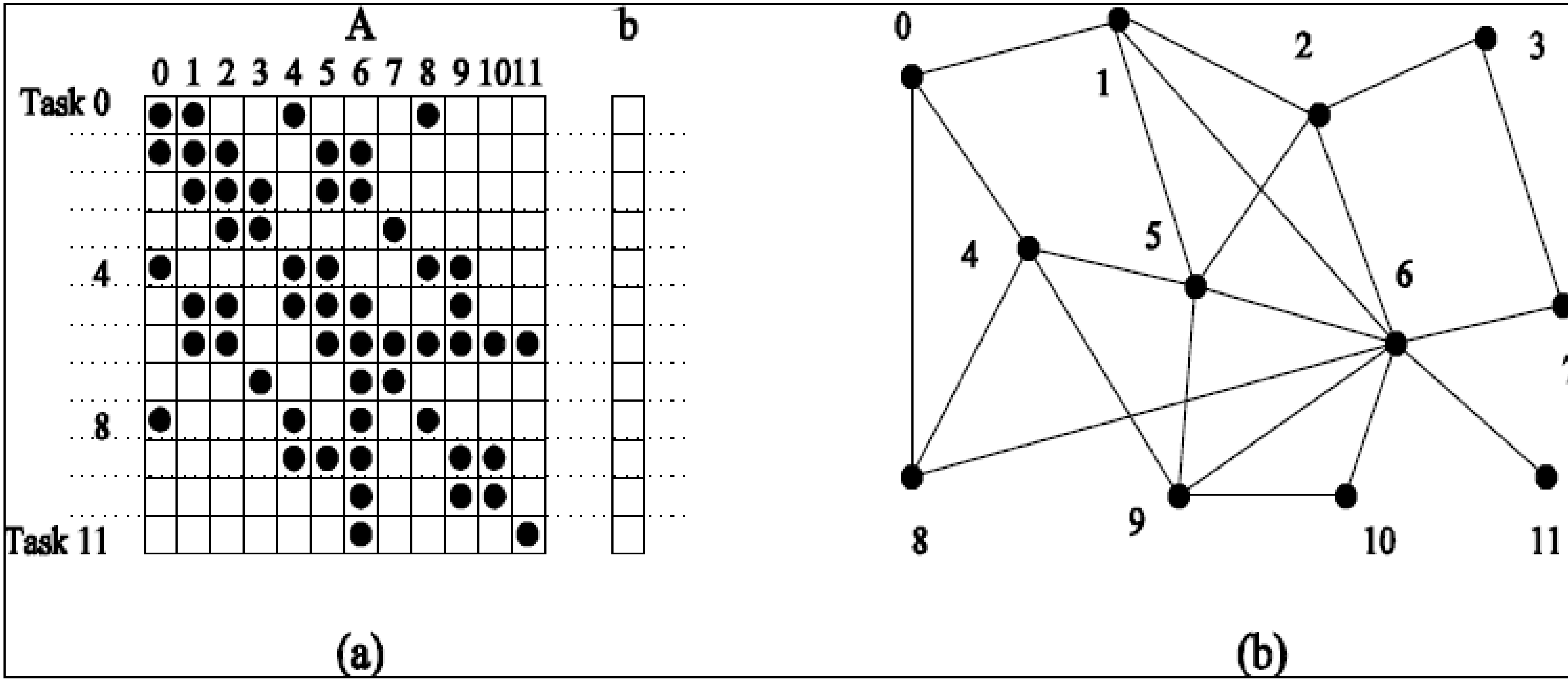
- Static interaction
  - Tasks and associated interactions are predetermined: task-interaction graph and times that interactions occur are known: matrix multiplication
  - Easy to program
- Dynamic interaction
  - Timing of interaction or sets of tasks to interact with can not be determined prior to the execution.
  - Difficult to program using massage-passing; Shared-memory space programming may be simple

Regular vs. Irregular Interactions

- Regular interactions
  - Interaction has a spatial structure that can be exploited for efficient implementation: ring, mesh

    Example: Explicit finite difference for solving PDEs.
- Irregular Interactions
  - Interactions has no well-defined structure

    Example: Sparse matrix-vector multiplication


A
b
0 1 2 3 4 5 6 7 8 9 10 11
Task 0
4
8
Task 11
0
1
2
3
4
5
6
7
8
9
10
11
(a)
(b)

Mapping Technique for Load Balancing

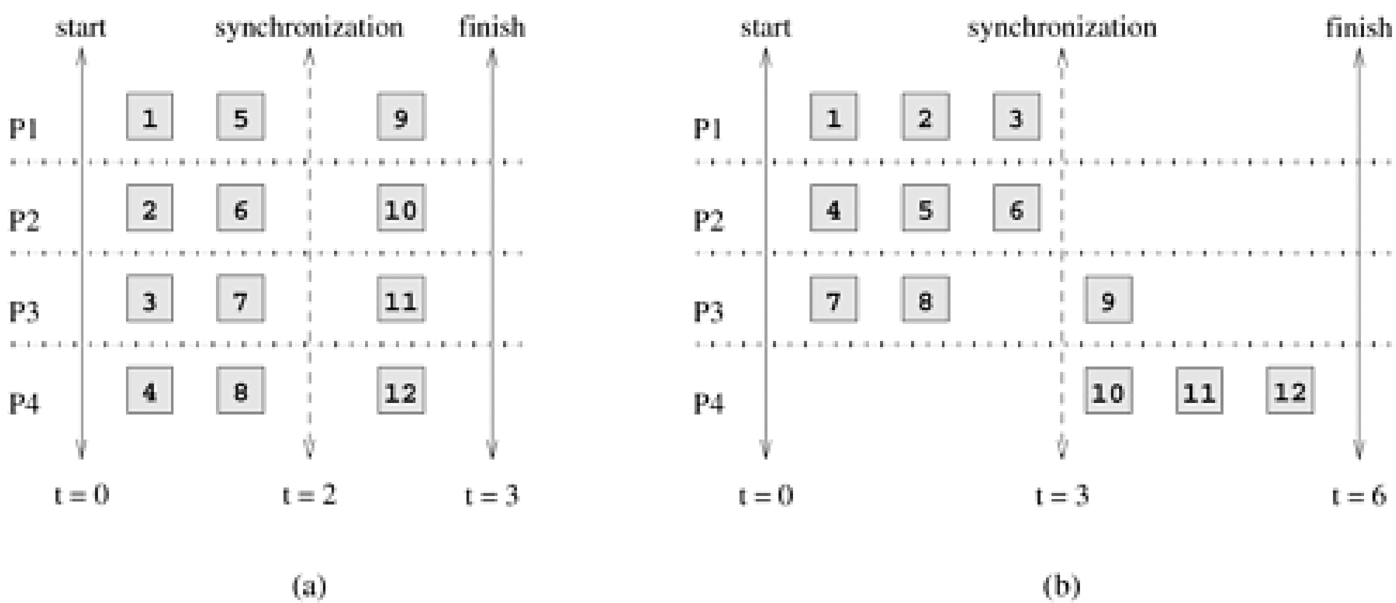
Minimize execution time → Reduce overheads of execution

- Sources of overheads:
  - Inter-process interaction
  - Idling
  - Both interaction and idling are often a function of mapping
- Goals to achieve:
  - To reduce interaction time
  - *To reduce total amount of time some processes being idle (goal of load balancing)*
  - Remark: these two goals often conflict
- Classes of mapping:
  - Static
  - Dynamic

Remark:

1. Loading balancing is **only** a necessary **but not** sufficient condition for reducing idling.
   - Task-dependency graph determines which tasks can execute in parallel and which must wait for some others to finish at a given stage.
2. Good mapping must ensure that computations and interactions among processes at each stage of execution are well balanced.

Figure 3.23. Two mappings of a hypothetical decomposition with a synchronization.



Two mappings of 12-task decomposition in which the last 4 tasks can be started only after the first 8 are finished due to task-dependency.

Schemes for Static Mapping

Static Mapping: It distributes the tasks among processes prior to the execution of the algorithm.

- Mapping Based on Data Partitioning
- Task Graph Partitioning
- Hybrid Strategies

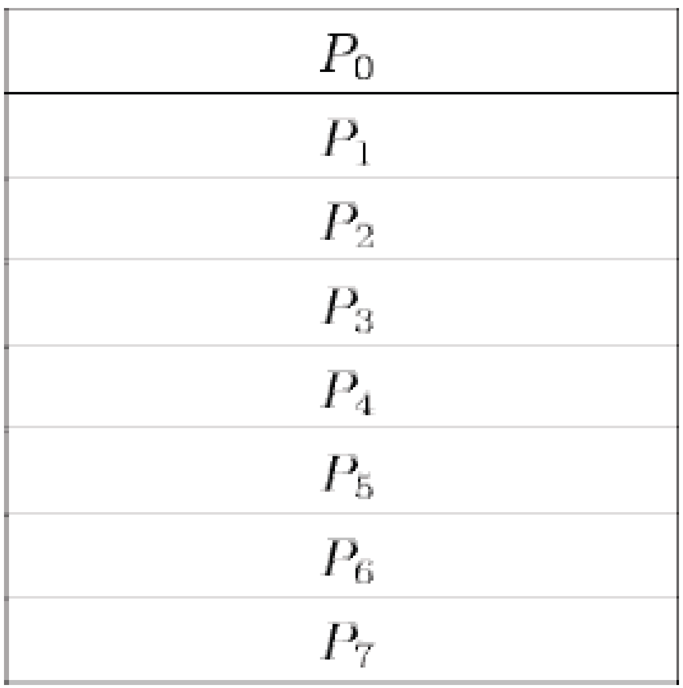
Mapping Based on Data Partitioning

- By owner-computes rule, mapping the relevant data onto processes is equivalent to mapping tasks onto processes
- Array or Matrices
  - Block distributions
  - Cyclic and block cyclic distributions
- Irregular Data
  - Example: data associated with unstructured mesh
  - Graph partitioning

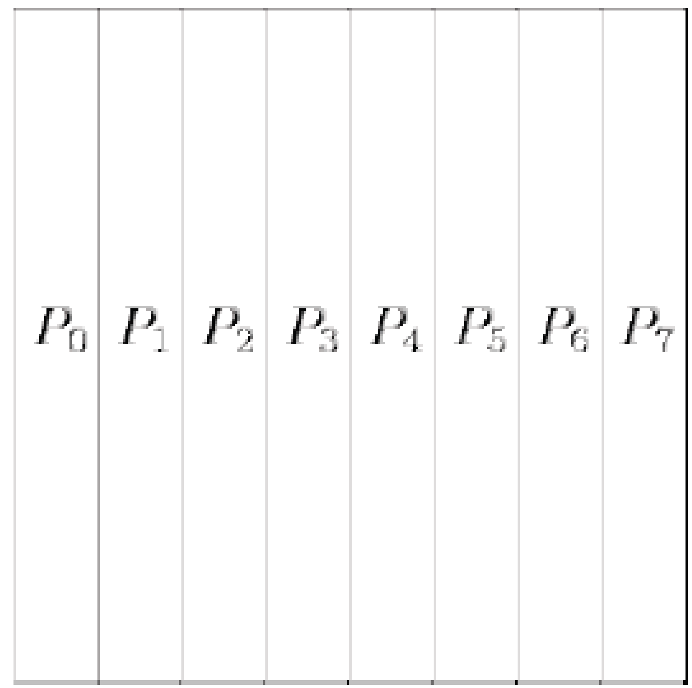
1D Block Distribution

Example. Distribute rows or columns of matrix to different processes

row-wise distribution



column-wise distribution



Multi-D Block Distribution

Example. Distribute blocks of matrix to different processes

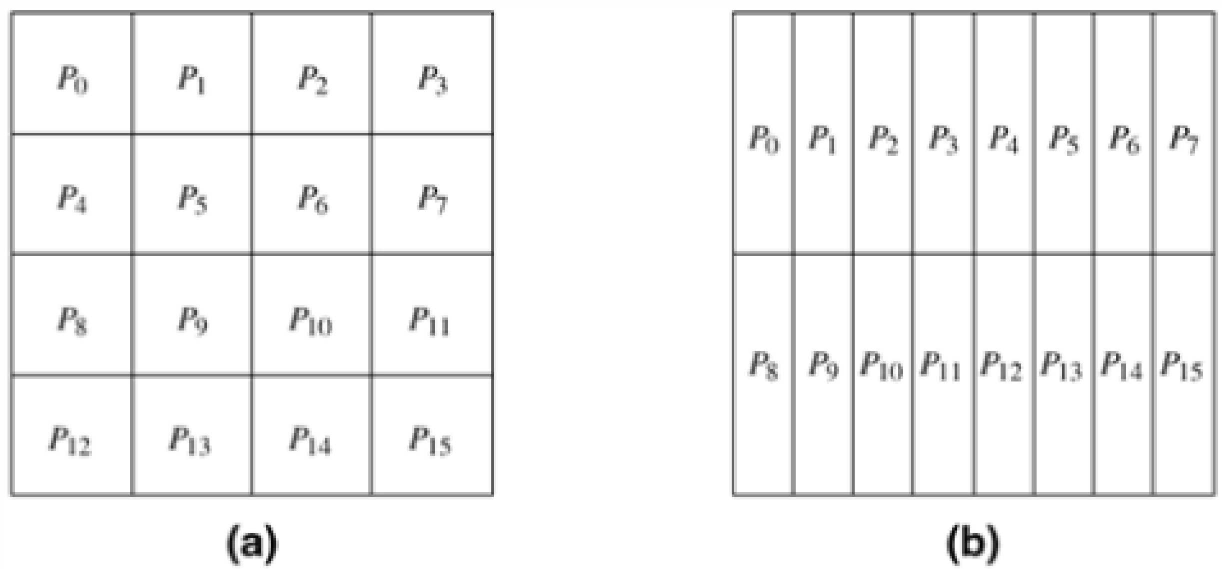


Figure 3.25. Examples of two-dimensional distributions of an array, (a) on a 4 × 4 process grid, and (b) on a 2 × 8 process grid.

Load-Balance for Block Distribution

Example. $n \times n$ dense matrix multiplication $C = A \times B$ using p processes

- Decomposition based on output data.
- Each entry of C use the same amount of computation.
- Either 1D or 2D block distribution can be used:
  - 1D distribution: $\frac{n}{p}$ rows are assigned to a process
  - 2D distribution: $n/\sqrt{p} \times n/\sqrt{p}$ size block is assigned to a process
- Multi-D distribution allows higher degree of concurrency.
- Multi-D distribution can also help to reduce interactions

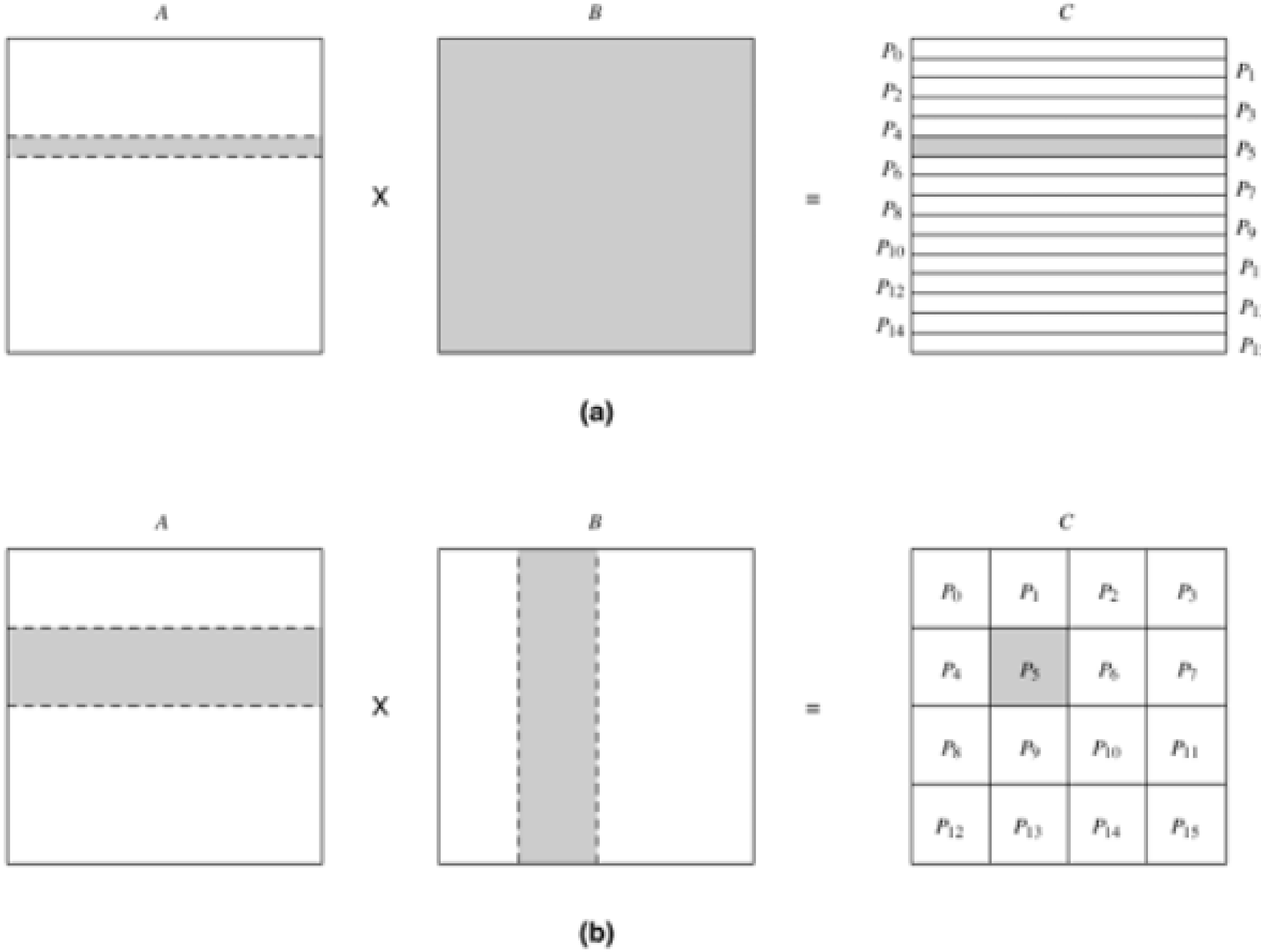


Figure 3.26. Data sharing needed for matrix multiplication with (a) one-dimensional and (b) two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C.

Suppose the size of matrix is $n \times n$, and p processes are used.

(a): A process need to access $\frac{n^2}{p} + n^2$ amount of data

(b): A process need to access $O(n^2/\sqrt{p})$ amount of data

Cyclic and Block Cyclic Distributions

- If the amount of work differs for different entries of a matrix, a block distribution can lead to load imbalances.
- Example. Doolittle's method of LU factorization of dense matrix
  - The amount of computation increases from the top left to the bottom right of the matrix.

Doolittle's method of LU factorization

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} = LU = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

By matrix-matrix multiplication

$u_{1j} = a_{1j},$ $\quad j = 1,2,\dots,n$ $(1st$ row of $U)$

$l_{j1} = a_{j1}/u_{11},$ $\quad j = 1,2,\dots,n$ $(1st$ column of $L)$

For $i = 2,3,\dots,n-1$ **do**

$u_{ii} = a_{ii} - \sum_{t=1}^{i-1} l_{it}u_{ti}$

$u_{ij} = a_{ij} - \sum_{t=1}^{i-1} l_{it}u_{tj}$ $\quad$ for $j = i+1,\dots,n$ $(ith$ row of $U)$

$l_{ji} = \frac{a_{ji} - \sum_{t=1}^{i-1} l_{jt}u_{ti}}{u_{ii}}$ $\quad$ for $j = i+1,\dots,n$ $(ith$ column of $L)$

End

$u_{nn} = a_{nn} - \sum_{t=1}^{n-1} l_{nt}u_{tn}$

Serial Column-Based LU

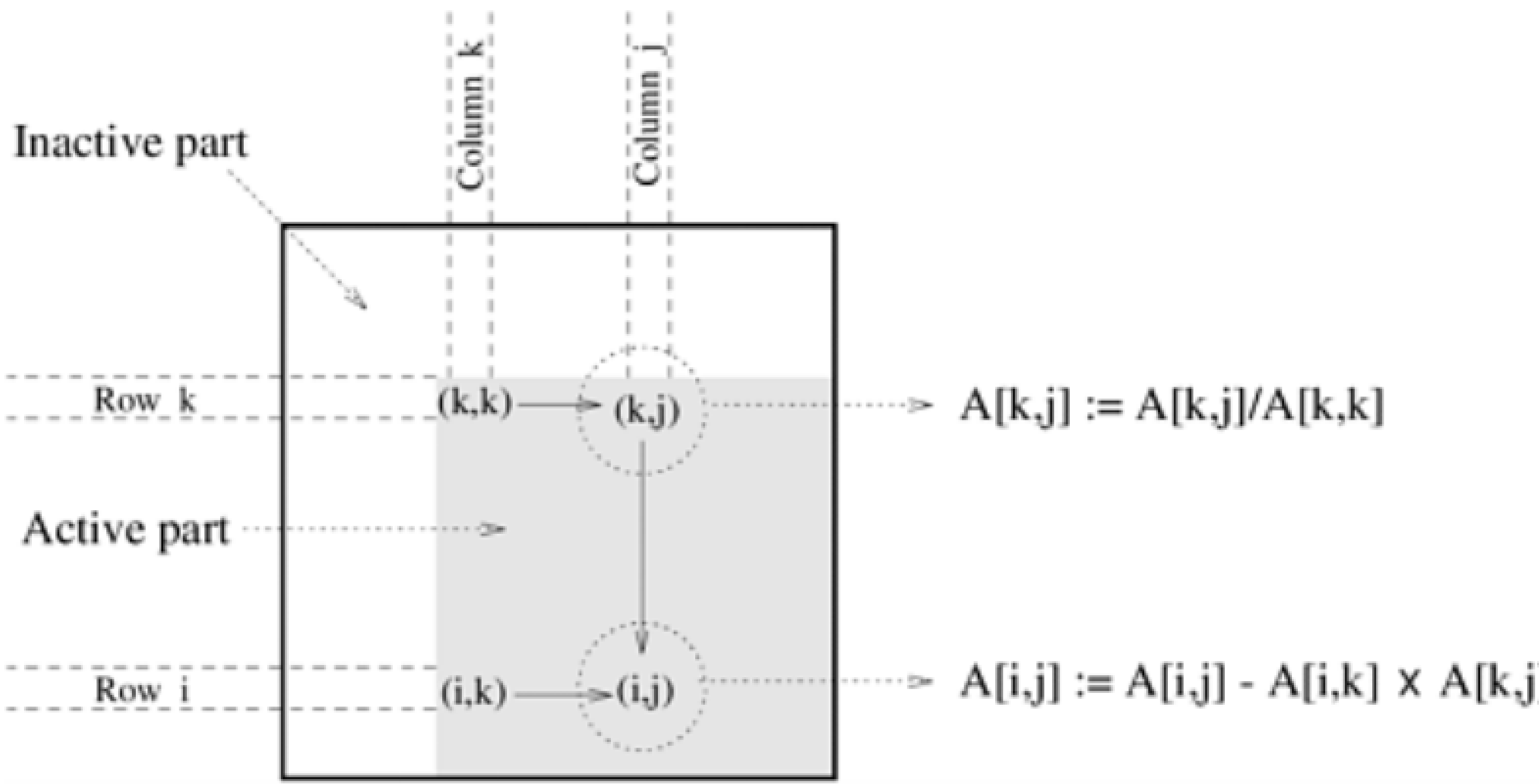
```
procedure COL_LU (A)
begin
   for k := 1 to n do
       for j := k to n do
           A[j, k]:= A[j, k]/A[k, k];
       endfor;
       for j := k + 1 to n do
           for i := k + 1 to n do
               A[i, j] := A[i, j] - A[i, k] × A[k, j];
           endfor;
       endfor;
  /*
After this iteration, column A[k + 1 : n, k] is logically the kth
column of L and row A[k, k : n] is logically the kth row of U.
  */
   endfor;
end COL_LU
```

- Remark: Matrices L and U share space with A

Work used to compute Entries of L and U

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

1: $A_{1,1} \rightarrow L_{1,1}U_{1,1}$
2: $L_{2,1} = A_{2,1}U_{1,1}^{-1}$
3: $L_{3,1} = A_{3,1}U_{1,1}^{-1}$
4: $U_{1,2} = L_{1,1}^{-1}A_{1,2}$
5: $U_{1,3} = L_{1,1}^{-1}A_{1,3}$
6: $A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$
7: $A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$
8: $A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$
~~9:~~ $\cancel{A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}}$
10: $A_{2,2} \rightarrow L_{2,2}U_{2,2}$
11: $L_{3,2} = A_{3,2}U_{2,2}^{-1}$
12: $U_{2,3} = L_{2,2}^{-1}A_{2,3}$
~~13:~~ $\cancel{A_{3,3} = A_{3,3} - L_{3,2}}U_{2,3}$
~~14:~~ $\cancel{A_{3,3} \rightarrow L_{3,3}}U_{3,3}$



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

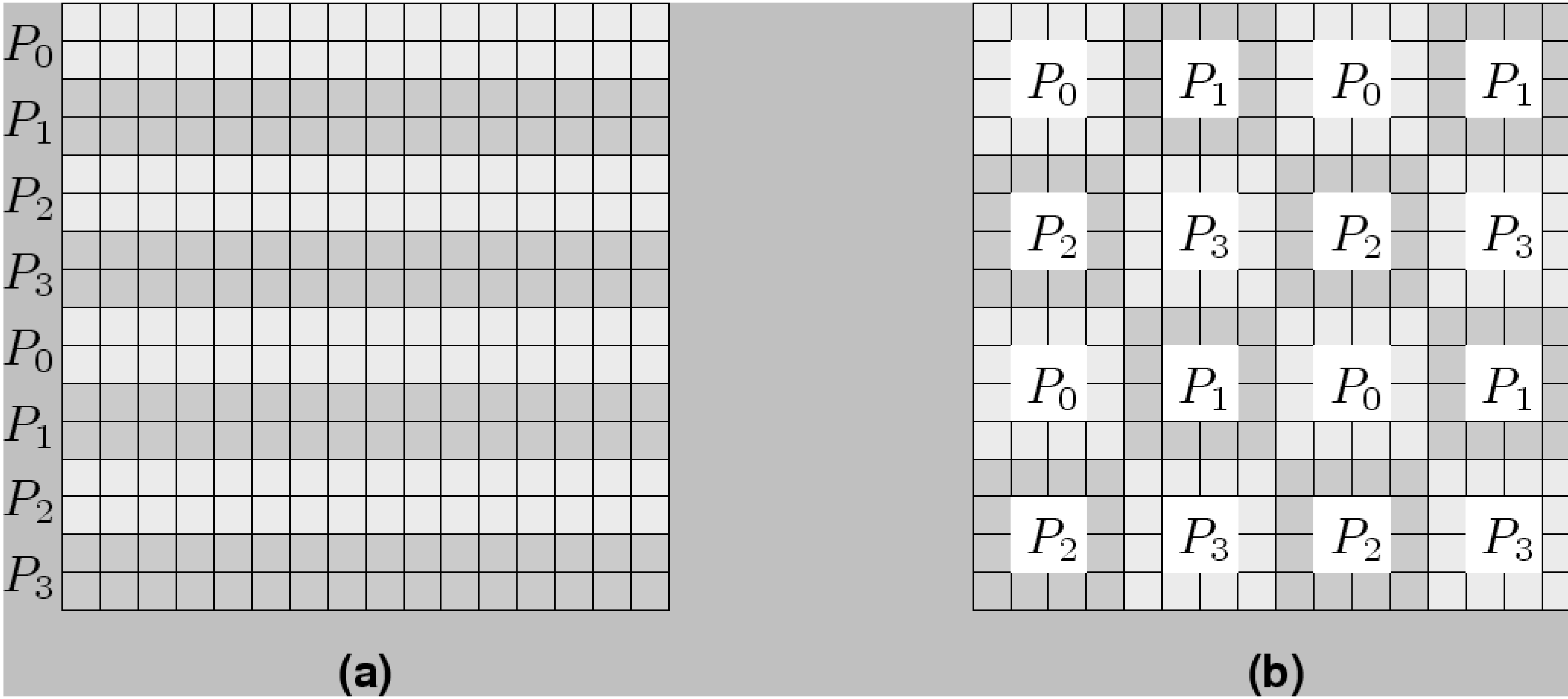
- Block distribution of LU factorization tasks leads to load imbalance.

P_0 T_1	P_3 T_4	P_6 T_5
P_1 T_2	P_4 T_6 T_{10}	P_7 T_8 T_{12}
P_2 T_3	P_5 T_7 T_{11}	P_8 T_9 T_{13} T_{14}

Block-Cyclic Distribution

- A variation of block distribution that can be used to alleviate the load-imbalance.

- Steps
  1. Partition an array into many more blocks than the number of available processes
  2. Assign blocks to processes in a *round-robin manner* so that each process gets several non-adjacent blocks.



(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 *wrap-around* fashion.

(b) The matrix is blocked into 16 blocks each of size 4×4, and it is mapped onto a 2×2 grid of processes in a wraparound fashion.

- **Cyclic distribution:** when the block size =1

Randomized Block Distribution

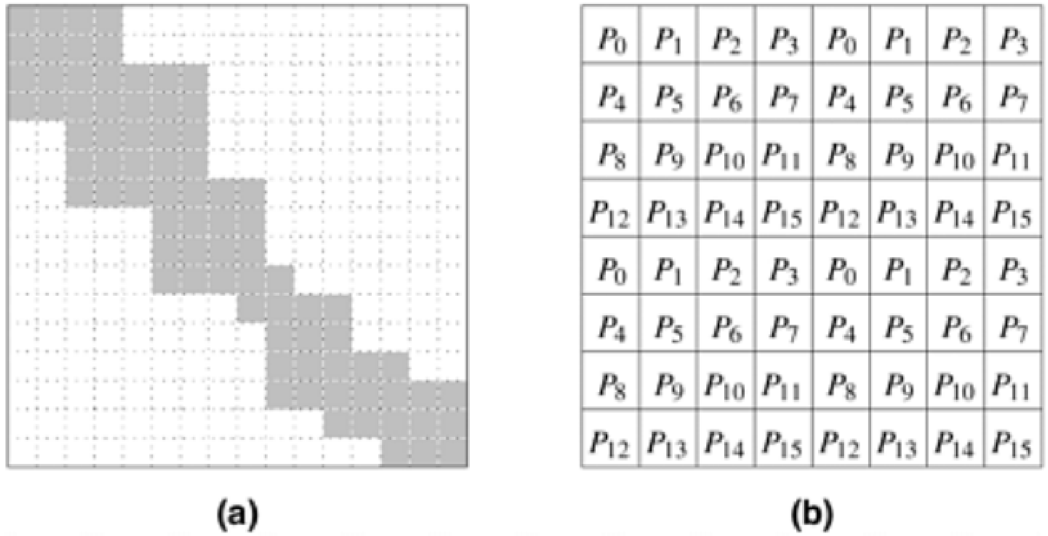


Figure 3.31. Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.

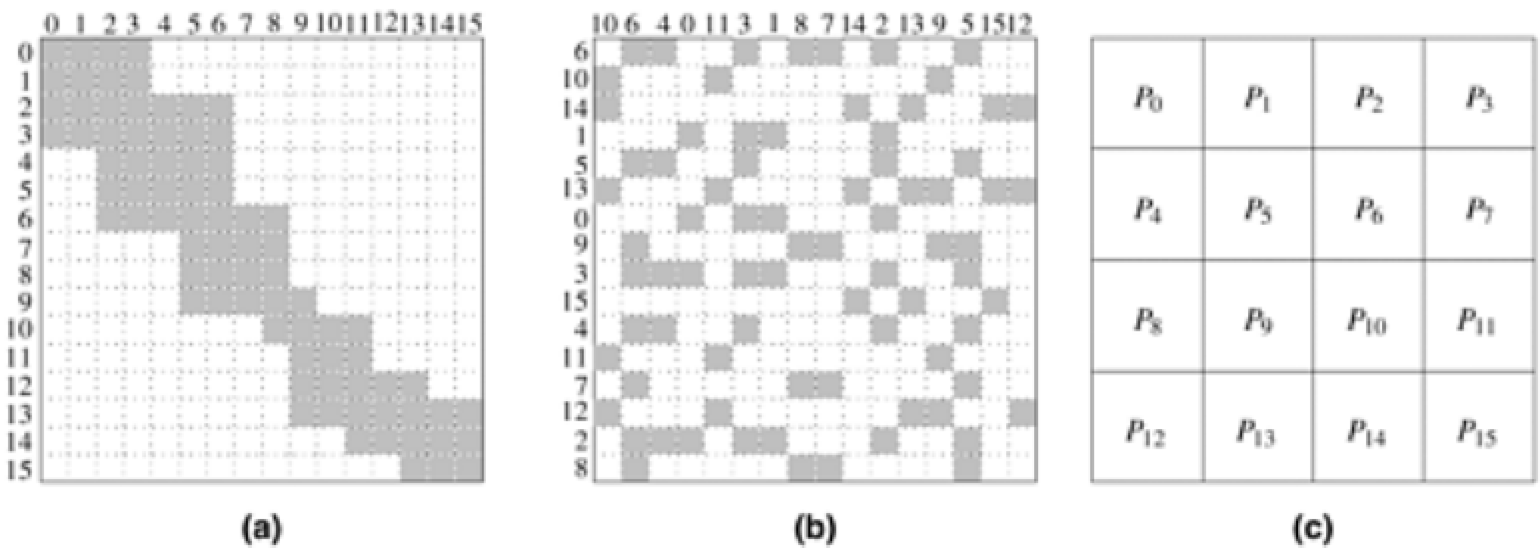
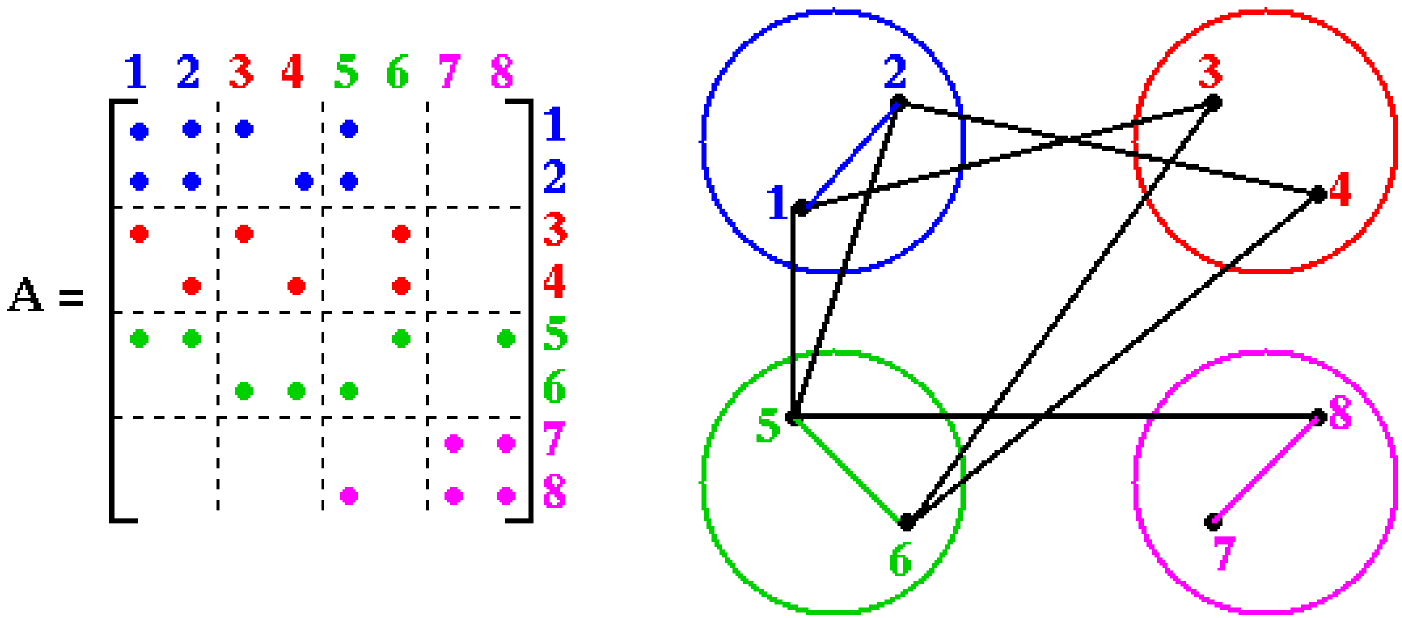


Figure 3.33. Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c).

Graph Partitioning

Sparse-matrix vector multiplication



Work: nodes
Interaction/communication: edges

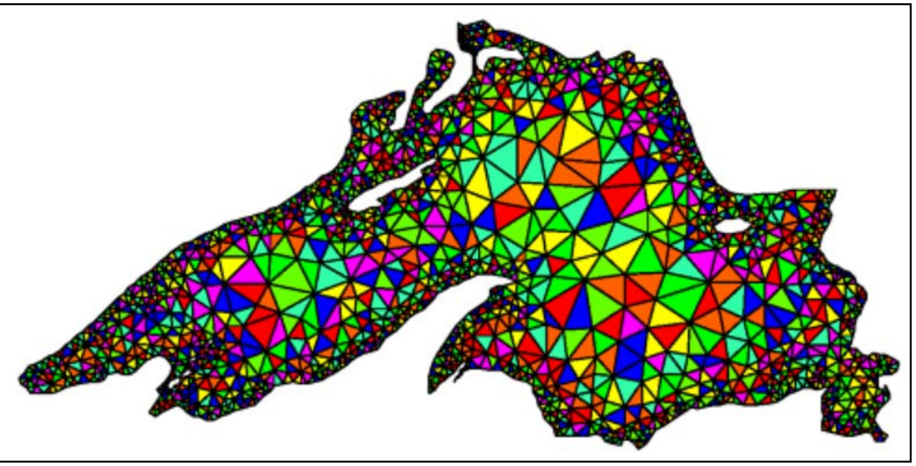
Partition the graph:

Assign roughly same number of nodes to each process

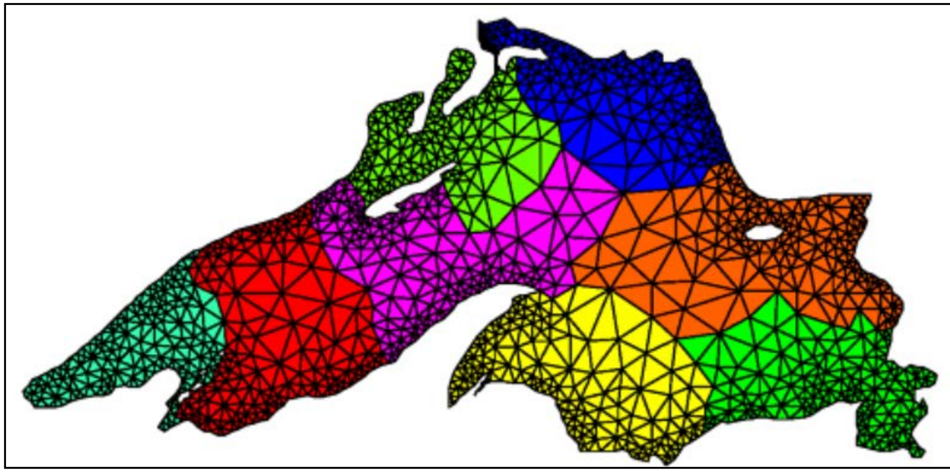
Minimize edge count of graph partition

Finite element simulation of water contaminant in a lake.

- Goal of partitioning: balance work & minimize communication



Random Partitioning



Partitioning for Minimizing Edge-Count

- Assign equal number of nodes (or cells) to each process
  - Random partitioning may lead to high interaction overhead due to data sharing
- Minimize edge count of the graph partition
  - Each process should get roughly the same number of elements and the number of edges that cross partition boundaries should be minimized as well.

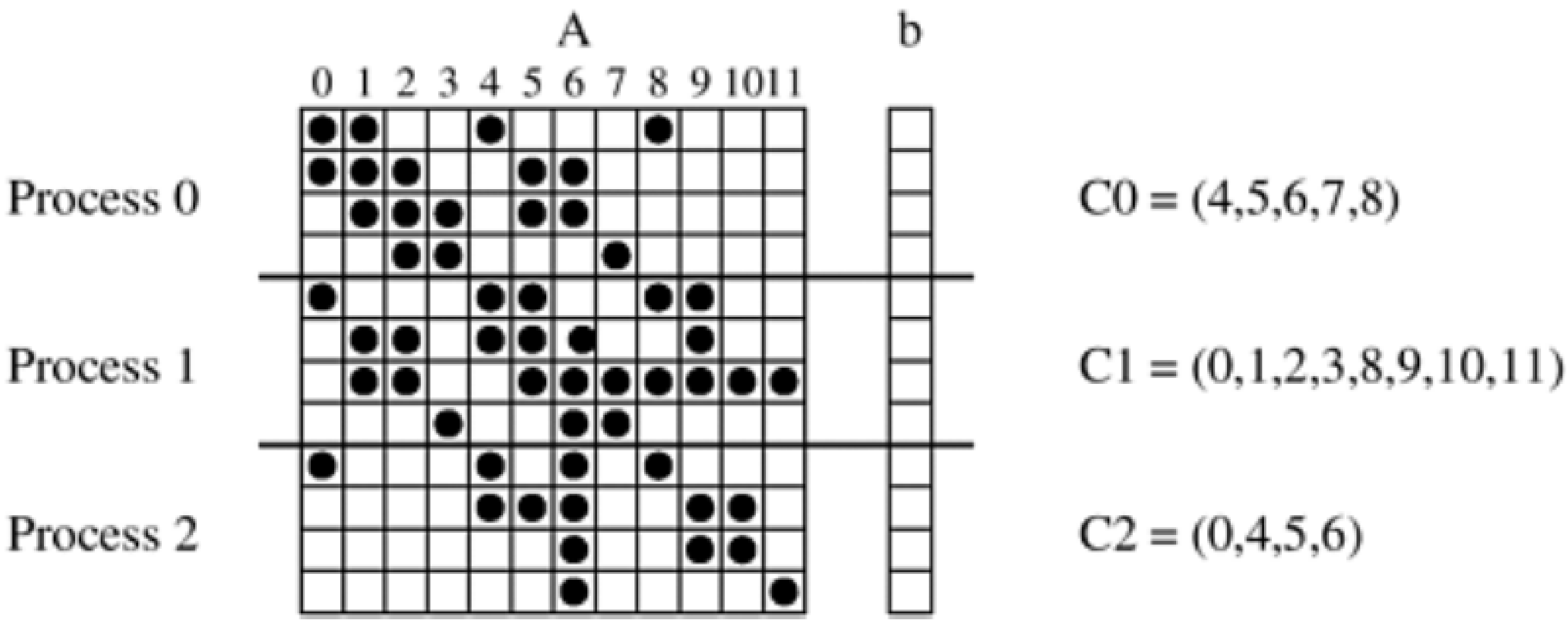
Mappings Based on Task Partitioning

- Mapping based on task partitioning can be used when computation is naturally expressed in the form of a *static task-dependency graph* with known sizes.
- Finding optimal mapping minimizing idle time and minimizing interaction time is NP-complete
- Heuristic solutions exist for many structured graphs

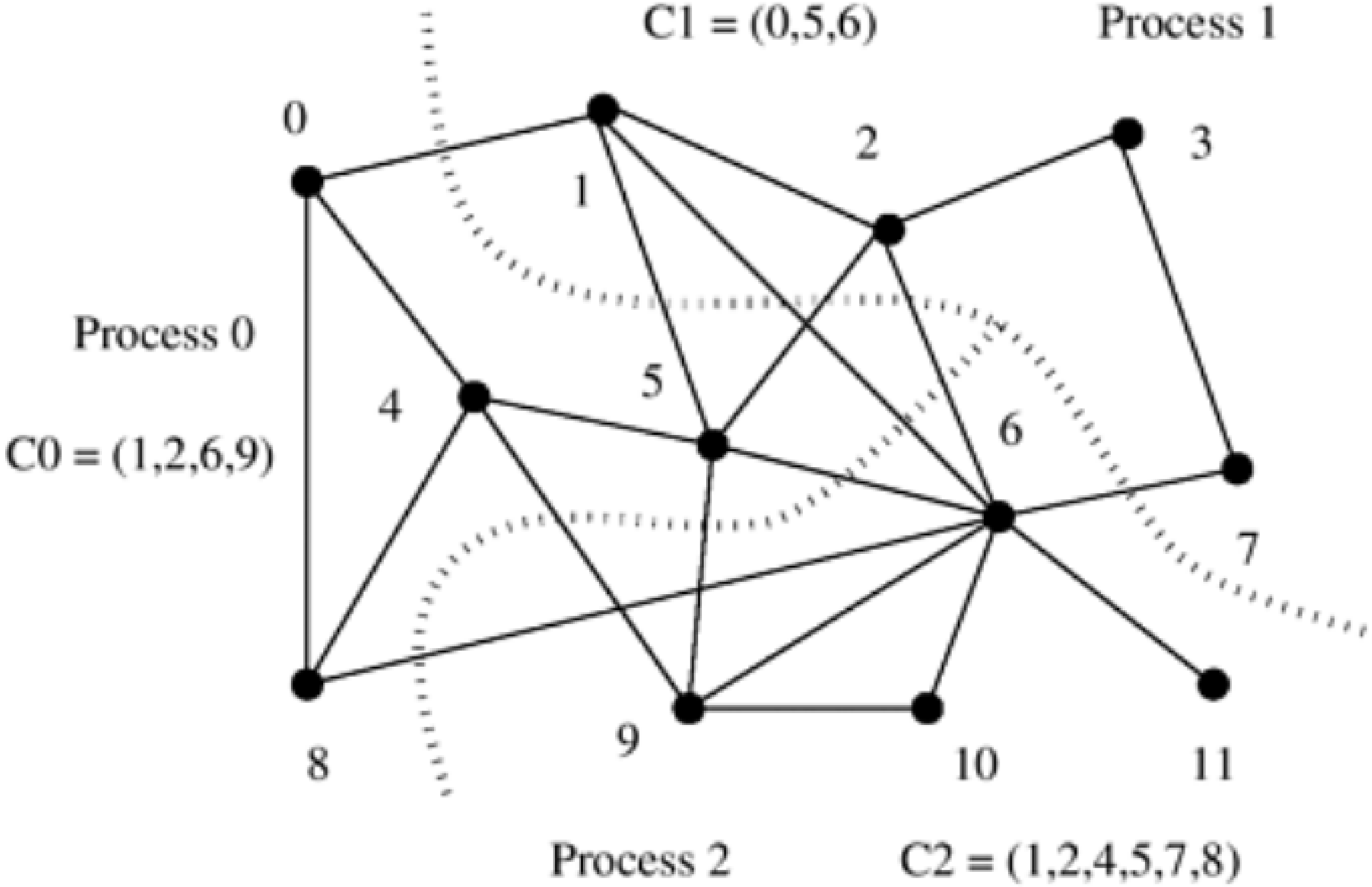
Mapping a Sparse Graph

Example. Sparse matrix-vector multiplication using 3 processes

- Arrow distribution



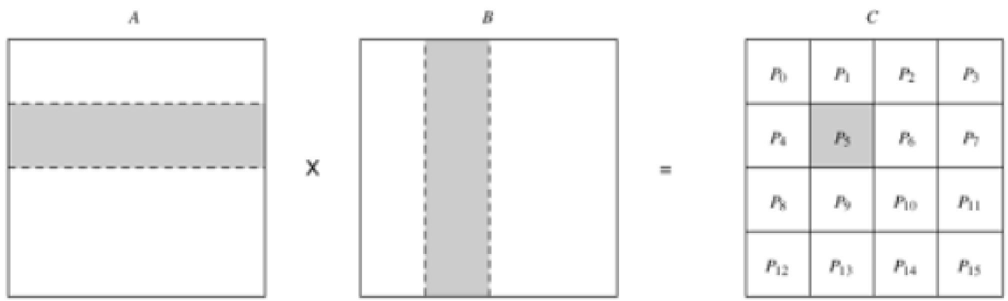
- Partitioning task-interaction graph to reduce interaction overhead



Techniques to Minimize Interaction Overheads

- Maximize data locality
  - Maximize the reuse of recently accessed data
  - Minimize volume of data-exchange
    - Use high dimensional distribution. Example: 2D block distribution for matrix multiplication
  - Minimize frequency of interactions
    - Reconstruct algorithm such that shared data are accessed and used in large pieces.
    - Combine messages between the same source-destination pair

- Minimize contention and hot spots
  - Competition occur when multi-tasks try to access the same resources concurrently: multiple processes sending message to the same process; multiple simultaneous accesses to the same memory block



- Using $C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} B_{k,j}$ causes contention. For example, $C_{0,0}$, $C_{0,1}$, $C_{0,\sqrt{p}-1}$ attempt to read $A_{0,0}$, at the same time.
- A contention-free manner is to use:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} B_{(i+j+k)\%\sqrt{p},j}$$

All tasks $P_{*,j}$ that work on the same row of C access block $A_{i,(i+j+k)\%\sqrt{p}}$, which is different for each task.

- Overlap computations with interactions
  - Use non-blocking communication
- Replicate data or computations
  - Some parallel algorithm may have read-only access to shared data structure. If local memory is available, replicate a copy of shared data on each process if possible, so that there is only initial interaction during replication.
- Use collective interaction operations
- Overlap interactions with other interactions

Parallel Algorithm Models

- Data parallel
  - Each task performs similar operations on different data
  - Typically statically map tasks to processes
- Task graph
  - Use task dependency graph to promote locality or reduce interactions
- Master-slave
  - One or more master processes generating tasks
  - Allocate tasks to slave processes
  - Allocation may be static or dynamic
- Pipeline/producer-consumer
  - Pass a stream of data through a sequence of processes
  - Each performs some operation on it
- Hybrid
  - Apply multiple models hierarchically, or apply multiple models in sequence to different phases

- Reference
  - A. Grama, et al. Introduction to Parallel Computing. Chapter 3.