Principles of Parallel Algorithm Design

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text “Introduction to Parallel Computing”,
Addison Wesley, 2003.

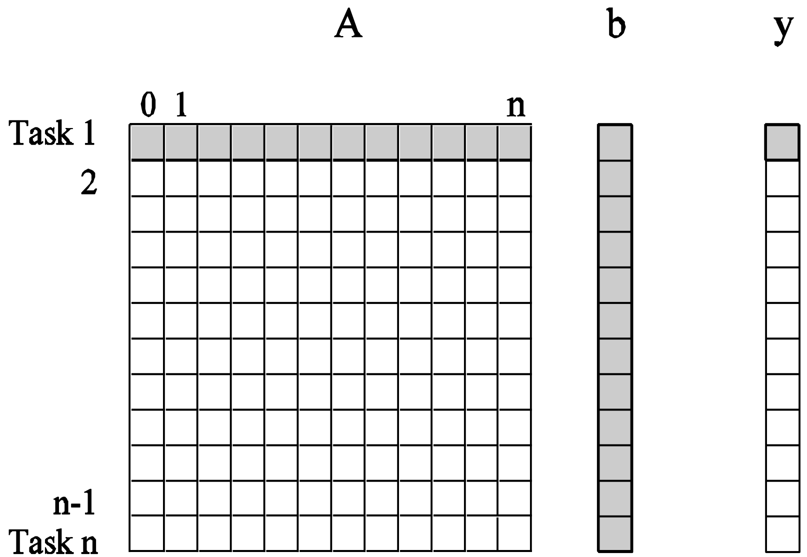
Chapter Overview: Algorithms and Concurrency

- Introduction to Parallel Algorithms
  - Tasks and Decomposition
- Decomposition Techniques
  - Recursive Decomposition
  - Data Decomposition
  - Exploratory Decomposition
  - Hybrid Decomposition
- Characteristics of Tasks
- Mapping Techniques for Load Balancing
  - Static and Dynamic Mapping

Preliminaries: Decomposition, Tasks, and Dependency Graphs

- The first step in developing a parallel algorithm is to **decompose the problem into tasks** that can be executed concurrently
- A given problem may be decomposed 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** 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: 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:

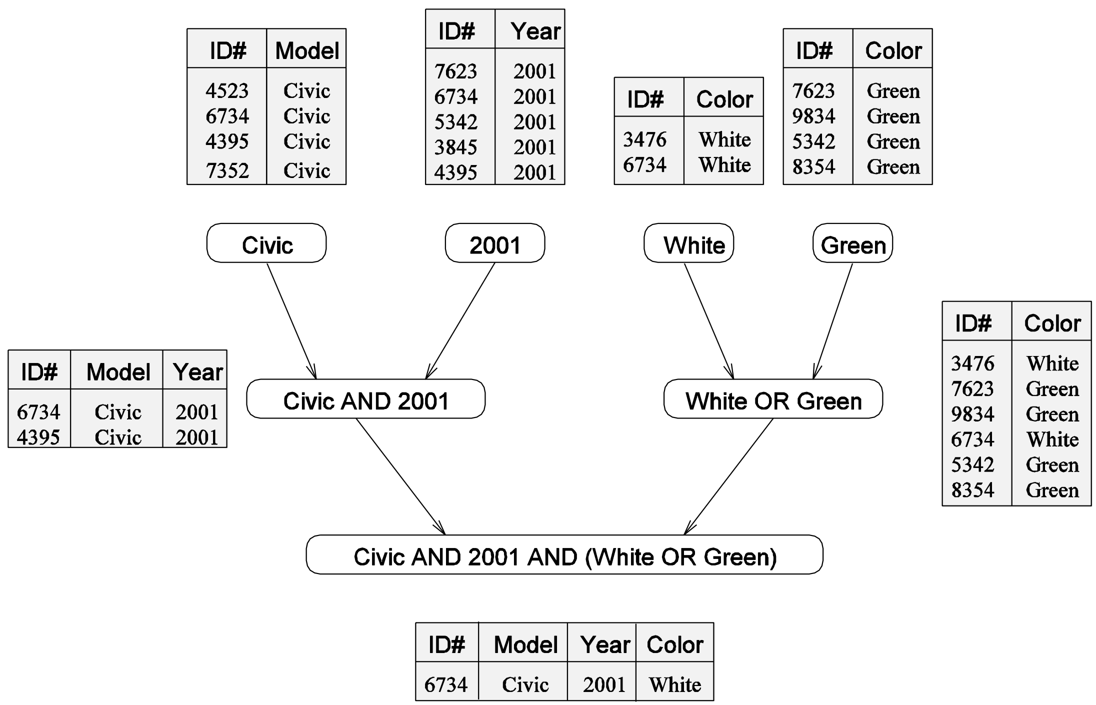
MODEL = ``CIVIC" AND YEAR = 2001 AND
(COLOR = ``GREEN" OR COLOR = ``WHITE)

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

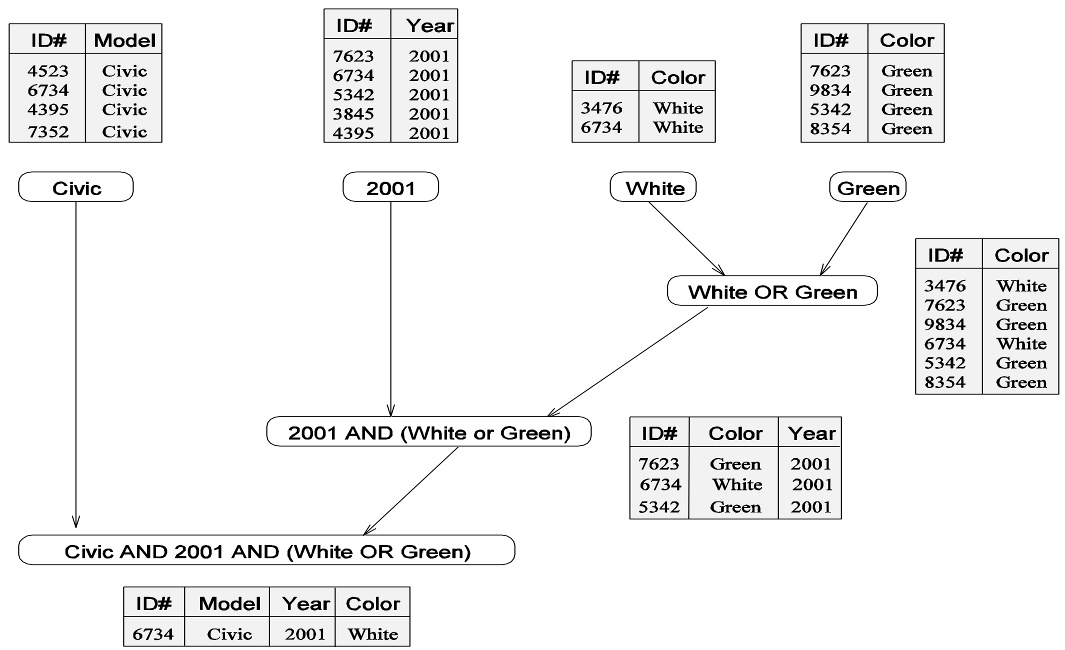
The execution of the **query can be divided into subtasks in various ways**. Each task can be thought of as generating an **intermediate table** of entries that satisfy a particular clause.



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 into subtasks in other ways as well.

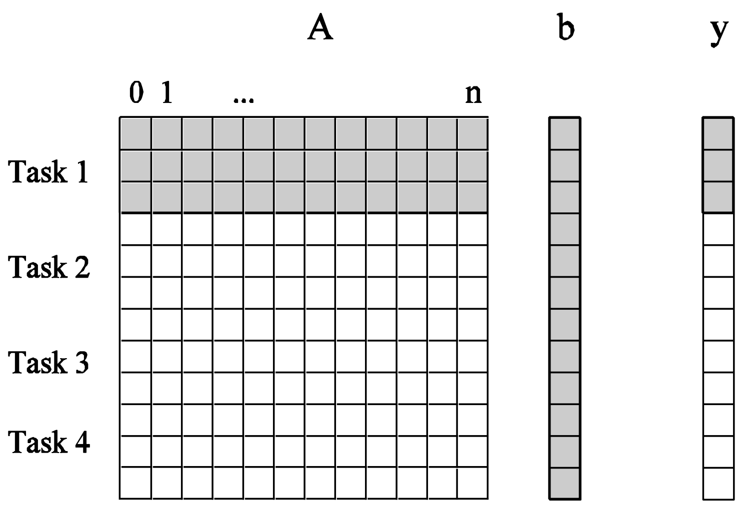


An alternate decomposition of the given problem into subtasks, along with their data dependencies.

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** and that into a small number of tasks results in a **coarse 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

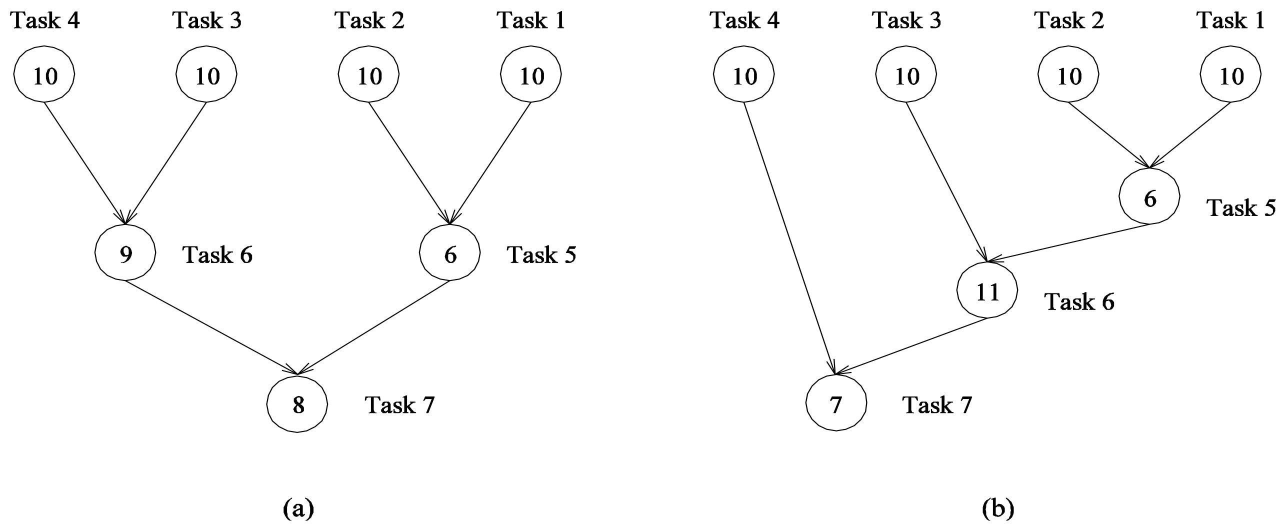
- The **number of tasks that can be executed in parallel** is the ***degree of concurrency*** of a decomposition.
- Since the number of tasks that can be executed in parallel **may change** over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution. *What is the **maximum degree of concurrency** of the database query examples?*
- The ***average degree of concurrency*** is the average number of tasks that can be processed in parallel over the execution of the program. In other words, it is a ratio of the total amount of work to the critical path length (see next slide).
- 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?

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 and average 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* $\Theta(n^2)$ *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.

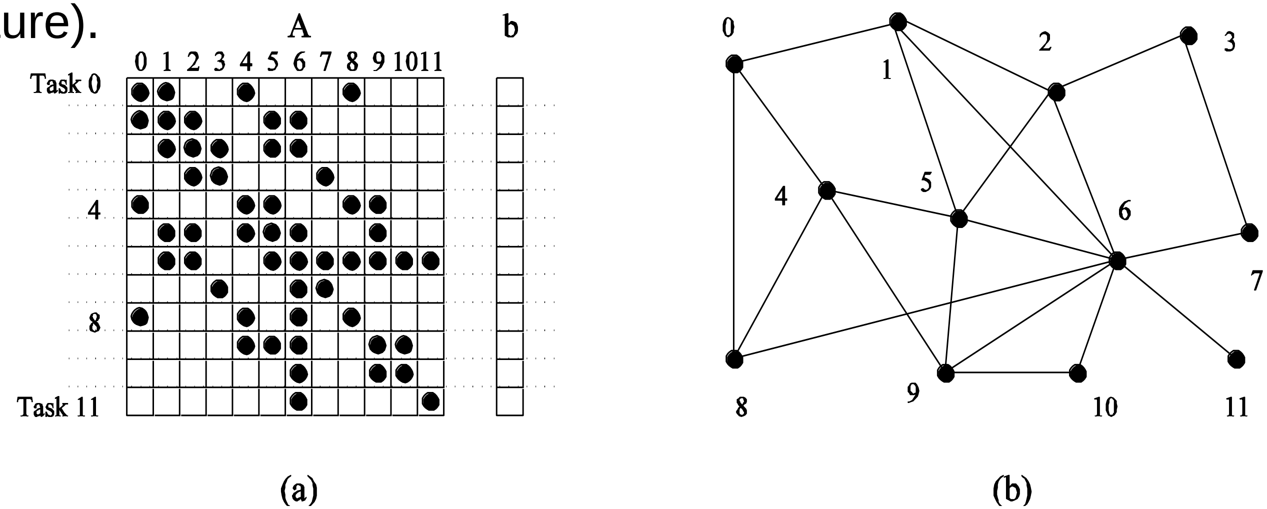
Task Interaction Graphs

- **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).



Decomposition Techniques

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

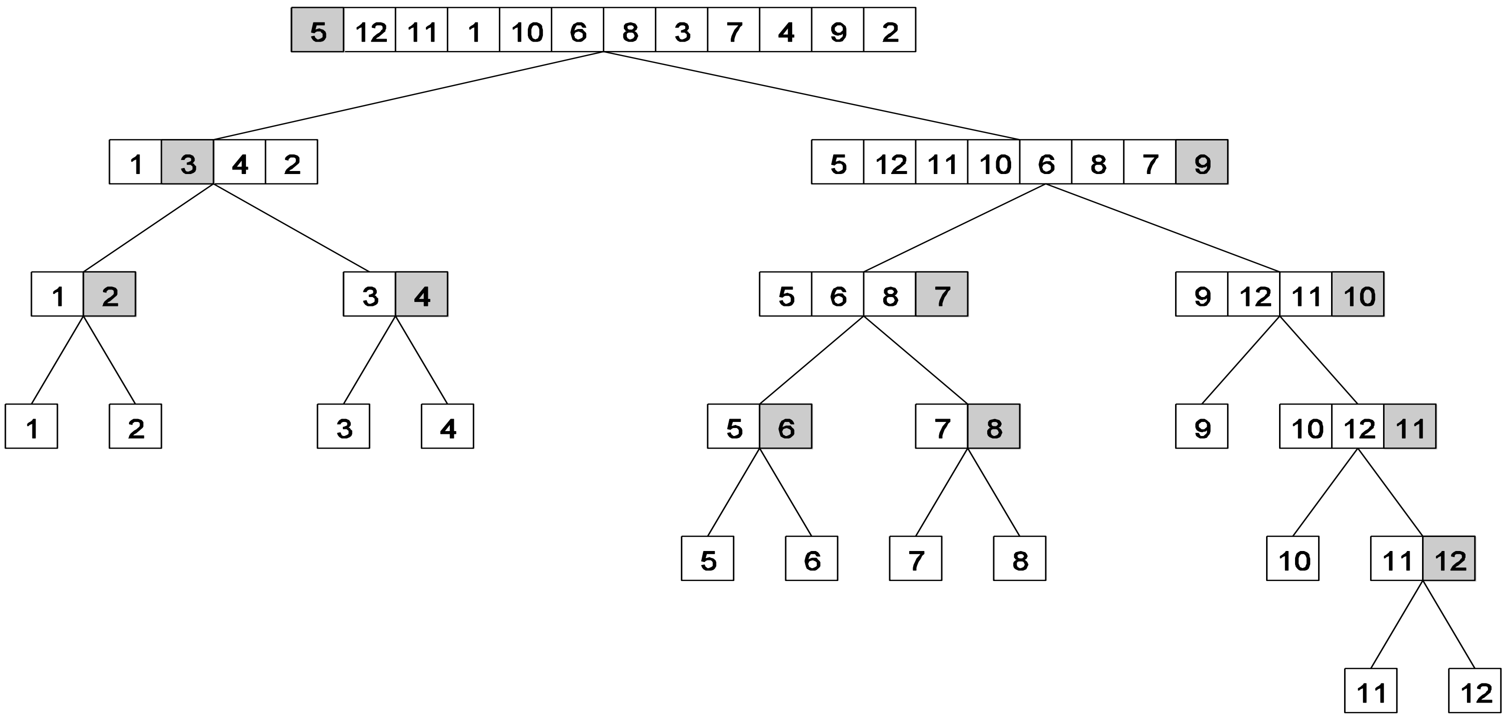
- Generally suited to problems that are solved using the **divide-and-conquer strategy**.

- 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, 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.

We first start with a **simple serial loop** for computing the minimum entry in a given list:

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

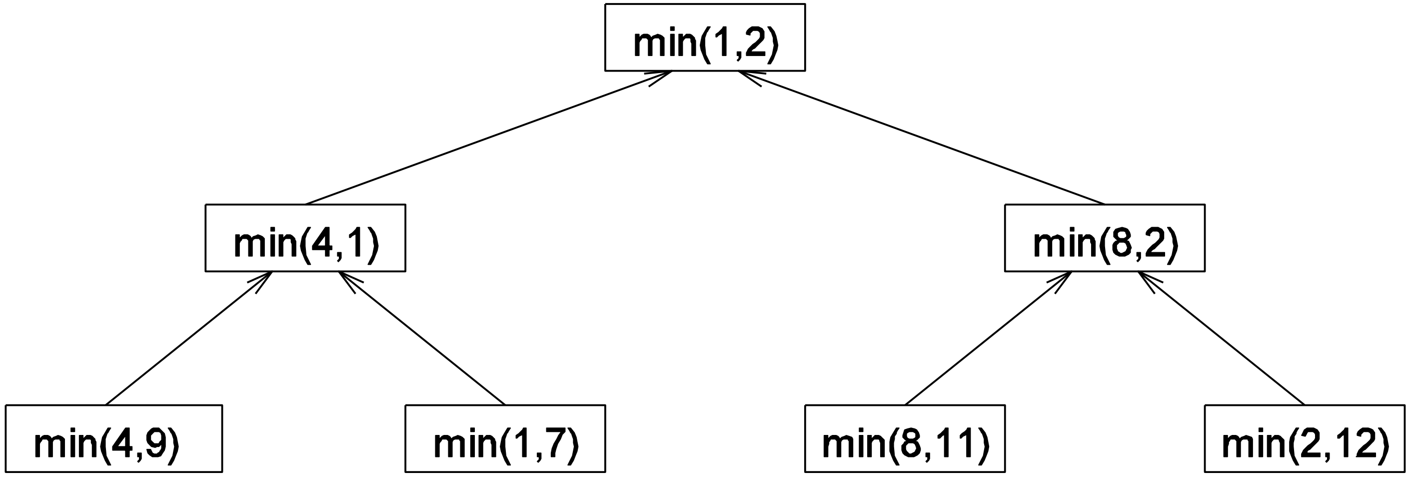
Recursive Decomposition: Example

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:**
  - Output data partitioning
  - Input data partitioning
  - Intermediate data partitioning
- The partitioning critically impacts performance of a parallel algorithm.

Data Decomposition: Output Data Decomposition

- 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: 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 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: $\boldsymbol{C}_{1,1} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,1}$	Task 1: $\boldsymbol{C}_{1,1} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,1}$
Task 2: $\boldsymbol{C}_{1,1} = \boldsymbol{C}_{1,1} + \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,1}$	Task 2: $\boldsymbol{C}_{1,1} = \boldsymbol{C}_{1,1} + \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,1}$
Task 3: $\boldsymbol{C}_{1,2} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,2}$	Task 3: $\boldsymbol{C}_{1,2} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,2}$
Task 4: $\boldsymbol{C}_{1,2} = \boldsymbol{C}_{1,2} + \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,2}$	Task 4: $\boldsymbol{C}_{1,2} = \boldsymbol{C}_{1,2} + \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,2}$
Task 5: $\boldsymbol{C}_{2,1} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,1}$	Task 5: $\boldsymbol{C}_{2,1} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,1}$
Task 6: $\boldsymbol{C}_{2,1} = \boldsymbol{C}_{2,1} + \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,1}$	Task 6: $\boldsymbol{C}_{2,1} = \boldsymbol{C}_{2,1} + \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,1}$
Task 7: $\boldsymbol{C}_{2,2} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,2}$	Task 7: $\boldsymbol{C}_{2,2} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,2}$
Task 8: $\boldsymbol{C}_{2,2} = \boldsymbol{C}_{2,2} + \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$	Task 8: $\boldsymbol{C}_{2,2} = \boldsymbol{C}_{2,2} + \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$

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.

(a) Transactions (input), itemsets (input), and frequencies (output)

Database Transactions	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

(b) Partitioning the frequencies (and itemsets) 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	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 2

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

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.

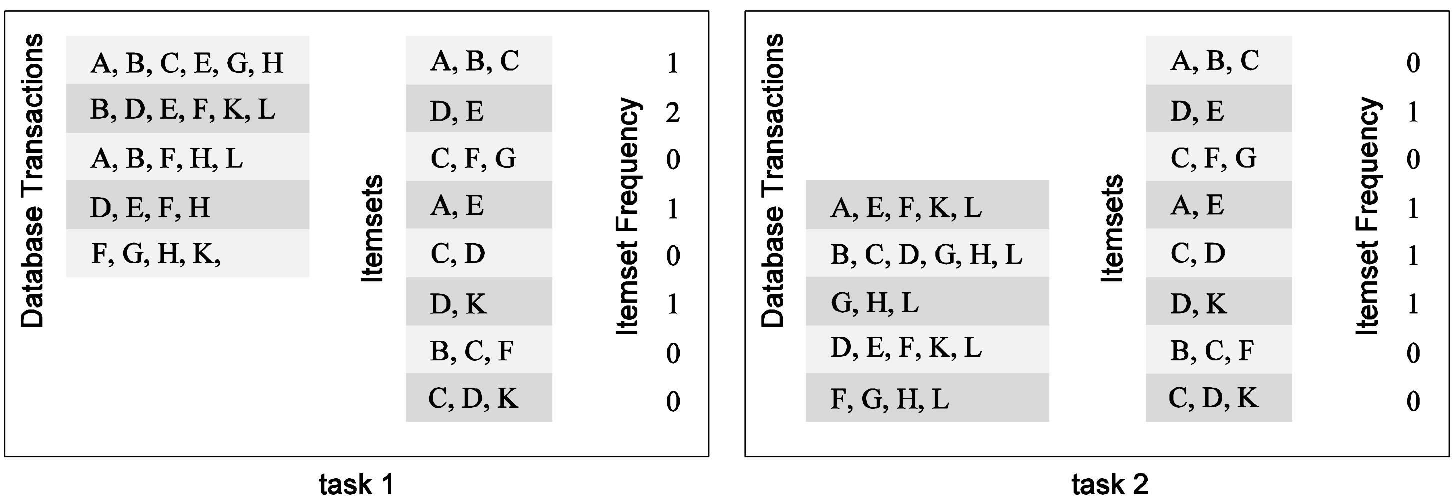
Input Data Partitioning

- 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

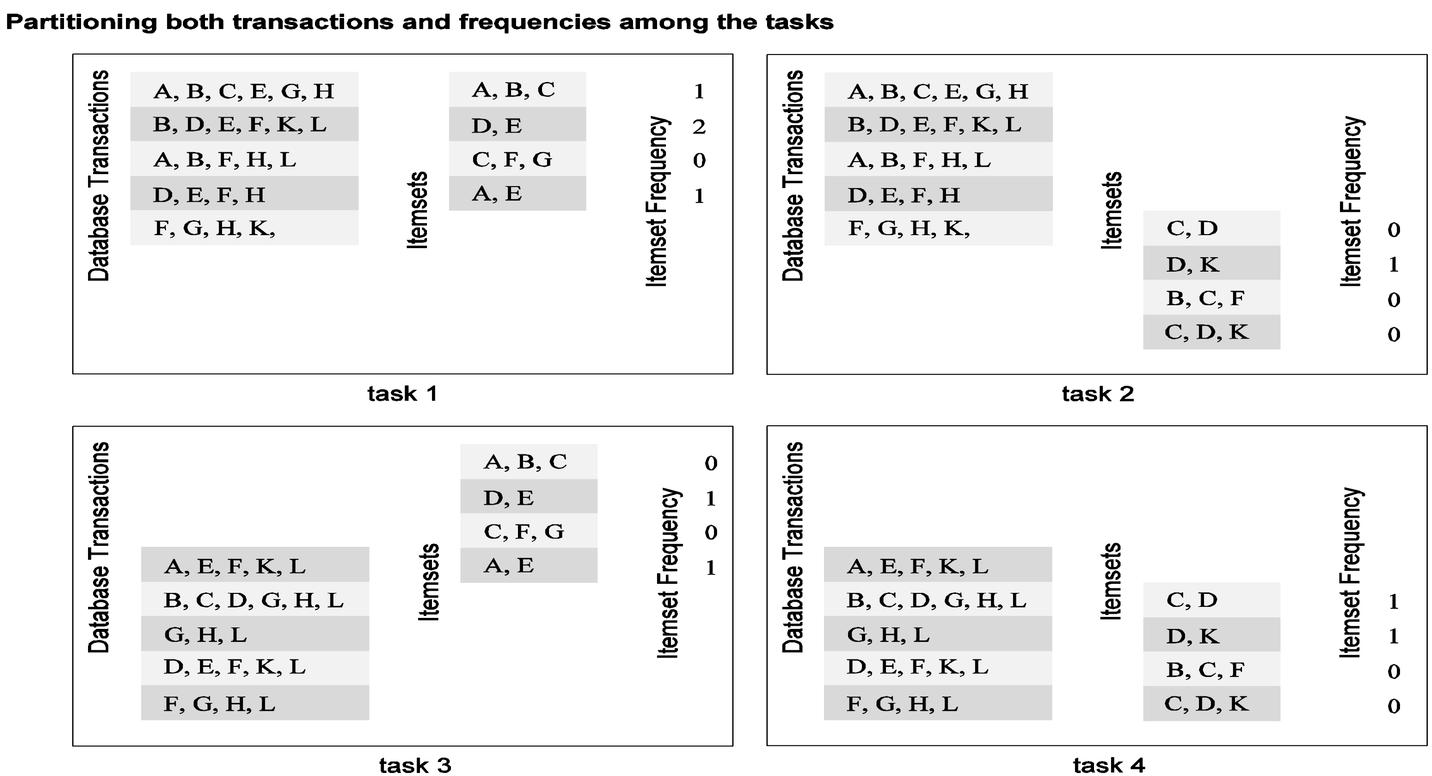
In the database counting example, the **input (i.e., the transaction set) can be partitioned**. This induces a task decomposition in which each task **generates partial counts** for all itemsets. These are combined subsequently for **aggregate counts**.

Partitioning the transactions among the tasks



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:

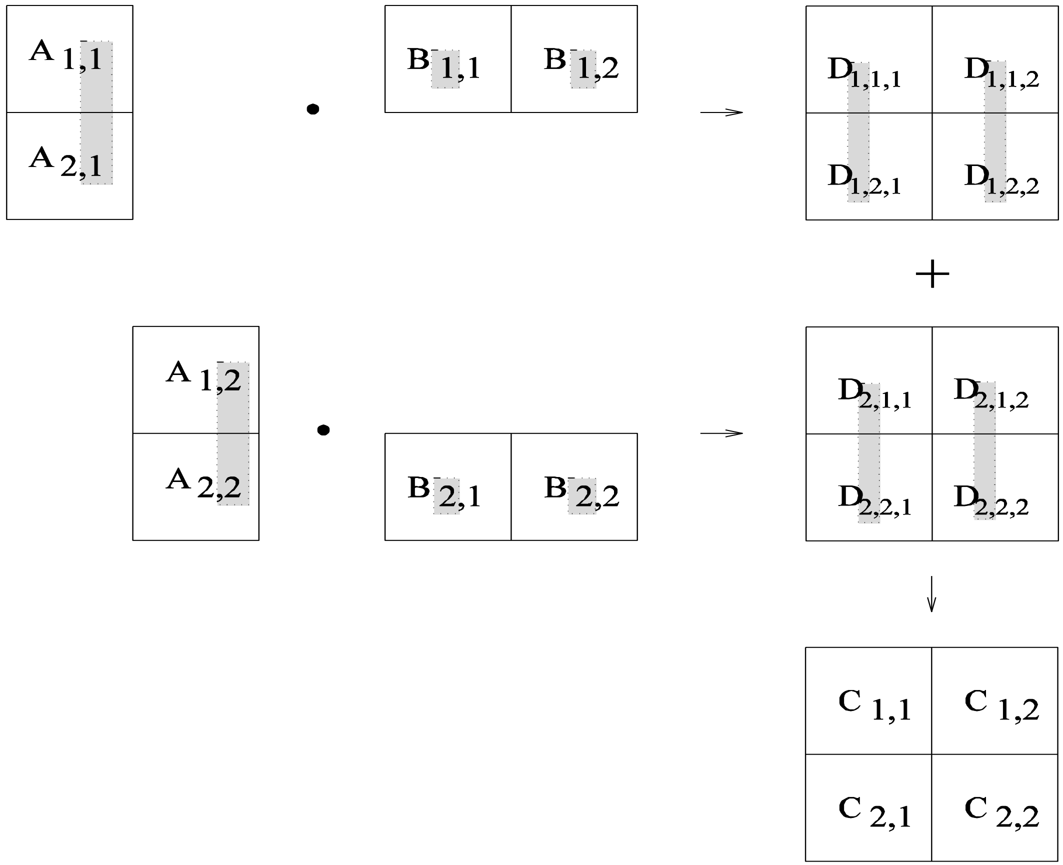


Intermediate Data Partitioning

- 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

Let us revisit the example of **dense matrix-matrix multiplication**. We first show how we can visualize this computation in terms of **intermediate matrices** ***D***.



Intermediate Data Partitioning: Example

A decomposition of intermediate data structure leads to the following decomposition into 8 + 4 tasks:

Stage I

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

Stage II

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

Task 01: $\boldsymbol{D}_{1,1,1} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,1}$

Task 02: $\boldsymbol{D}_{2,1,1} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,1}$

Task 03: $\boldsymbol{D}_{1,1,2} = \boldsymbol{A}_{1,1}\,\boldsymbol{B}_{1,2}$

Task 04: $\boldsymbol{D}_{2,1,2} = \boldsymbol{A}_{1,2}\,\boldsymbol{B}_{2,2}$

Task 05: $\boldsymbol{D}_{1,2,1} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,1}$

Task 06: $\boldsymbol{D}_{2,2,1} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,1}$

Task 07: $\boldsymbol{D}_{1,2,2} = \boldsymbol{A}_{2,1}\,\boldsymbol{B}_{1,2}$

Task 08: $\boldsymbol{D}_{2,2,2} = \boldsymbol{A}_{2,2}\,\boldsymbol{B}_{2,2}$

Task 09: $\boldsymbol{C}_{1,1} = \boldsymbol{D}_{1,1,1} + \boldsymbol{D}_{2,1,1}$

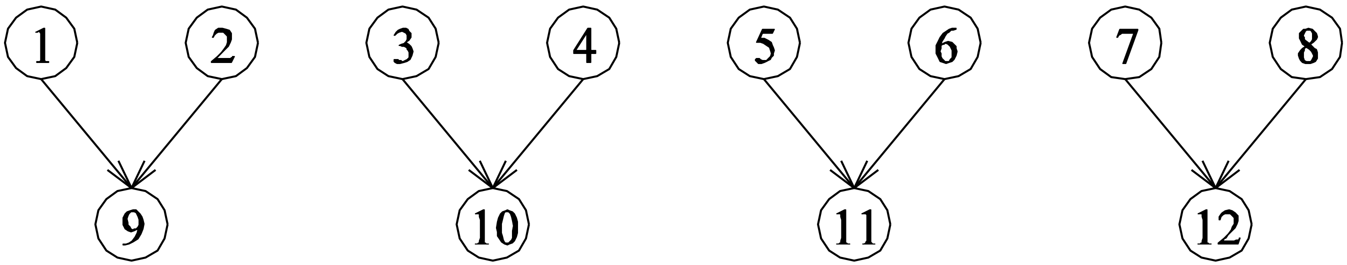
Task 10: $\boldsymbol{C}_{1,2} = \boldsymbol{D}_{1,1,2} + \boldsymbol{D}_{2,1,2}$

Task 11: $\boldsymbol{C}_{2,1} = \boldsymbol{D}_{1,2,1} + \boldsymbol{D}_{2,2,1}$

Task 12: $\boldsymbol{C}_{2,,2} = \boldsymbol{D}_{1,2,2} + \boldsymbol{D}_{2,2,2}$

Intermediate Data Partitioning: Example

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

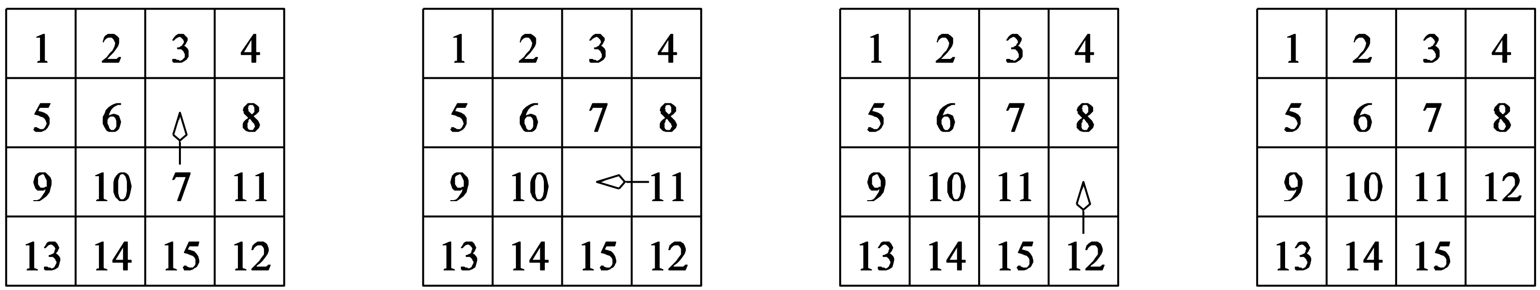


Exploratory Decomposition

- 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 transforms a given initial state (a) to desired **final state (d)**.

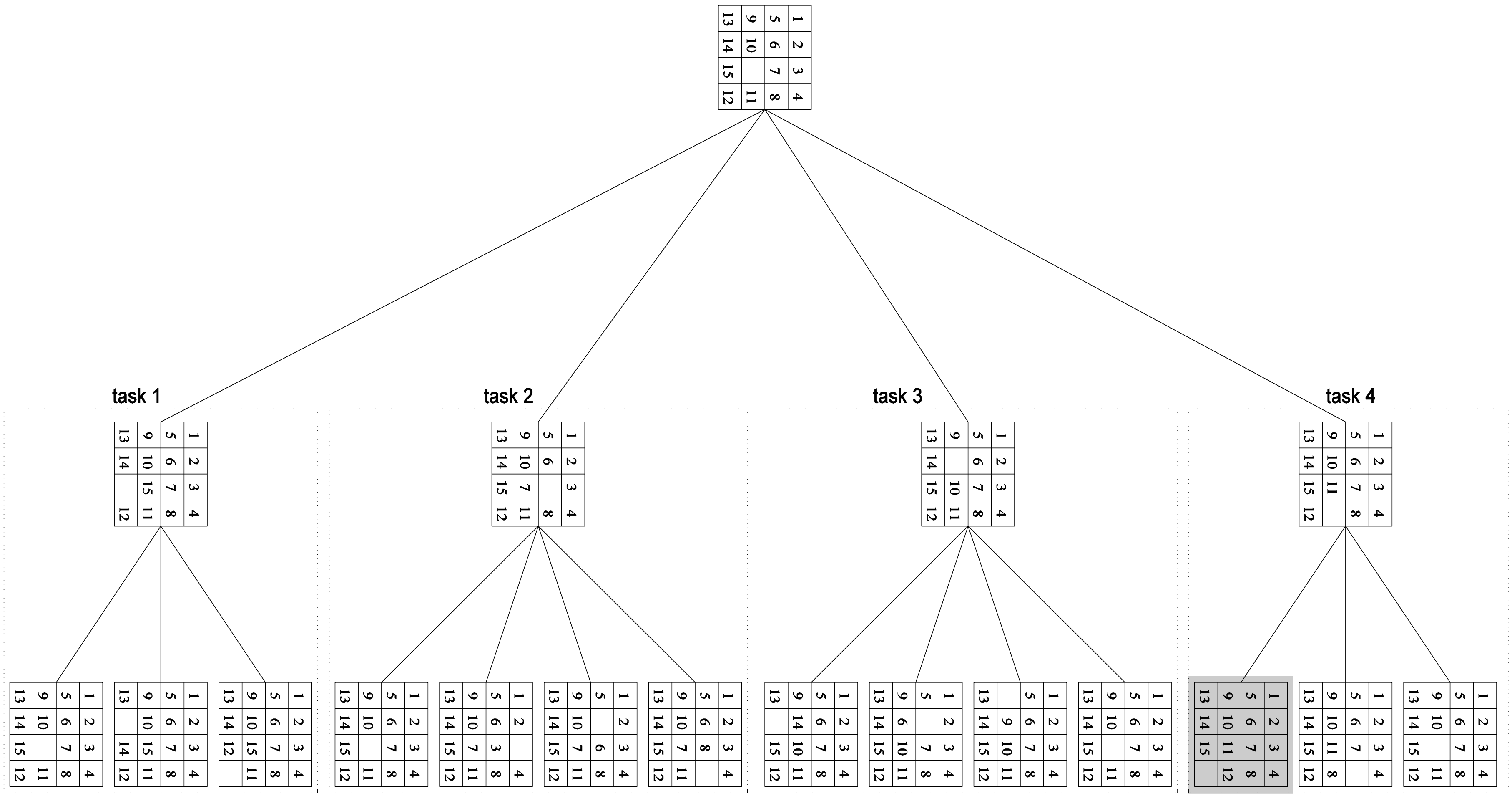


(a) (b) (c) (d)

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

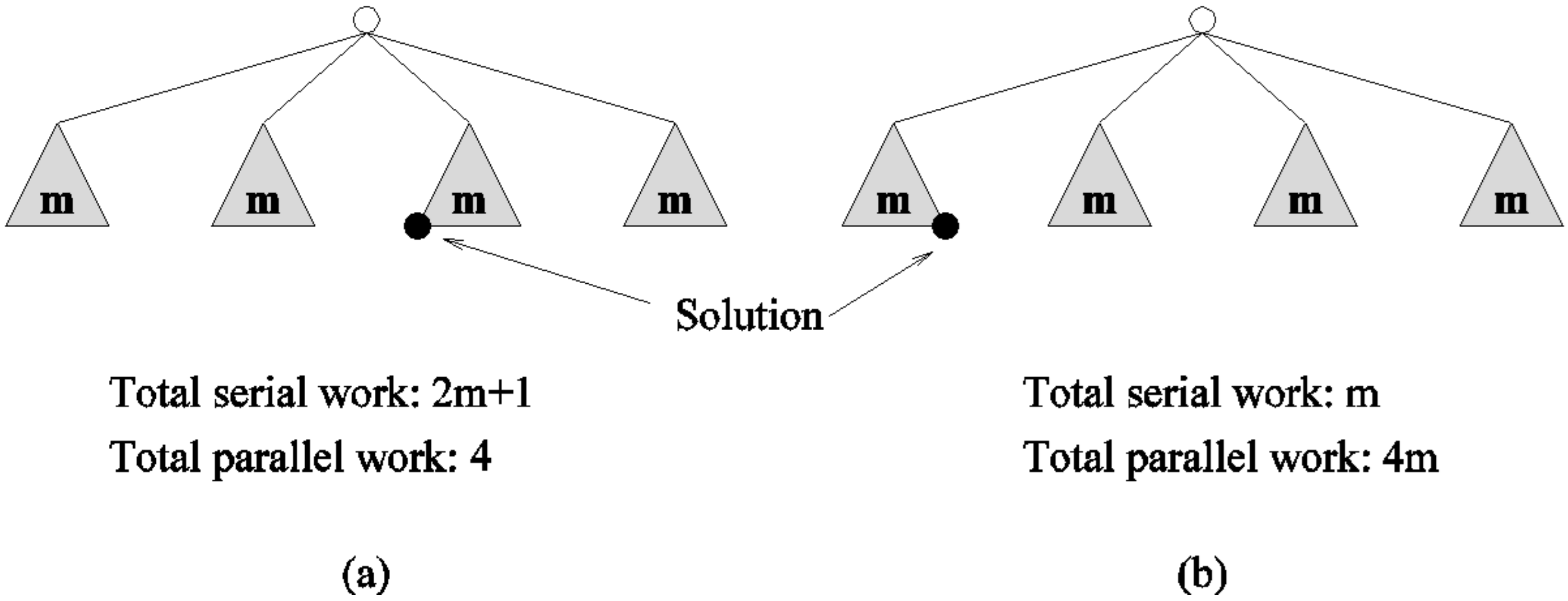
Exploratory Decomposition: Example

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



Exploratory Decomposition: Anomalous Computations

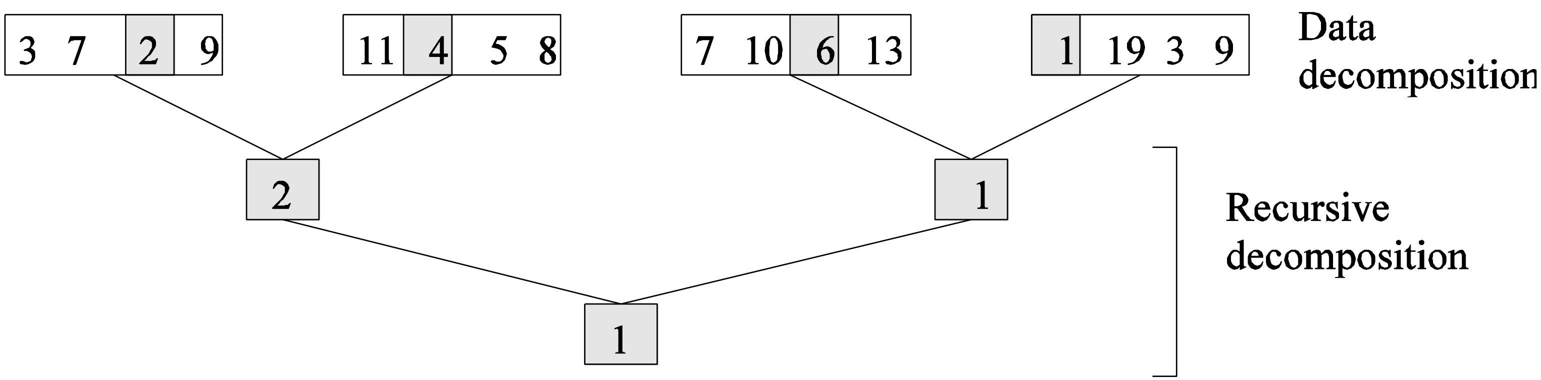
- In many instances of exploratory decomposition, the decomposition technique **may change the amount of work done** by the parallel formulation.
- This change results in **super- or sub-linear speedups**.



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.
- 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, and other **regularly structured problems** fall in this class. These can typically be decomposed using **data or recursive decomposition techniques**.
- **Dynamic task generation**: Tasks are generated as we perform computation. A classic example of this is in game playing - each 15 puzzle board is generated from the previous one. These applications are typically decomposed using **exploratory or speculative decompositions**.

Task Sizes

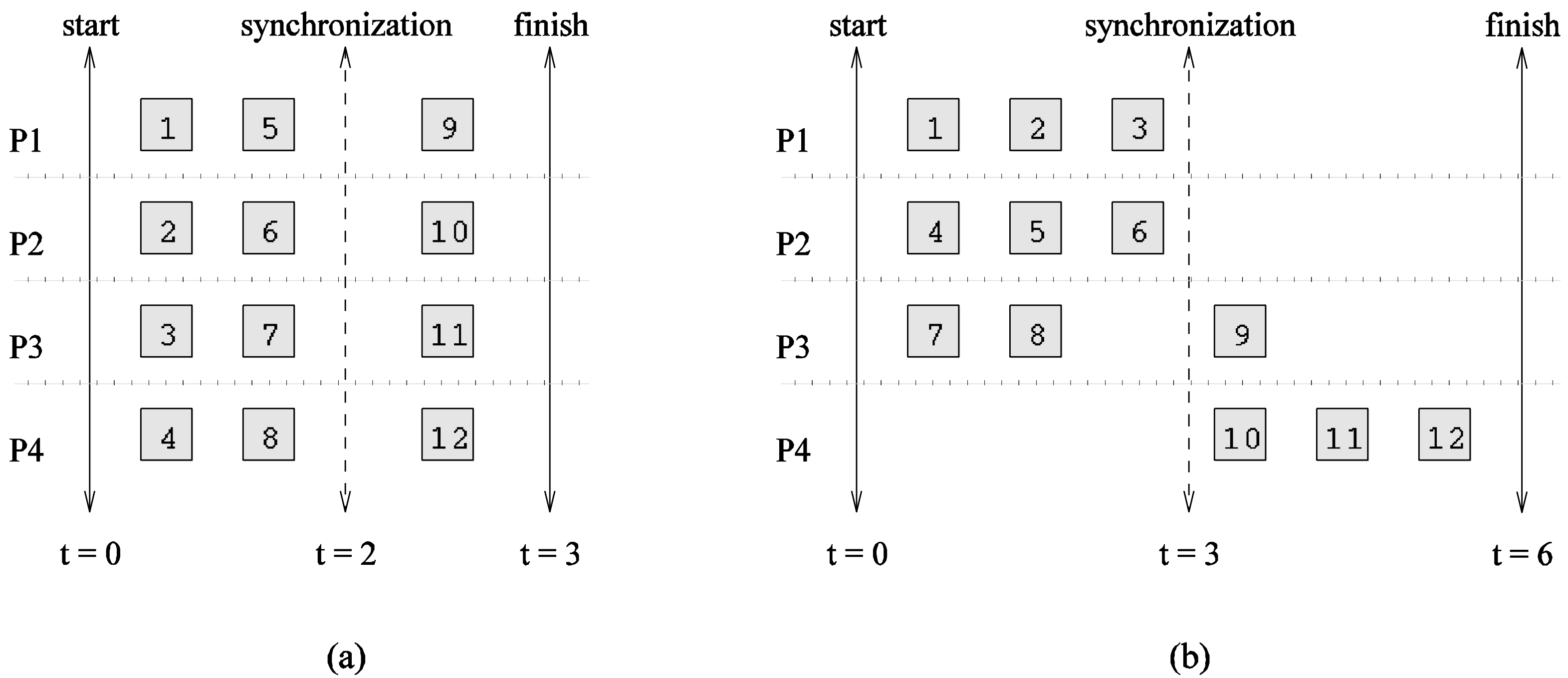
- Task sizes may be **uniform** (i.e., all tasks are the same size) or non-uniform.
- **Non-uniform** task sizes may be such that they can be determined (or estimated) a-priori or not.
- Examples in this class include discrete optimization problems, in which it is difficult to estimate the effective size of a state space.

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.

Mapping of Tasks and Complexity

Determining the optimal mapping of tasks is an **NP-complete problem** in general. Some examples:

- Mapping of tasks with dependencies on a **single processor** is solvable in polynomial time.
- Mapping of tasks **without dependencies** on a **parallel processors** (even 2) is NP-complete.
- The same problem but with **uniform task** size can be solved in polynomial time.
- If we **add dependencies** the problem becomes NP-complete.

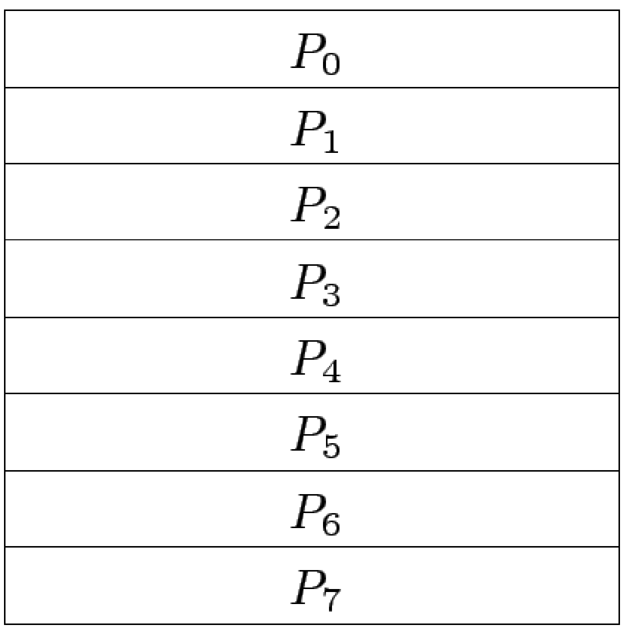
Schemes for Static Mapping

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

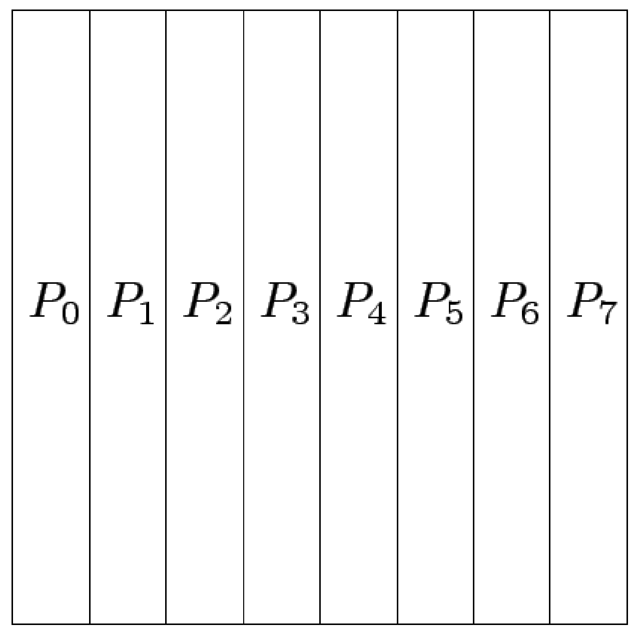
Mappings Based on Data Partitioning

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.

row-wise distribution

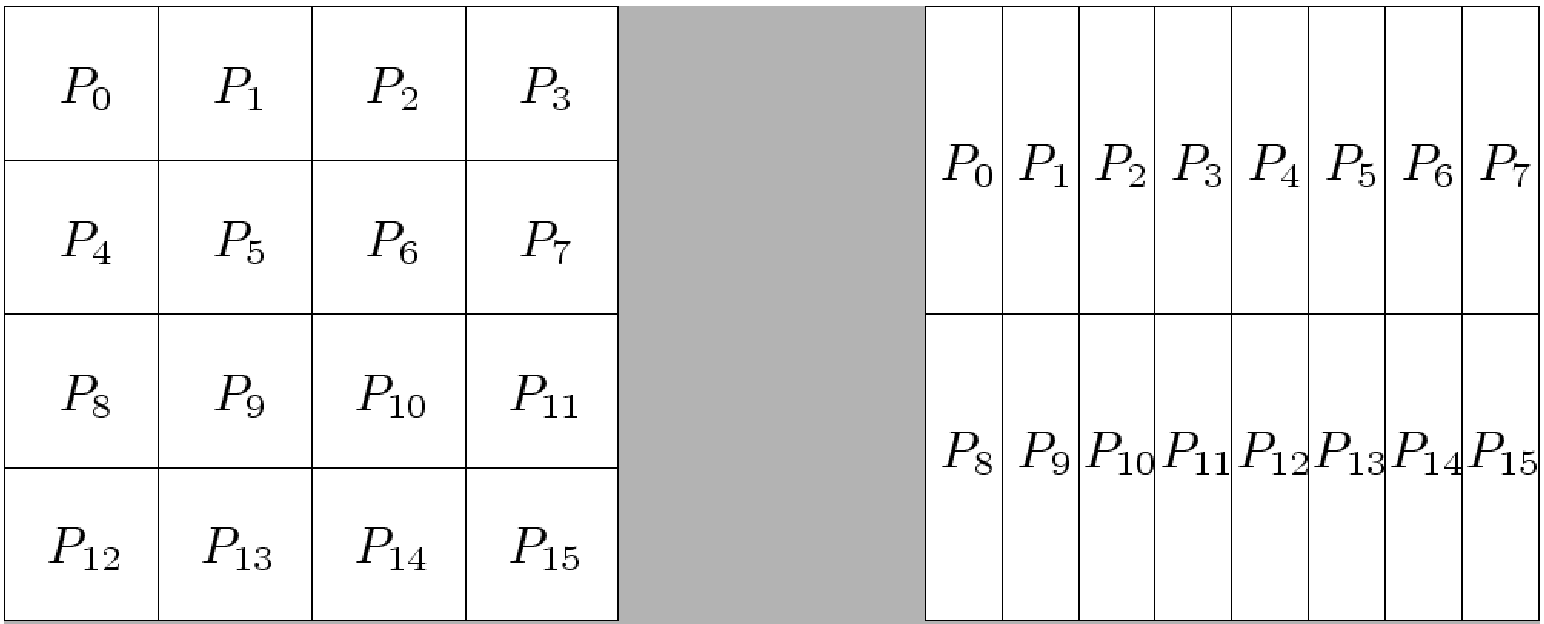


column-wise distribution



Block Array Distribution Schemes

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

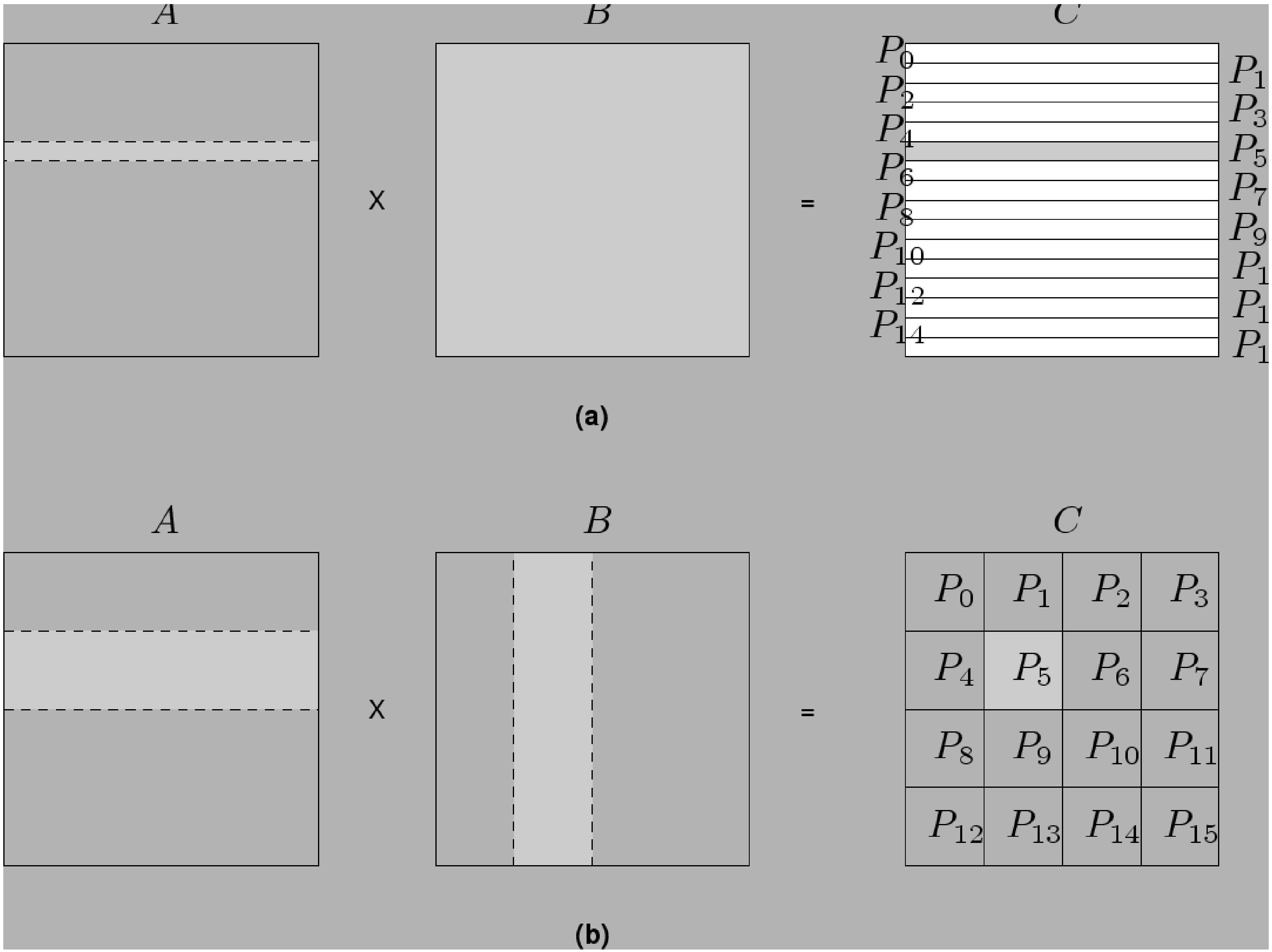


(a) **(b)**

Block Array Distribution Schemes: Examples

- For multiplying two dense matrices ***A*** and ***B***, we can partition the output matrix ***C*** using a **block decomposition**.
- For load balance, we give **each task the same number of elements** of ***C***. (Note that each element of ***C*** corresponds to a single dot product.)
- **The choice of precise decomposition (1-D or 2-D) is determined by the associated communication overhead**.
- In general, **higher dimension decomposition allows the use of larger number of processes**.

Data Sharing in Dense Matrix Multiplication



Cyclic and Block Cyclic Distributions

- If the **amount of computation associated with data items varies**, a block decomposition may lead to **significant load imbalances**.
- A simple example of this is in **LU decomposition** (or Gaussian Elimination) of dense matrices.

LU Factorization of a Dense Matrix

A decomposition of LU factorization into 14 tasks - notice the significant load imbalance.

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

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

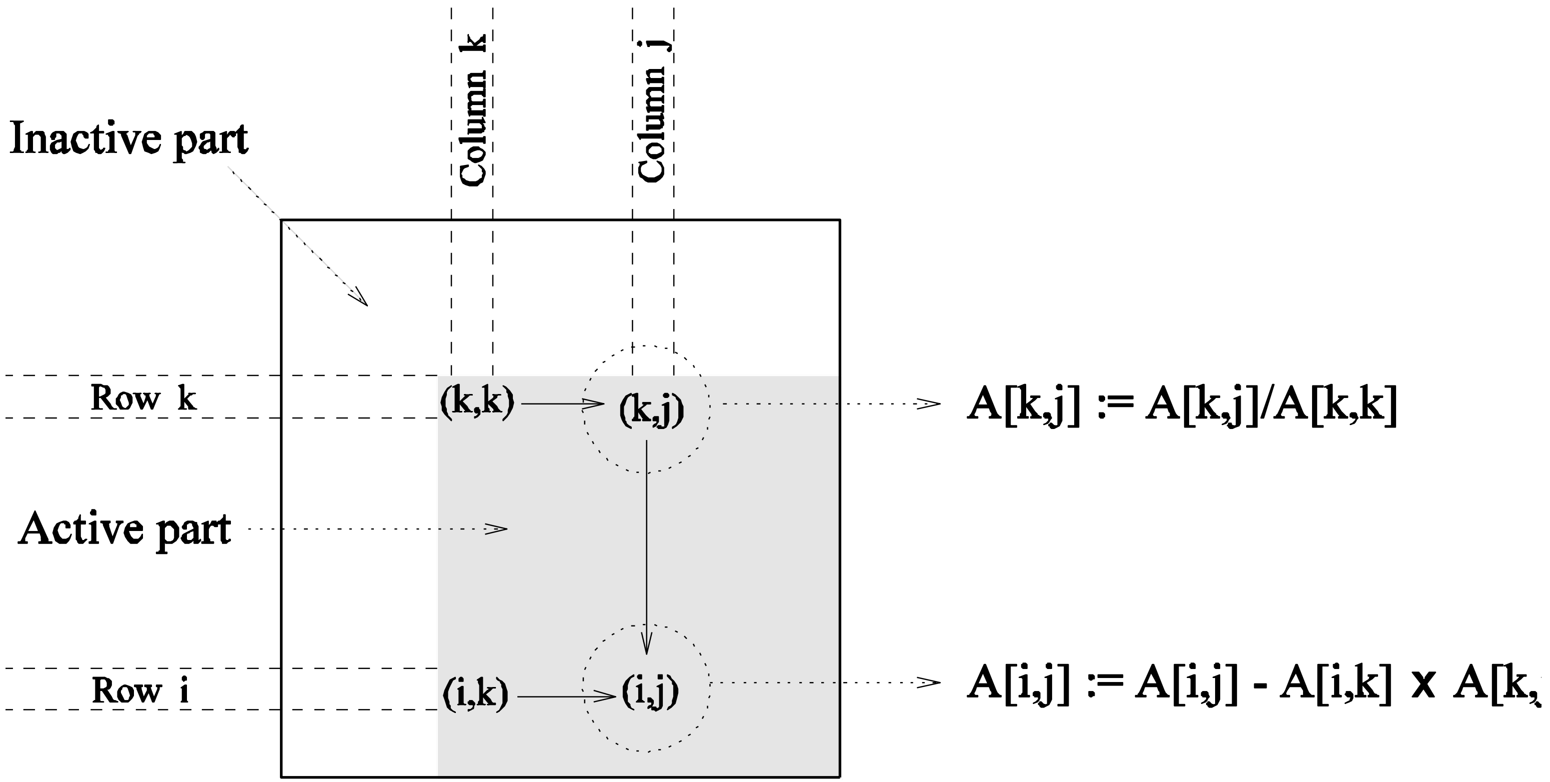
LU Factorization of a Dense Matrix

A **serial column-based algorithm** to factor a nonsingular matrix A into a lower-triangular matrix L and an upper-triangular matrix U.

```
procedure COL_LU (A)
begin
   for k := 1 to n do
       for j := k + 1 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] x 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
```

Block-Cyclic Distribution for Gaussian Elimination

The **active part** of the matrix in Gaussian Elimination changes. By assigning blocks in a block-cyclic fashion, each processor receives blocks from different parts of the matrix.

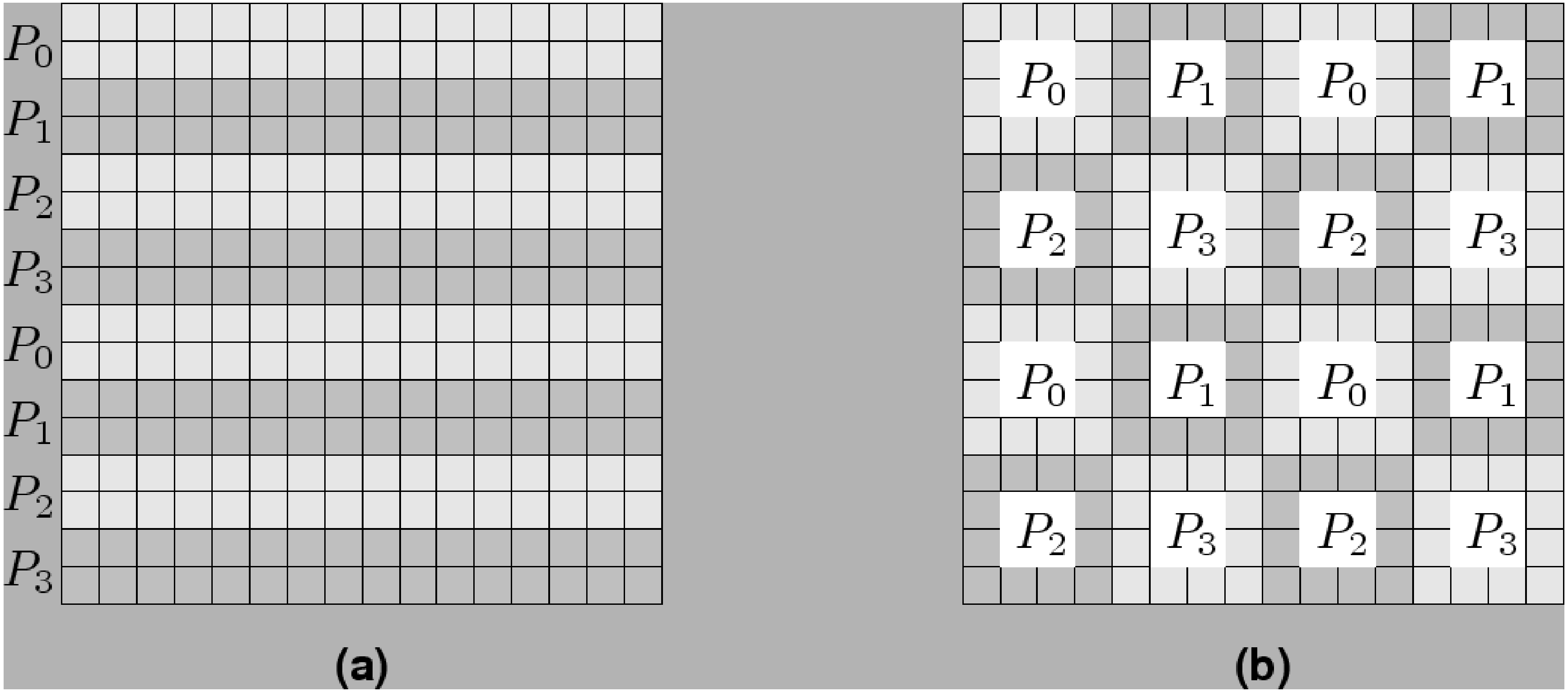


Block Cyclic Distributions

- Variation of the **block distribution scheme that can be used to alleviate the load-imbalance** and idling problems.
- Partition an array into many **more blocks than the number of available processes**.
- Blocks are assigned to processes in a **round-robin manner** so that each **process gets several non-adjacent blocks**.

Block-Cyclic Distribution

- A **cyclic distribution** is a special case in which block size is one.
- A **block- cyclic distribution** is a case in which block size is ***n/p***, where ***n*** is the dimension of the matrix and ***p*** is the number of processes.

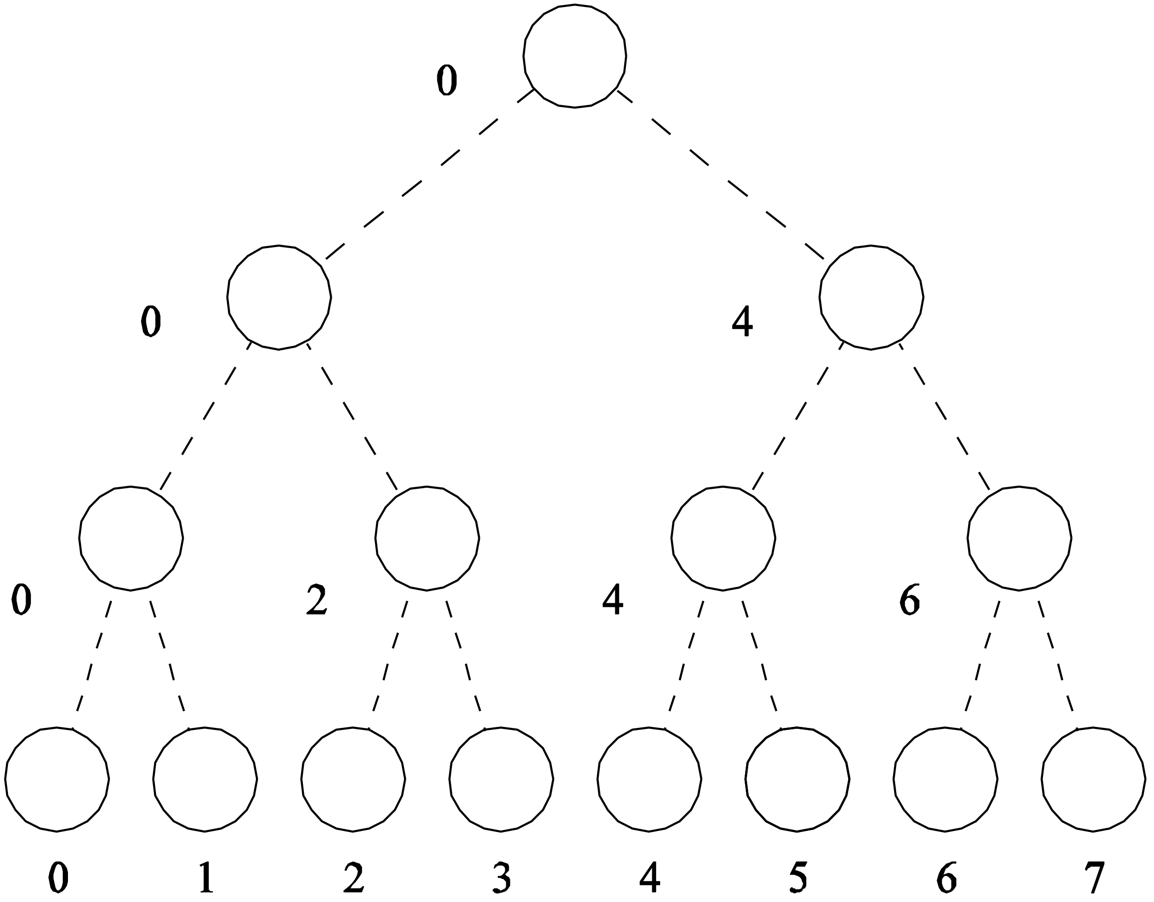


Mappings Based on Task Paritioning

- Partitioning a given task-dependency/task-interaction graph across processes.
- Determining an optimal mapping for a general task-dependency/task-interaction graph is an **NP-complete problem**.
- Excellent **heuristics** exist for structured graphs.

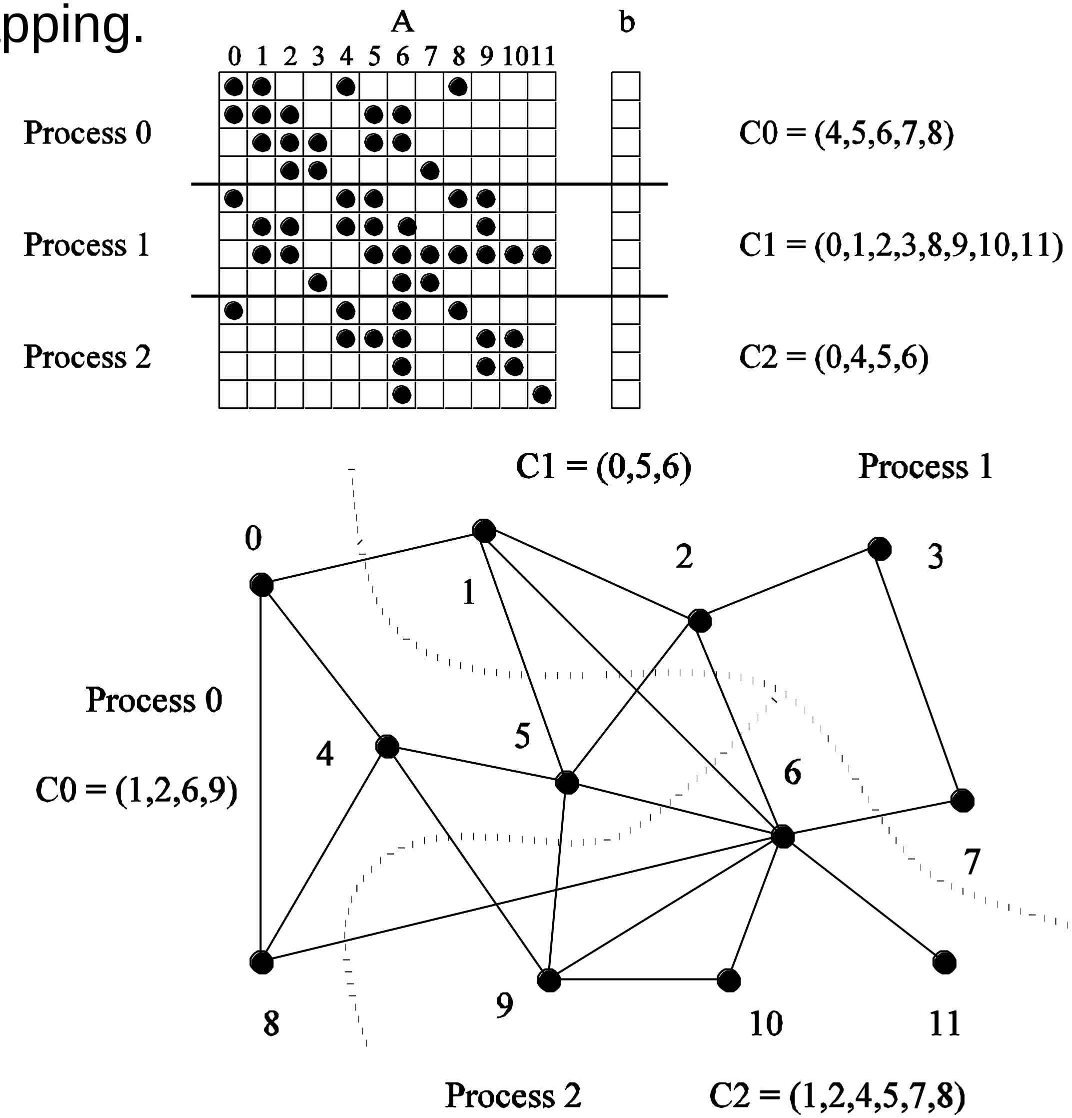
Task Paritioning: Mapping a Binary Tree Dependency Graph

Example illustrates the **dependency graph of one view of quick-sort** and how it can be assigned to processes in a hypercube.



Task Paritioning: Mapping a Sparse Graph

Sparse graph for computing a sparse matrix-vector product and its mapping.

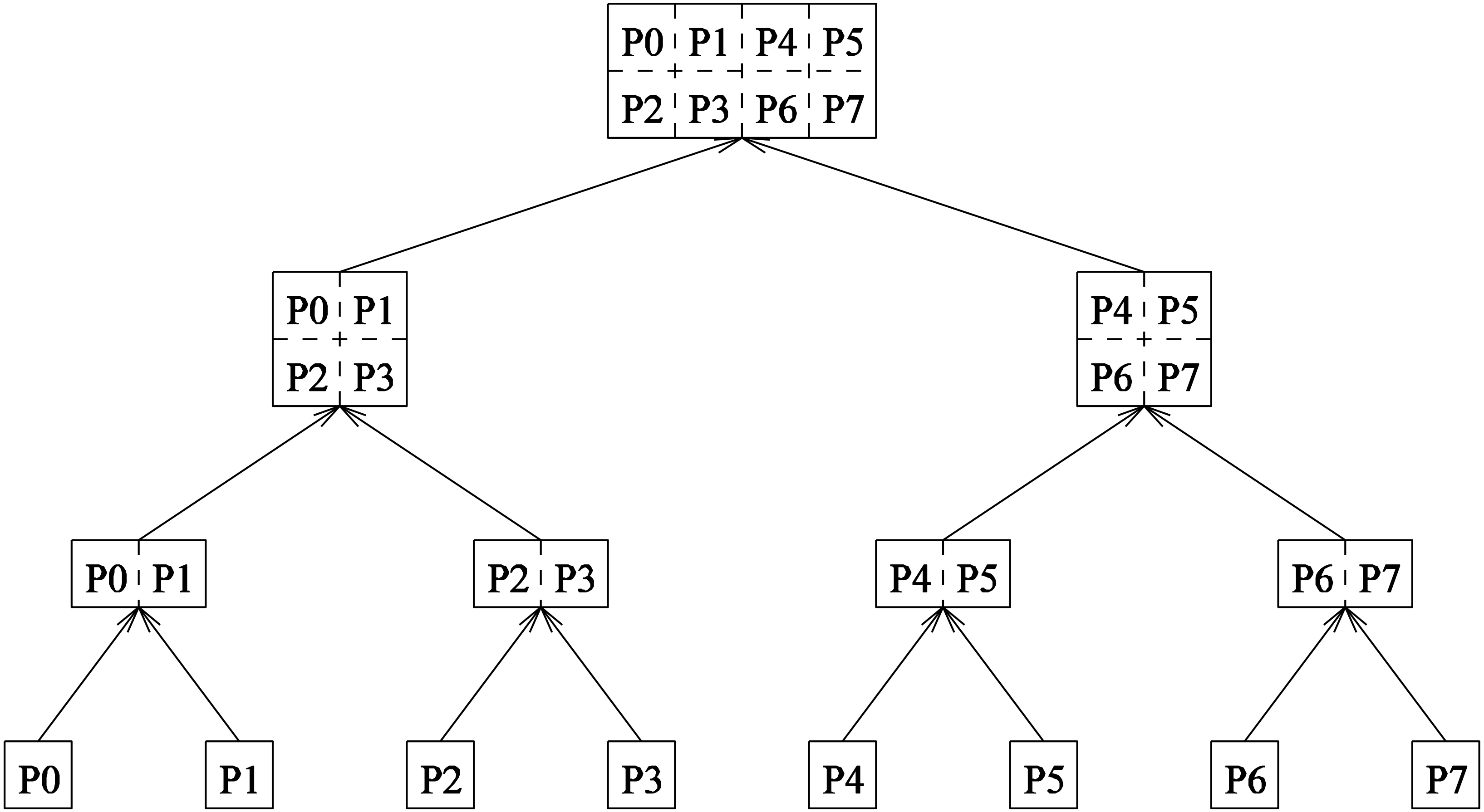


Hierarchical Mappings

- **Sometimes a single mapping technique is inadequate**.
- For example, the task mapping of the binary tree (quicksort) cannot use a large number of processors.
- For this reason, **task mapping can be used at the top level** and **data partitioning within each level**.

Hierarchical Mapping

An example of task partitioning at top level with data partitioning at the lower level.



Schemes for Dynamic Mapping

- Dynamic mapping is sometimes also referred to as **dynamic load balancing**, since load balancing is the primary motivation for dynamic mapping.
- Dynamic mapping schemes can be **centralized** or **distributed**.

Centralized Dynamic Mapping

- Processes are designated as **masters or slaves**.
- When a process runs out of work, it **requests the master for more work**.
- When the number of processes increases, the **master may become the bottleneck**.
- To alleviate this, a process may pick up a number of tasks (a chunk) at one time. This is called Chunk scheduling.
- Selecting large chunk sizes may lead to significant load imbalances as well.
- A number of schemes have been used to gradually decrease chunk size as the computation progresses.

Distributed Dynamic Mapping

- Each process can send or **receive work from other processes**.
- This **alleviates the bottleneck** in centralized schemes.
- There are four critical questions: how are sensing and receiving processes paired together, who initiates work transfer, how much work is transferred, and when is a transfer triggered?
- Answers to these questions are generally **application specific**. We will look at some of these techniques later in this class.