Lecture 4: Principles of Parallel Algorithm Design (part 2)

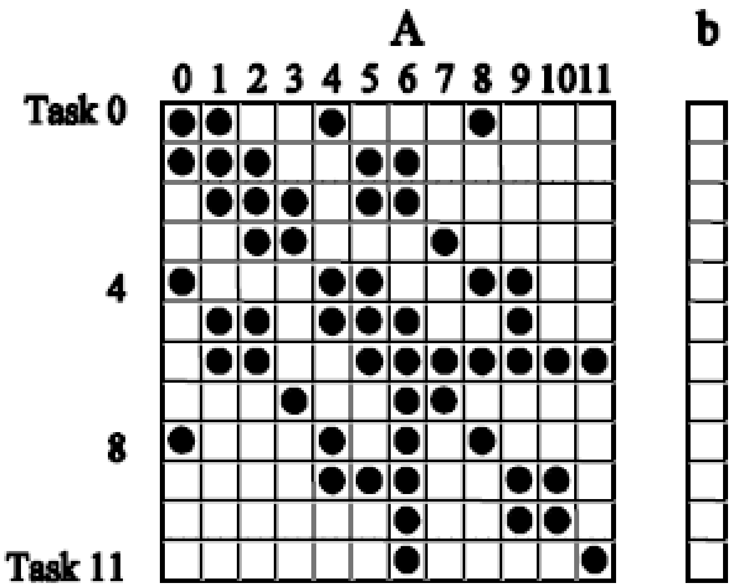
Task Interaction Graphs

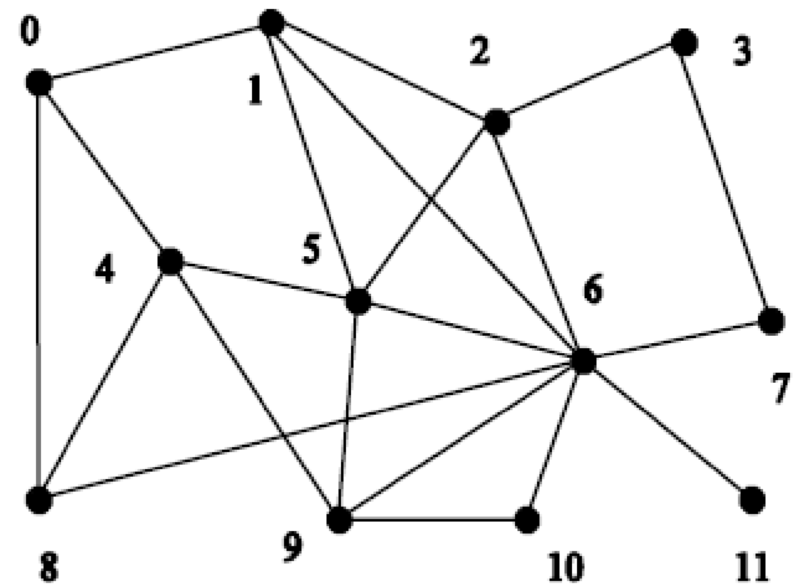
- Tasks generally share input, output or intermediate data
  - Ex. Matrix-vector multiplication: originally there is only one copy of *b*, tasks will have to communicate *b*.
- **Task-interaction graph**
  - Node = task
  - Edge(undirected/directed) = interaction or data exchange
- Task-dependency graph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

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*. <u>Also assign b[i] to Task *i*.</u>





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

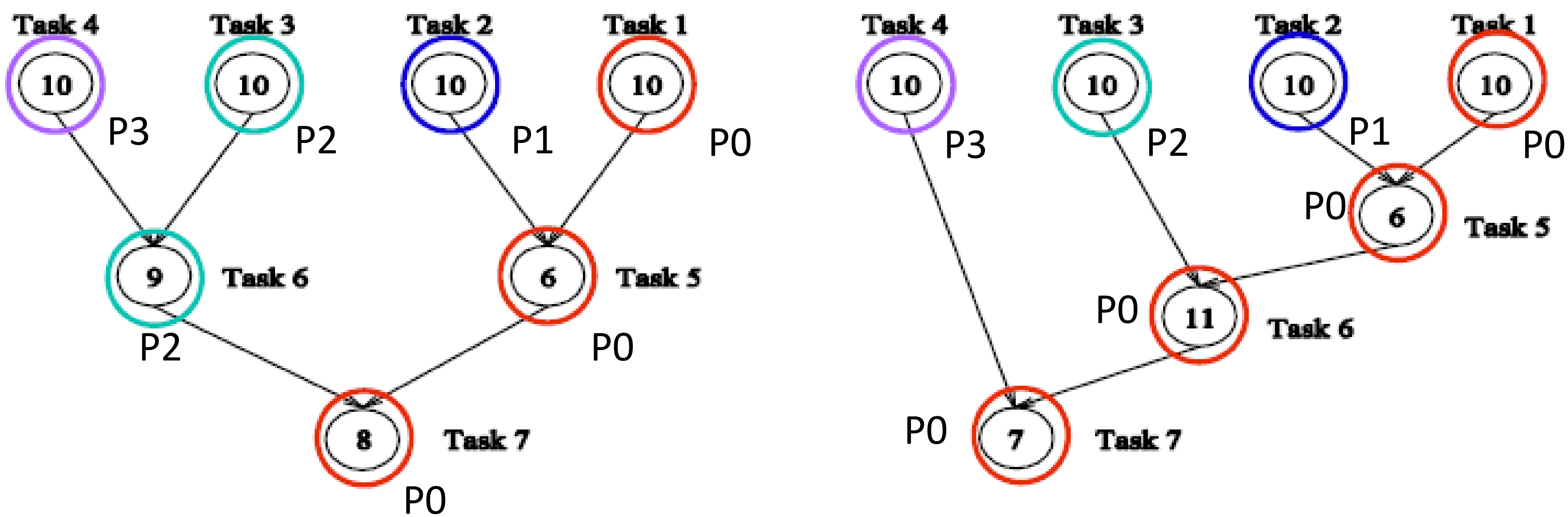
Task-dependency graph

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?

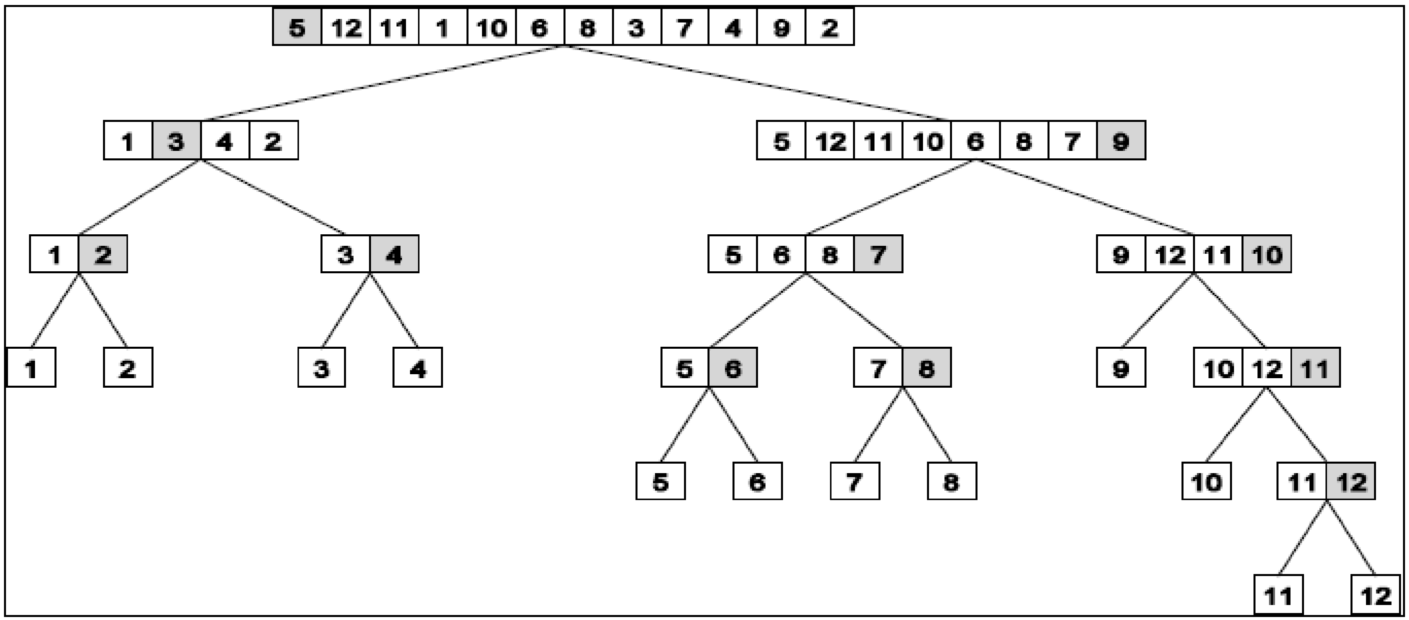
- Recursive decomposition
- Data decomposition
- Exploratory decomposition
- Speculative decomposition

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

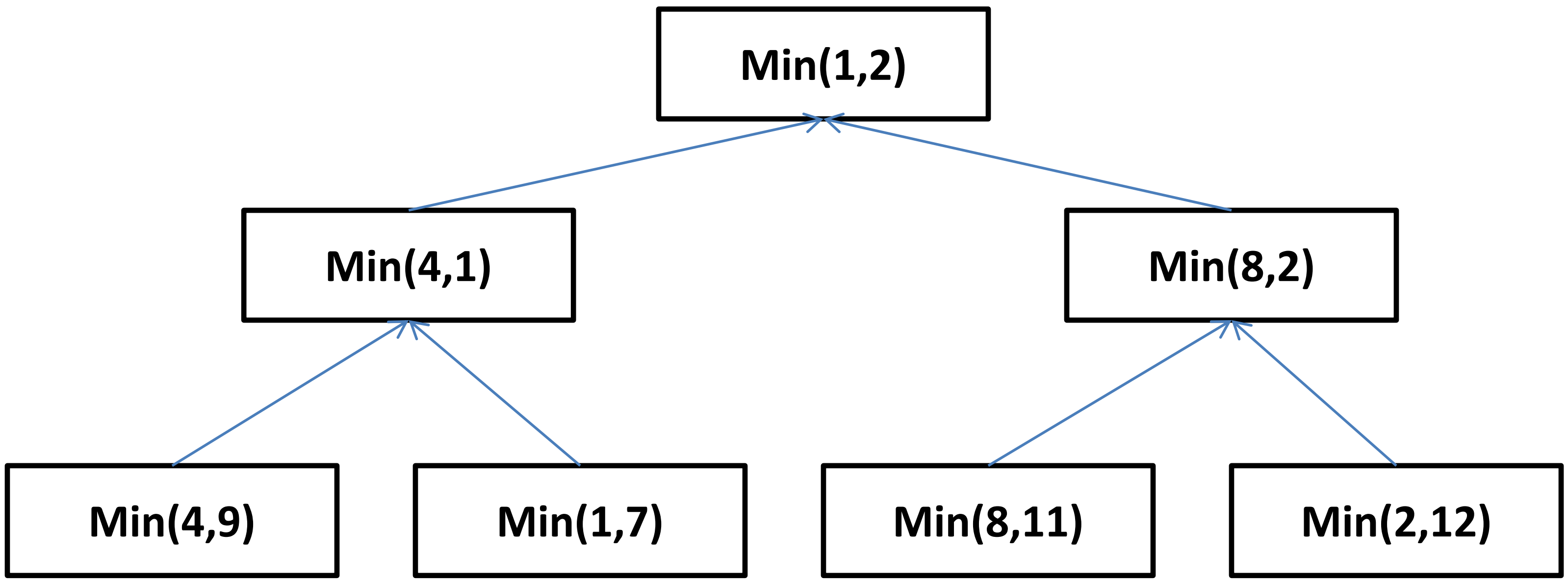
Find the minimum in an array of numbers A of length *n*

```
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
```

Task-Dependency Graph for Recursive_MIN

- Let the task be finding the minimum of two numbers.
- Example. Find min. from {4,9,1,7,8,11,2,12}



Data Decomposition

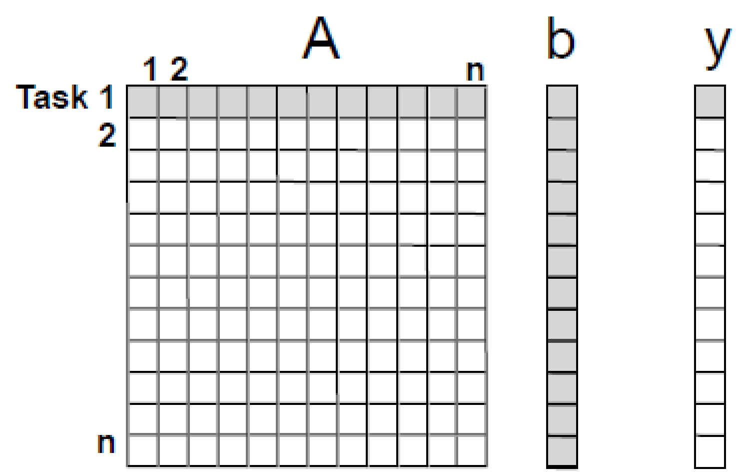
Steps

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

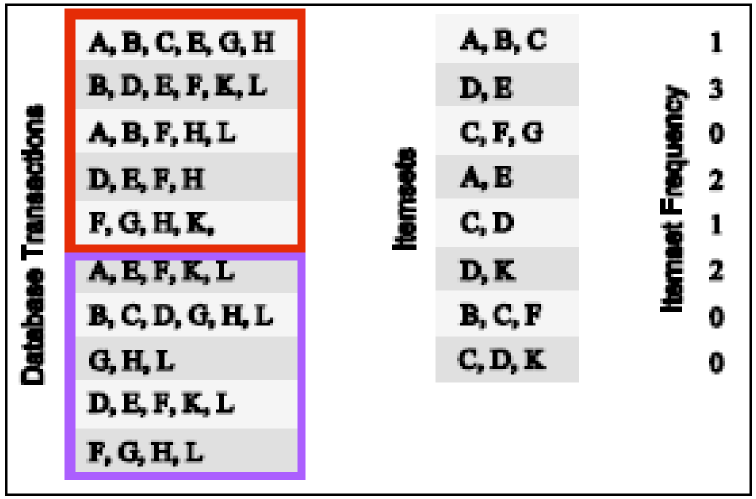
Remark: data-decomposition is different from task decomposition.
Same data decomposition can have different task decompositions.

Data Decomposition Based on Partitioning Input Data

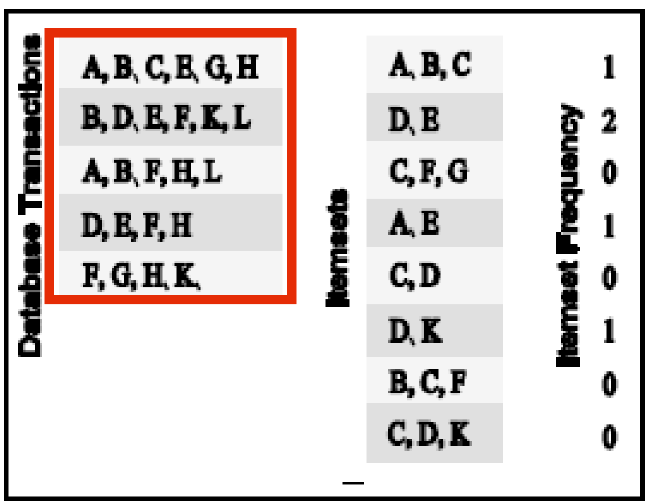
- Ideal if output is a single unknown value or the individual elements of the output can not be efficiently determined in isolation.
  - Example. Finding the minimum, maximum, or sum of a set of numbers.
  - Example. Sorting a set.
- Partitioning the input data and associating a task with each partition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

Example: Input Data Decomposition

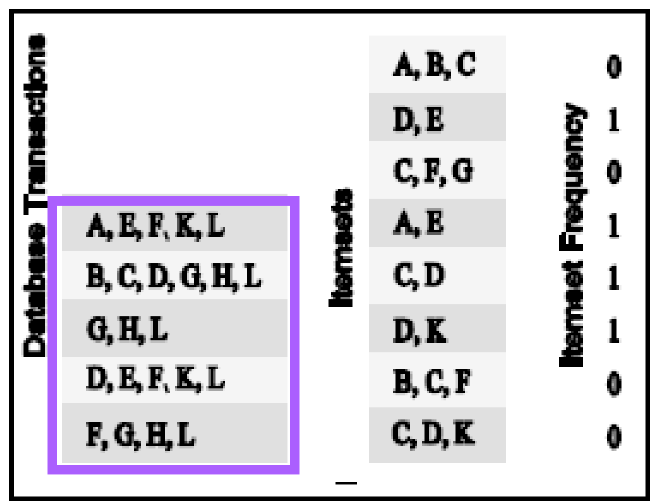
Count the frequency of itemsets in database transactions.



Partition computation by partitioning the set of transactions



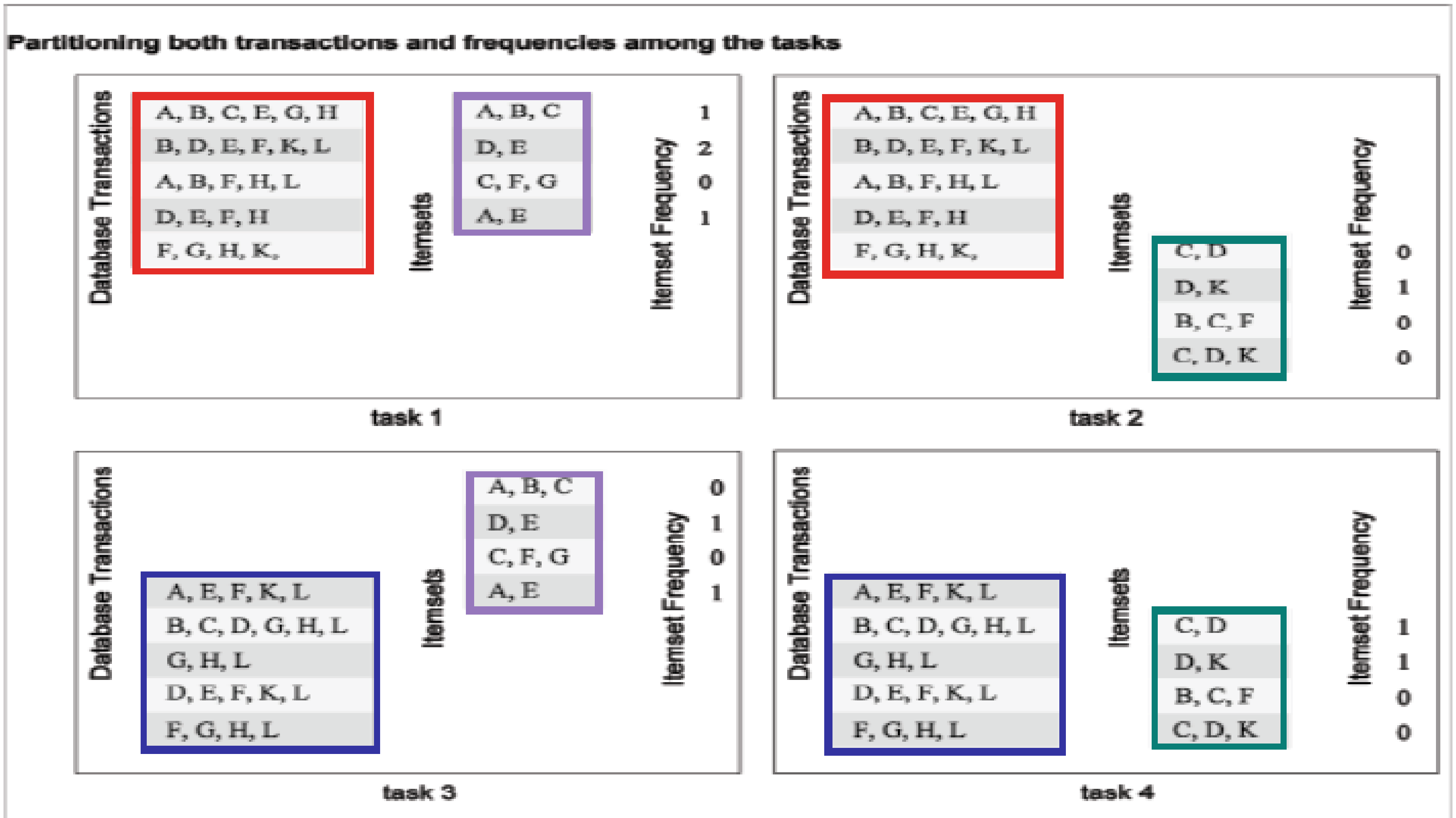
task 1



task 2

Data Decomposition Based on Partitioning Input and Output Data

Partitioning both input and output data to achieve additional concurrency



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.

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

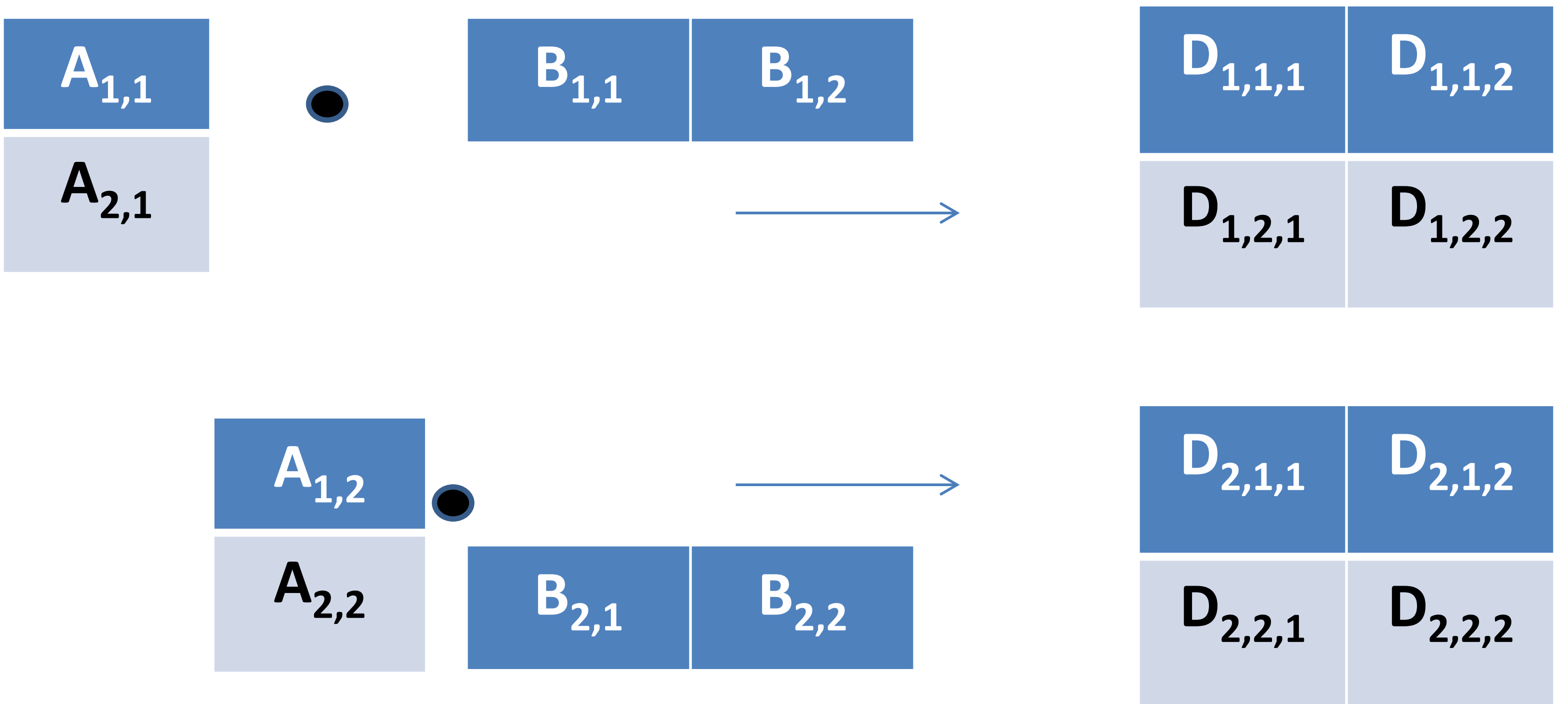
Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

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

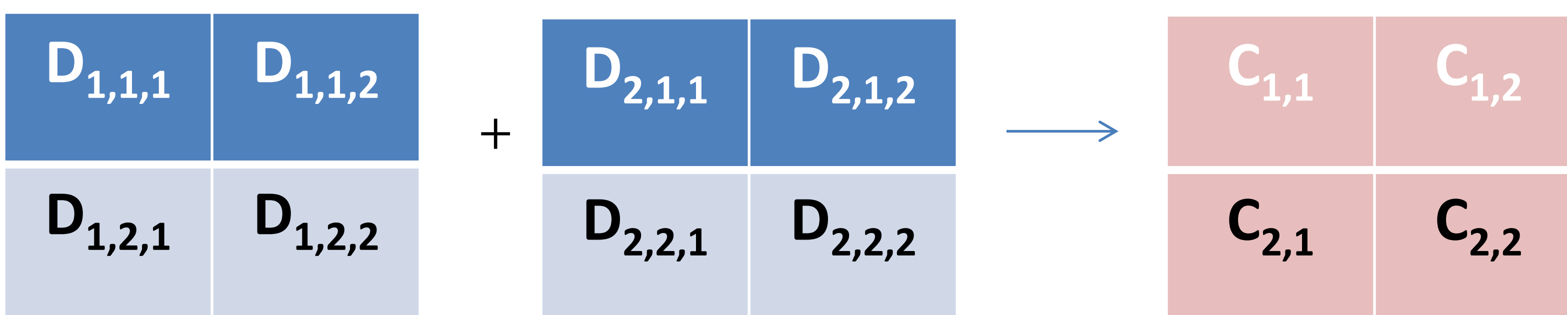
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

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

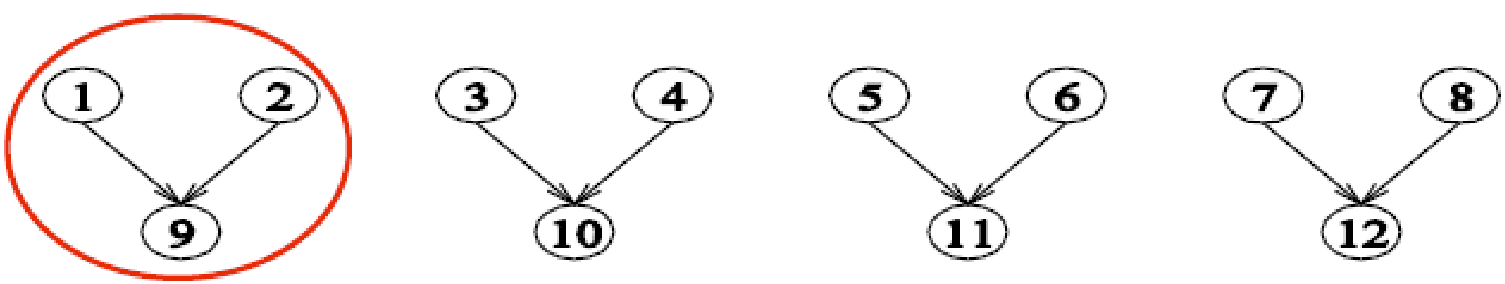
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

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.