Principles of Parallel Algorithm Design

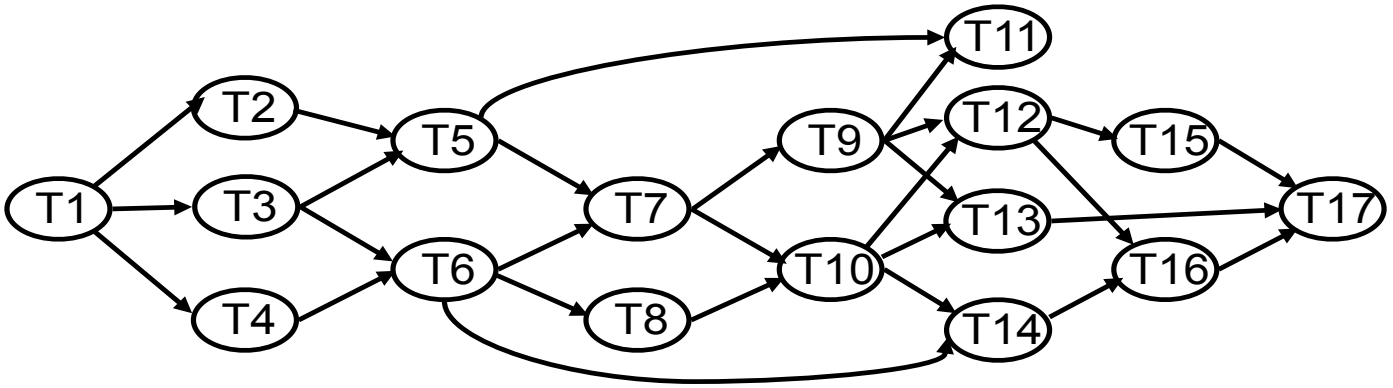
Concurrency and Mapping

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

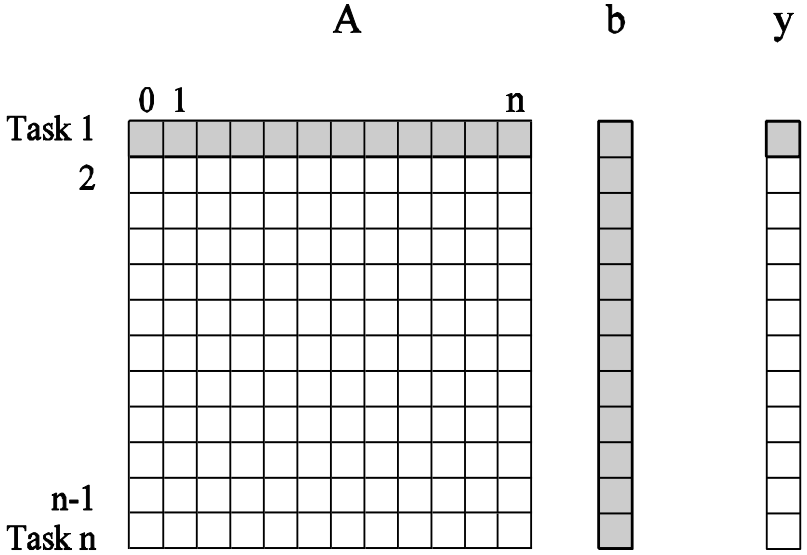
Combined with the slides of **John Mellor-Crummey**
Department of Computer Science Rice University

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 indeterminate 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

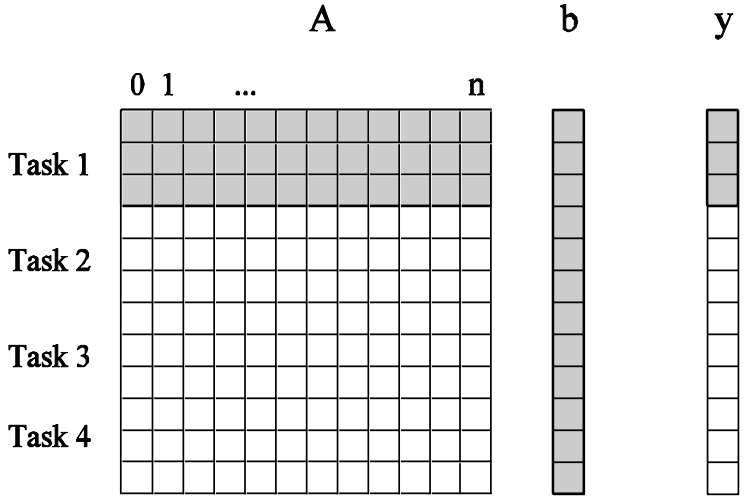


Computation of each element of output vector ***y*** is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into ***n*** tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

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?*

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.

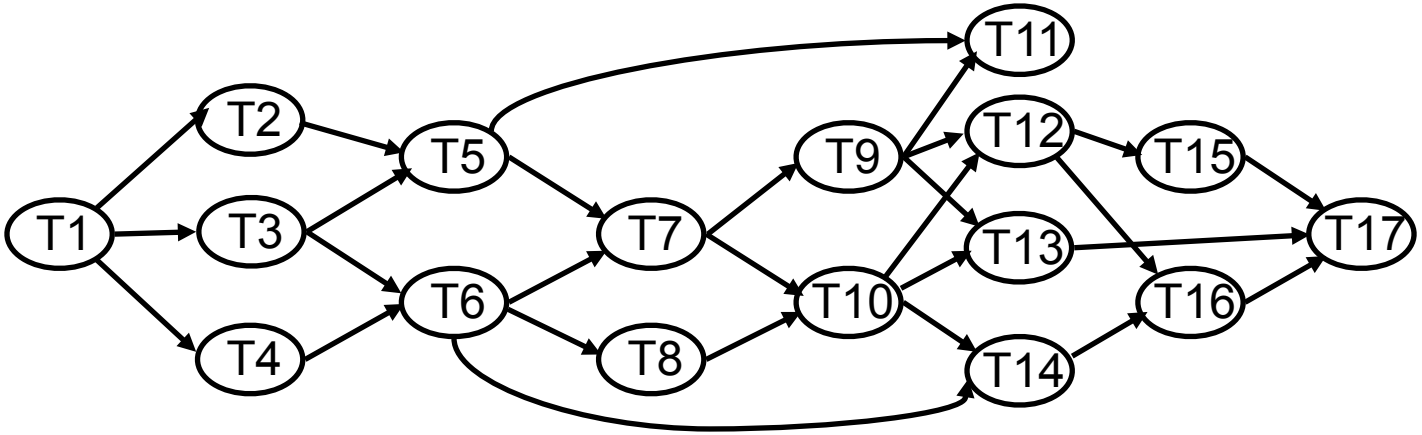
Degree of Concurrency

- The number of tasks that can be executed in parallel is the ***degree of concurrency*** of a decomposition.
  - ***maximum degree of concurrency***
    - ***largest # concurrent tasks at any point in the execution***
  - ***average degree of concurrency***
    - **average number of tasks that can be processed in parallel**

- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

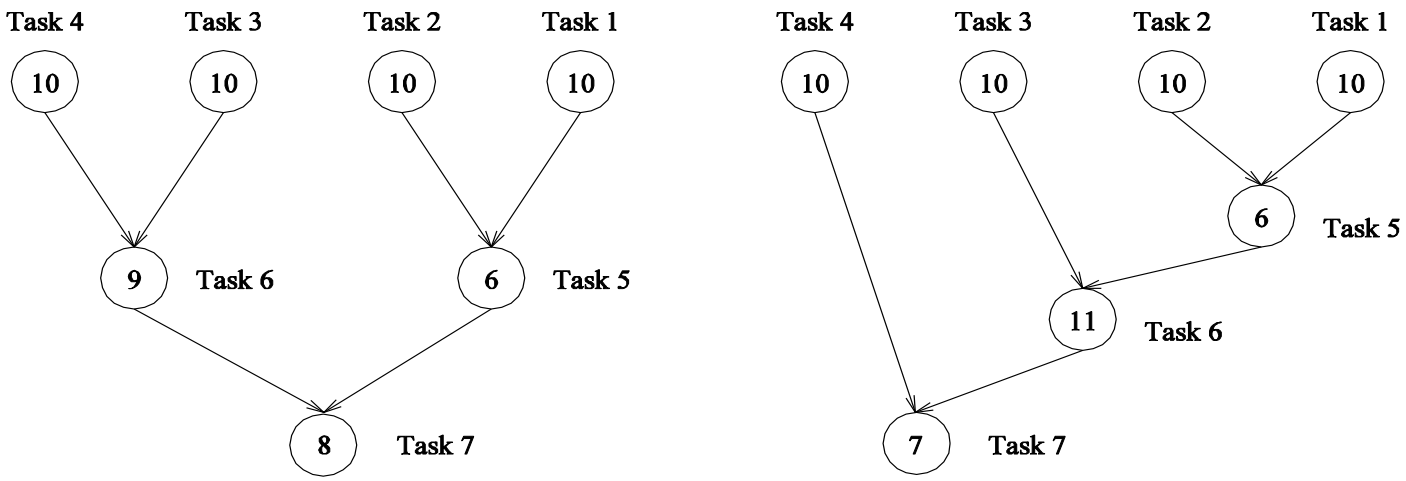
Critical Path Length

- 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 following task dependency graphs (e.g., database query)



Questions:

What are the tasks on the critical path for each dependency graph?

What is the shortest parallel execution time for each decomposition?

How many processors are needed to achieve the minimum time?

What is the maximum degree of concurrency?

Limits on Parallel Performance

- **What bounds parallel execution time?**
  - —minimum task granularity
    - *e.g. dense matrix-vector multiplication ≤ n^2 concurrent tasks*
  - —dependencies between tasks
  - —parallelization overheads
    - e.g., cost of communication between tasks
  - —fraction of application work that can't be parallelized
    - more about Amdahl's law in a later lecture ...
- **Measures of parallel performance**
  - —speedup = T_1/T_p
  - —parallel efficiency = $T_1/(pT_p)$

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.

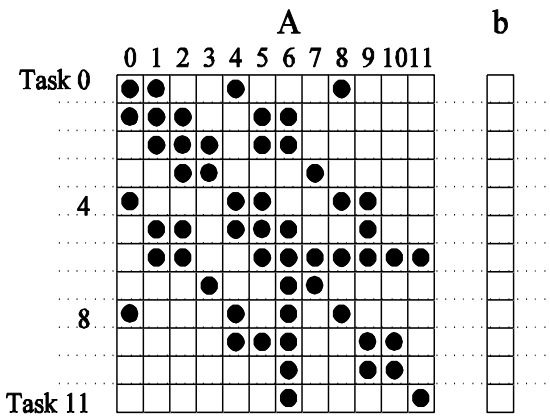
- The graph of tasks (**nodes**) and their interactions/data exchange (**edges**) is referred to as a ***task interaction graph***.

- Note that ***task interaction graphs*** represent data dependencies, whereas ***task dependency graphs*** represent control dependencies.

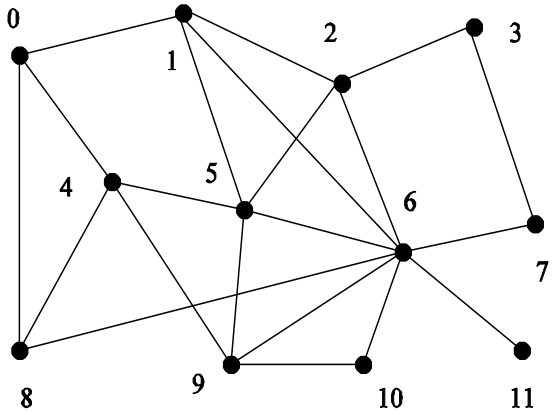
Task Interaction Graphs: An Example

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**).



(a)

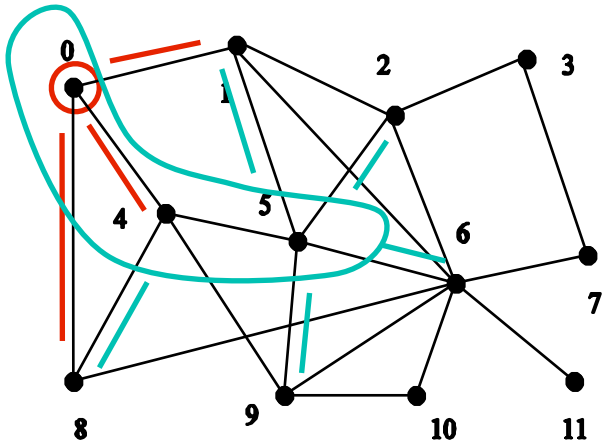


(b)

Task Interaction Graphs, Granularity, and Communication

In general, if the granularity of a decomposition is **finer**, the **associated overhead** (as a ratio of useful work associated with a task) increases.

Example: Consider the sparse matrix-vector product example from previous foil. Assume that each node takes unit time to process and each interaction (edge) causes an overhead of a unit time.



If node 0 is a task: communication = 3; computation = 4

If nodes 0, 4, and 5 are a task: communication = 5; computation = 15

—coarser-grain decomposition → smaller communication/computation ratio

Processors and Mapping

- In general, the number of tasks in a decomposition exceeds the number of processing elements available.

- For this reason, a parallel algorithm must also provide a mapping of tasks to processors.

Processes and Mapping

Mapping tasks to processes is critical for parallel performance

On what basis should one choose mappings?

- **—using task dependency graphs**
  - **schedule independent tasks on separate processes**
    - **minimum idling**
    - **optimal load balance**
- **—using task interaction graphs**
  - **want processes to have minimum interaction with one another**
    - **minimum communication**

Processes and Mapping

An appropriate mapping must minimize parallel execution time by:

- Mapping independent tasks to different processes.
- Assigning tasks on critical path to processes ASAP
- Minimizing interaction between processes by mapping tasks with dense interactions to the same process.

Note: These criteria often conflict with each other. For example, a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all!

Decomposition Techniques

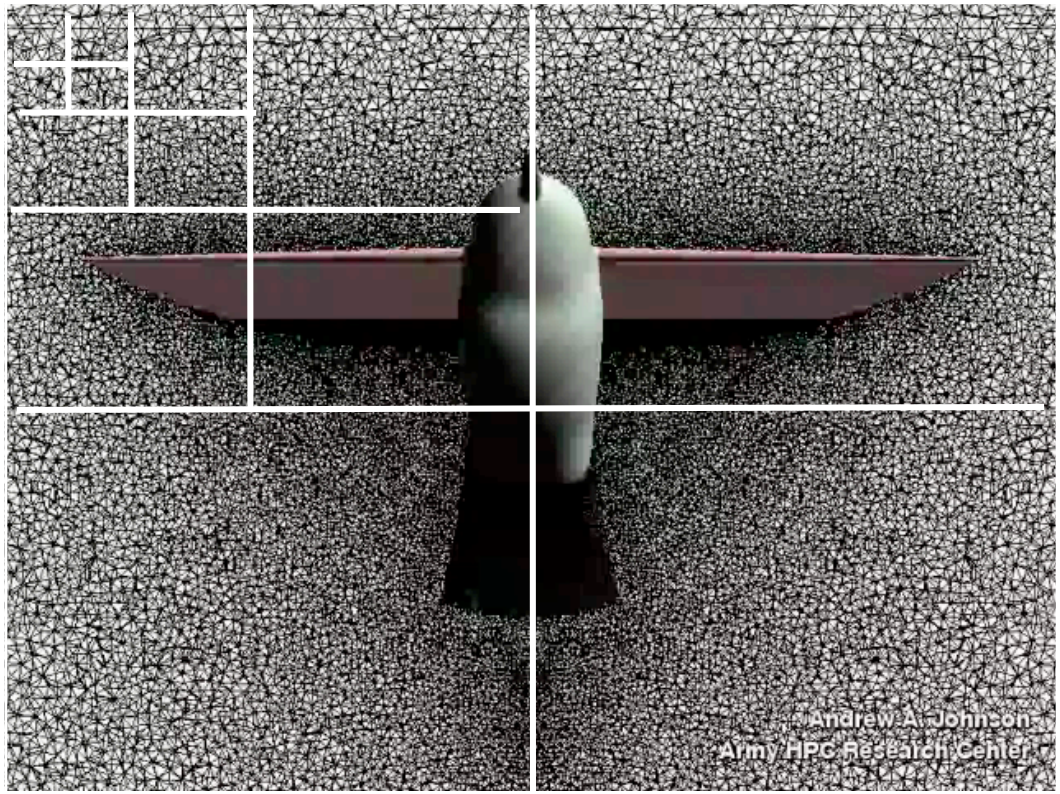
So how does one decompose a task into various subtasks?

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

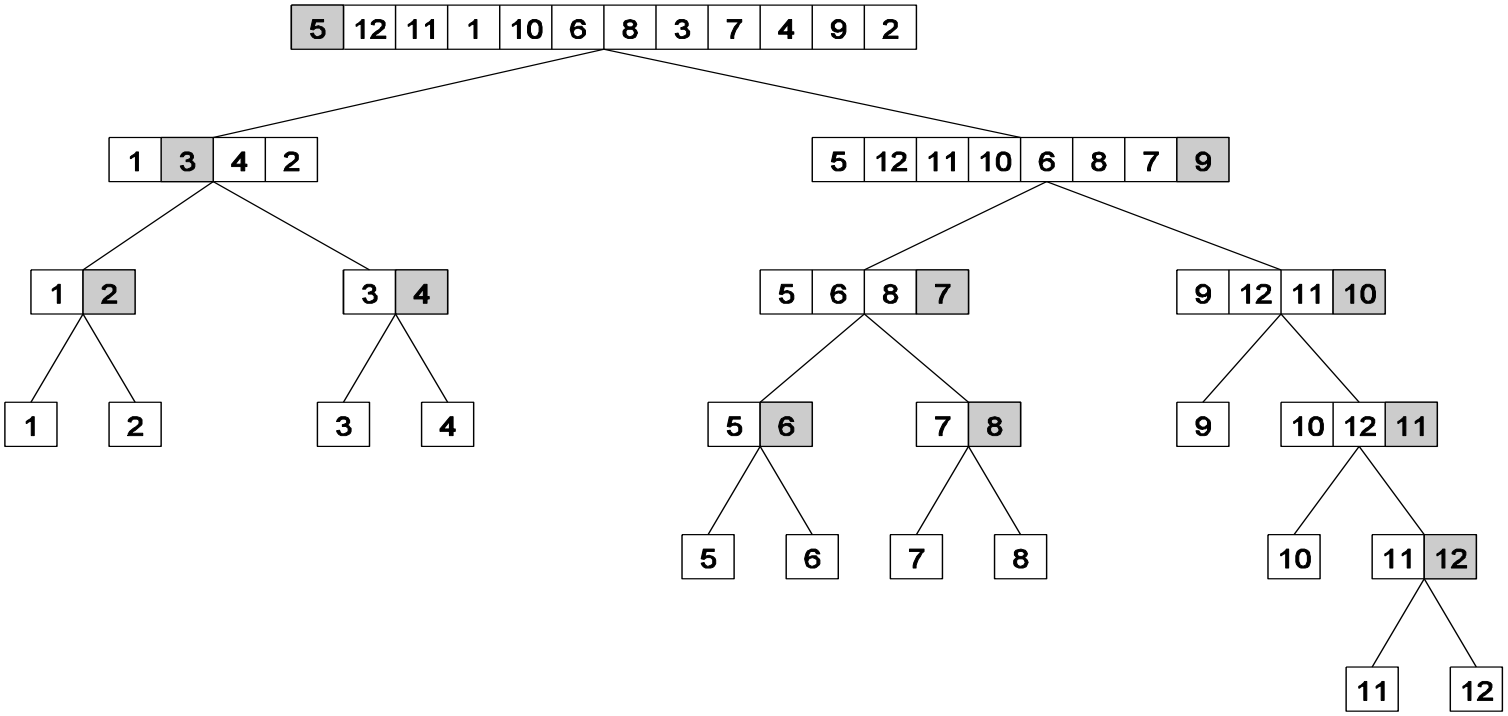
Recursive Decomposition

1. decompose a problem into a set of sub-problems
2. recursively decompose each sub-problem
3. stop decomposition when minimum desired granularity reached



Recursive Decomposition: Example

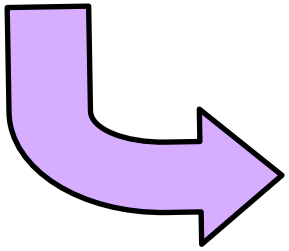
A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.



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

```
procedure SERIAL_MIN (A, n)
begin
   min = A[0];
   for i := 1 to n − 1 do
      if (A[i] < min) min := A[i];
   return min;
```



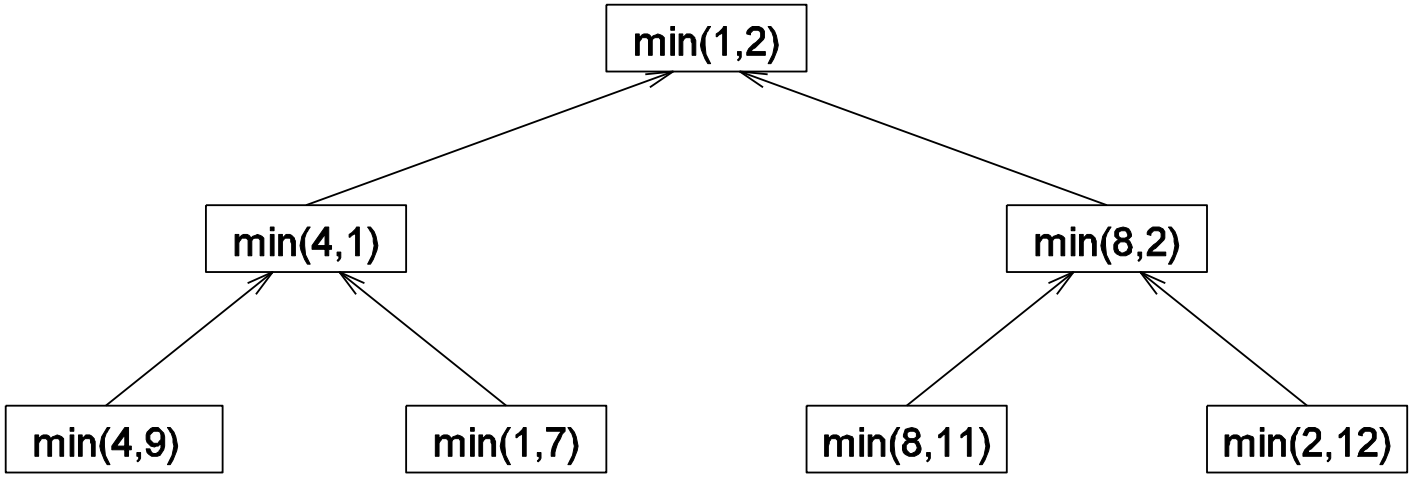
```
procedure RECURSIVE_MIN (A, n)
begin
   if ( n = 1 ) then
      min := A[0];
   else
      lmin := RECURSIVE_MIN(&A[0], n/2 );
      rmin := RECURSIVE_MIN(&A[n/2], n-n/2);
      if (lmin  < rmin) then
         min := lmin;
      else
         min := rmin;
    return min;
```

Recursive Decomposition: Example

The code in the previous slide 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

Steps

1. identify the data on which computations are performed
2. partition the data across various tasks
   - partitioning induces a decomposition of the problem

Data can be partitioned in various ways

— appropriate partitioning is critical to parallel performance

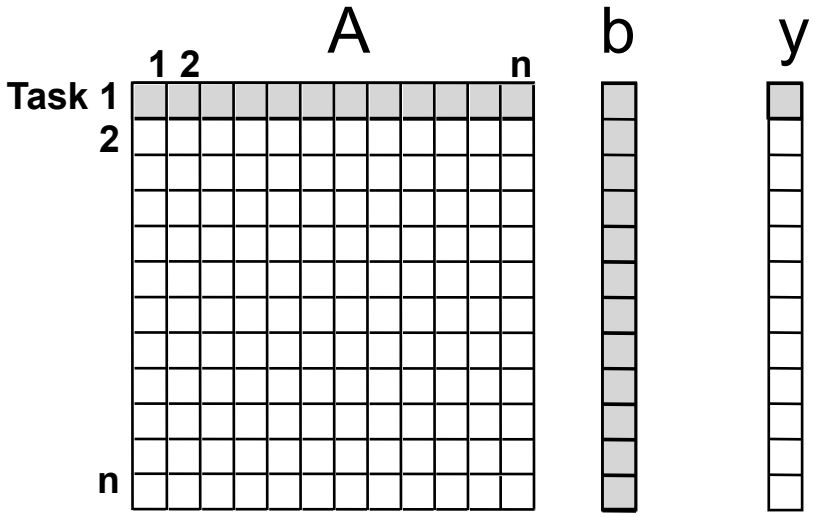
Decomposition based on

— input data
— output data
— input + output data
— intermediate data

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.

Example:
dense matrix-vector multiply



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

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

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		

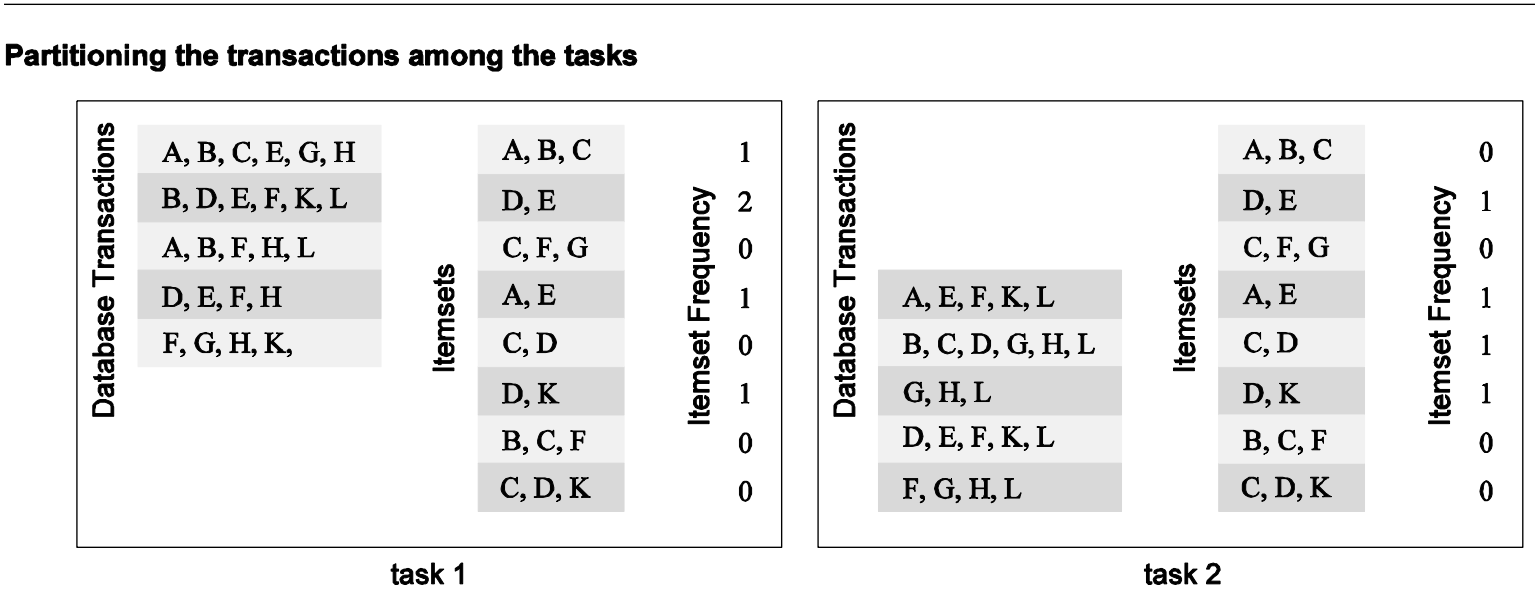
task 1

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		

task 2

Input Data Partitioning

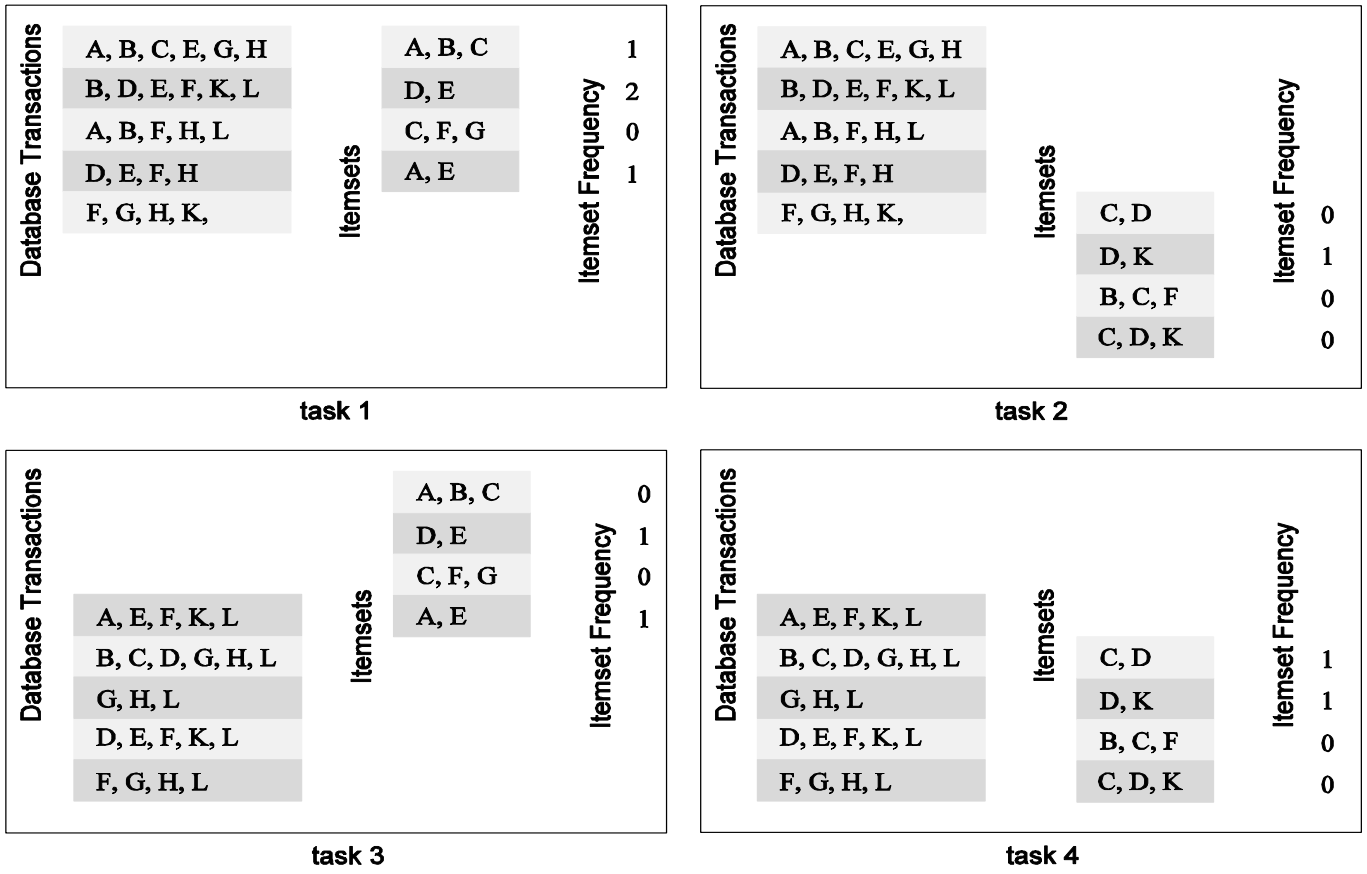
- Generally applicable **if each output can be naturally computed as a function of the input.**
- In many cases, this is the only natural decomposition
- A task is associated with each input data partition.
- Subsequent processing combines these partial results.



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:

Partitioning both transactions and frequencies among the tasks



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

Dense matrix mult. revisited: A decomposition of intermediate data structure leads to the following decomposition into 8 + 4 tasks:

Stage I

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

Stage II

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

Intermediate Data Partitioning: Example

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

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: $D_{1,2,2} = A_{2,1} B_{1,2}$

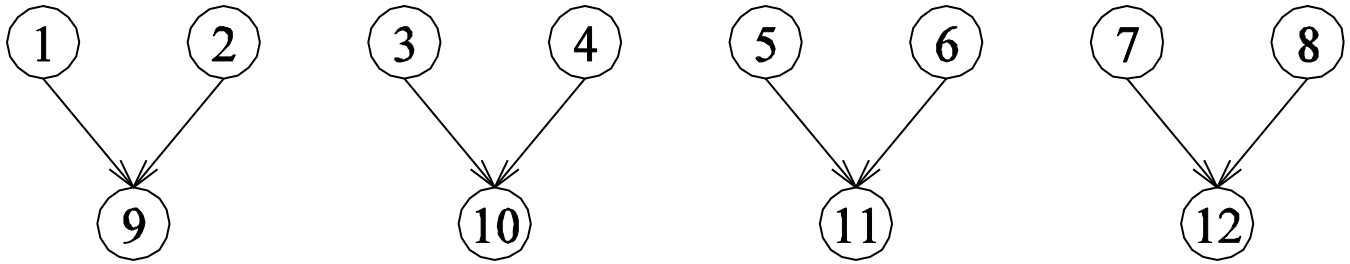
Task 08: $D_{2,2,2} = A_{2,2} B_{2,2}$

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



The Owner Computes Rule

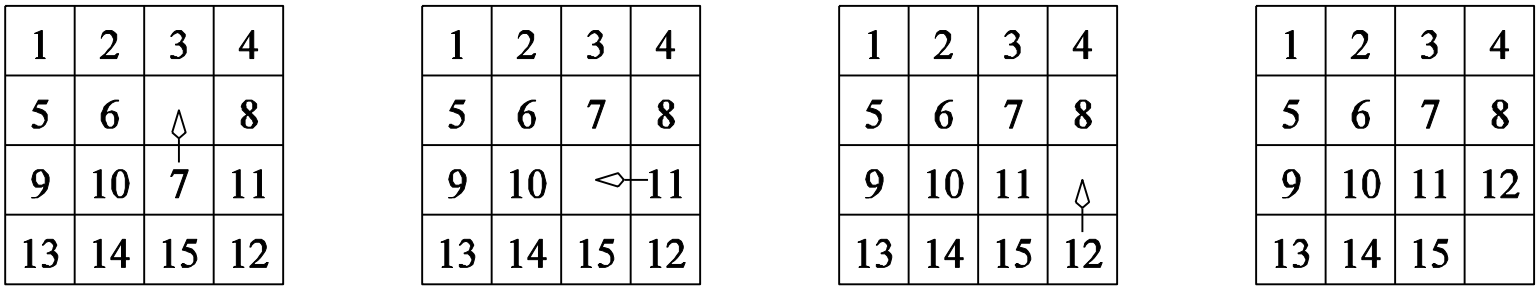
- The ***Owner Computes Rule*** generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that **all computations that use the input data are performed by the process**.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

Exploratory Decomposition

- In many cases, the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration (search) of a state space of solutions.
- Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.

Exploratory Decomposition: Example

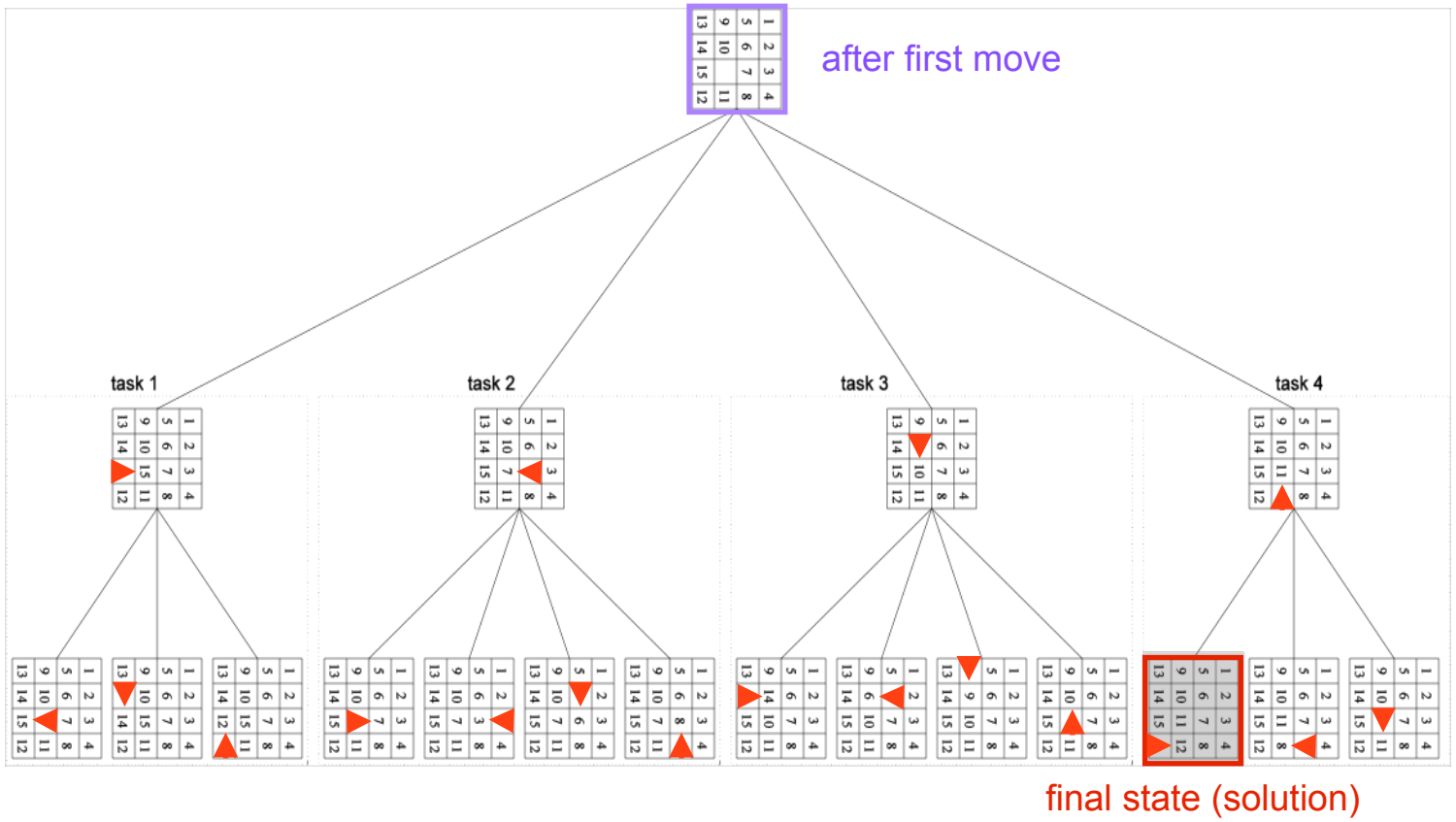
A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).



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

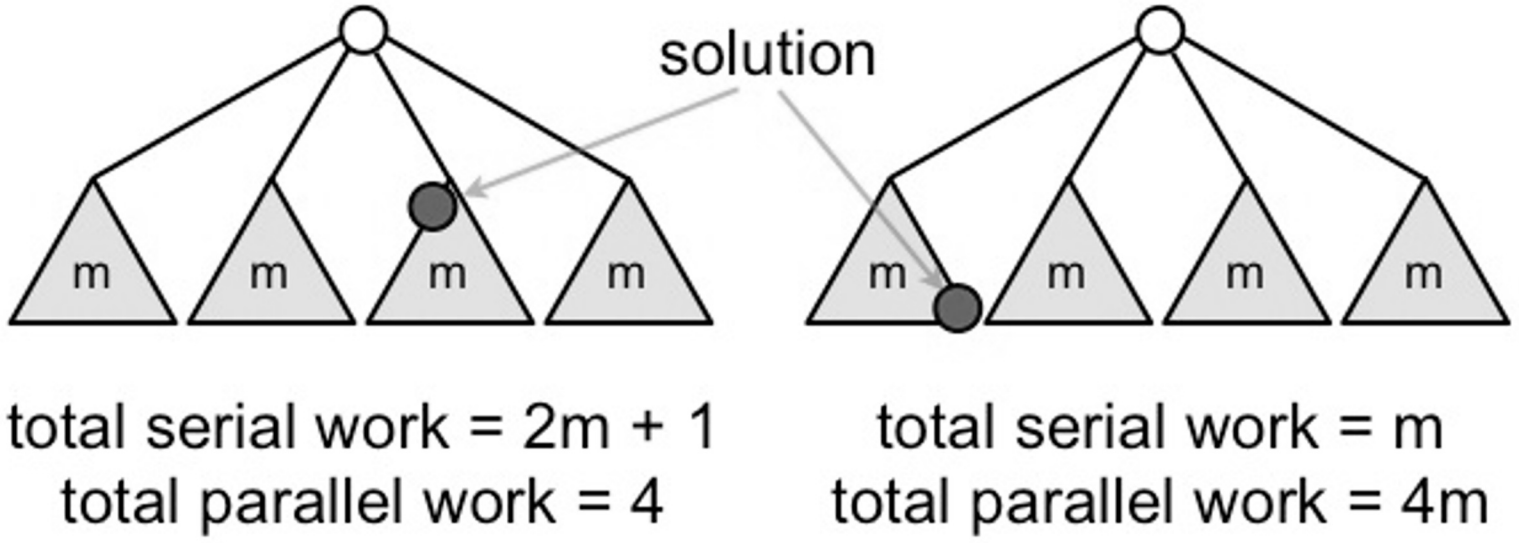
Exploratory Decomposition: Example

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



Exploratory Decomposition: Anomalous Speedups

- 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.



Speculative Decomposition

- In some applications, **dependencies between tasks are not known a-priori.**
- For such applications, it is **impossible to identify independent tasks**.
- **conservative approaches,**
  - identify independent tasks only when they are guaranteed to not have dependencies
- **optimistic approaches**
  - schedule tasks even when they may potentially be erroneous.
- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.

Characteristics of Tasks 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.

Size of Data Associated with Tasks

Data may be small or large compared to the computation

- —size(input) < size(computation), *e.g.*, 15 puzzle
- —size(input) = size(computation) > size(output), *e.g.*, min
- —size(input) = size(output) < size(computation), *e.g.*, sort

Implications

- —small data: task can easily migrate to another process
- —large data: ties the task to a process
  - possibly can avoid communicating the task context
    reconstruct/recompute the context elsewhere

Characteristics of Task Interactions

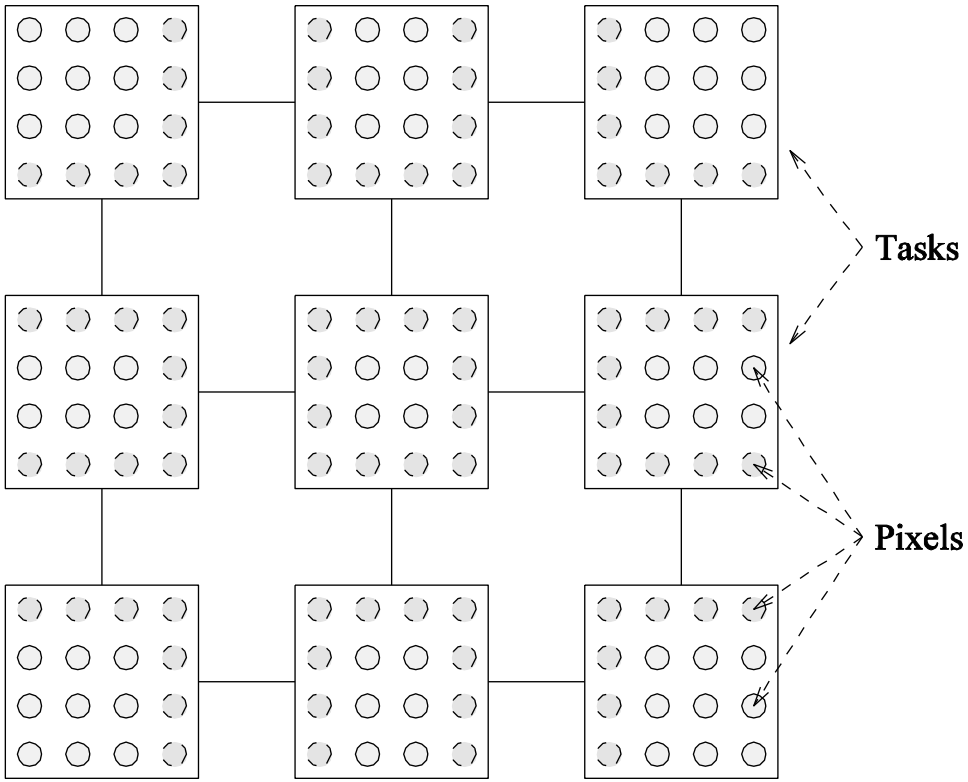
- Tasks may communicate with each other in various ways. The associated dichotomy is:
- **Static** interactions: The tasks and their interactions are known a-priori. These are relatively simpler to code into programs.
- **Dynamic** interactions: The timing or interacting tasks cannot be determined a-priori. These interactions are harder to code, especially using message passing APIs.

Characteristics of Task Interactions

- **Regular** interactions: There is a definite pattern (in the graph sense) to the interactions. These patterns can be exploited for efficient implementation.
- **Irregular** interactions: Interactions lack well-defined topologies.

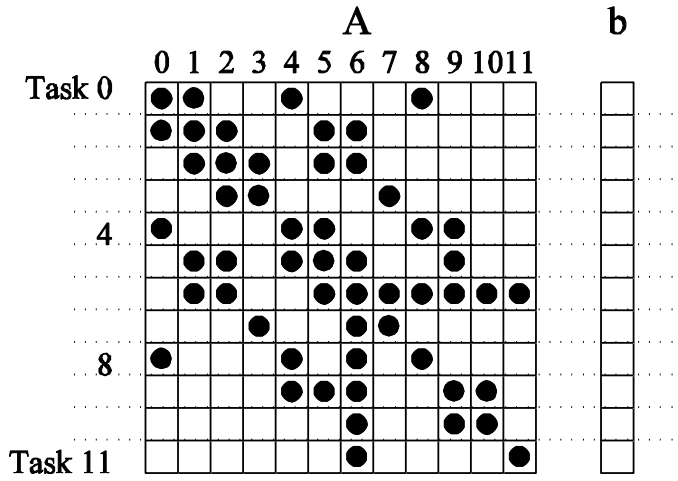
Characteristics of Task Interactions: Example

A simple example of a **static regular interaction** pattern is in **image dithering**. The underlying communication pattern is a structured (2-D mesh) one as shown here:

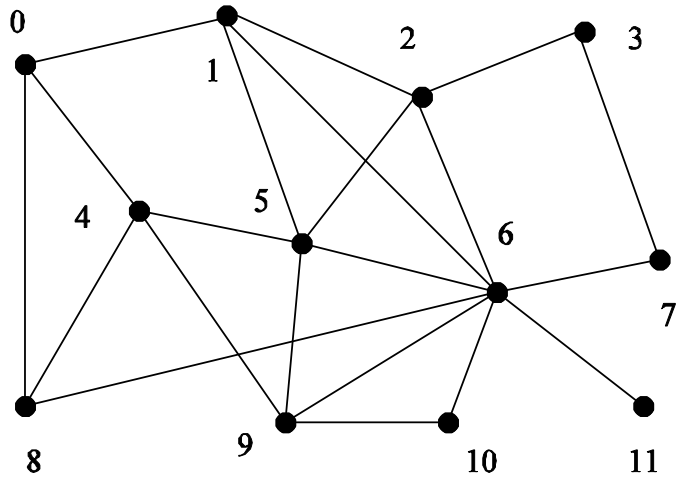


Characteristics of Task Interactions: Example

The multiplication of a sparse matrix with a vector is a good example of a **static irregular interaction** pattern. Here is an example of a sparse matrix and its associated interaction pattern.



(a)



(b)

Characteristics of Task Interactions

- Interactions may be **read-only** or **read-write**.
  - In *read-only* interactions, tasks just read data items associated with other tasks.
  - In *read-write* interactions tasks read, as well as modify data items associated with other tasks.
- In general, **read-write interactions are harder to code**,
  - they require additional synchronization primitives.

Characteristics of Task Interactions

- Interactions may be **one-way** or **two-way**.
  - A *one-way* interaction can be initiated and accomplished by one of the two interacting tasks.
  - A *two-way* interaction requires participation from both tasks involved in an interaction.
- One way interactions are somewhat harder to code in message passing APIs.
  - New versions of these libraries successfully support one-way communication

Chapter Overview: Part 2

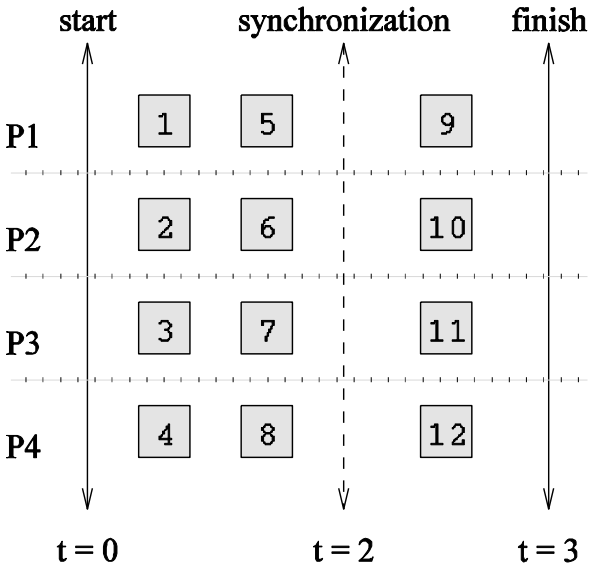
- Mapping Techniques for Load Balancing
  - Static and Dynamic Mapping
- Methods for Minimizing Interaction Overheads
  - Maximizing Data Locality
  - Minimizing Contention and Hot-Spots
  - Overlapping Communication and Computations
  - Replication vs. Communication
  - Group Communications vs. Point-to-Point Communication

Mapping Techniques

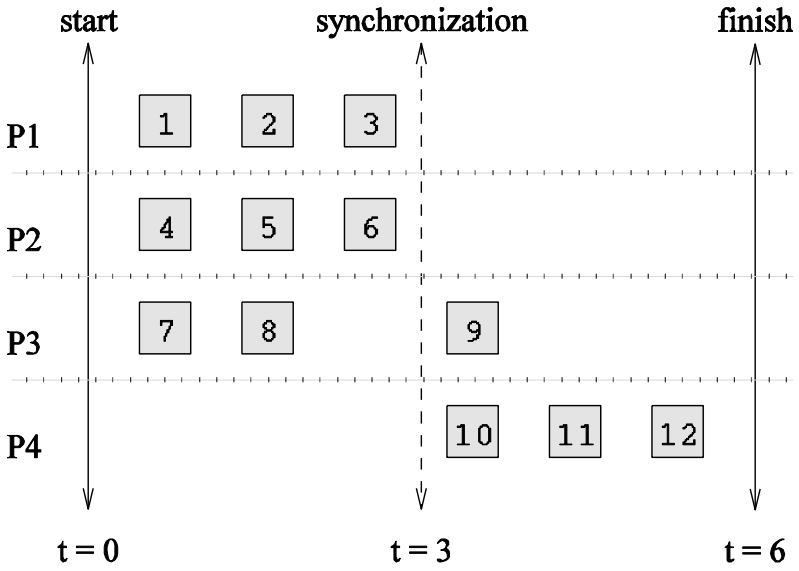
- Once a problem has been decomposed into concurrent tasks, these must be mapped to processes (that can be executed on a parallel platform).
- Mappings must minimize overheads.
- Primary overheads are communication and idling.
- Minimizing these overheads often represents contradicting objectives.
- 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 balance the load (as in (a) below). Merely balancing load does not minimize idling (see (b) for an example).



(a)



(b)

Mapping Techniques for Minimum Idling

Mapping techniques can be static or dynamic.

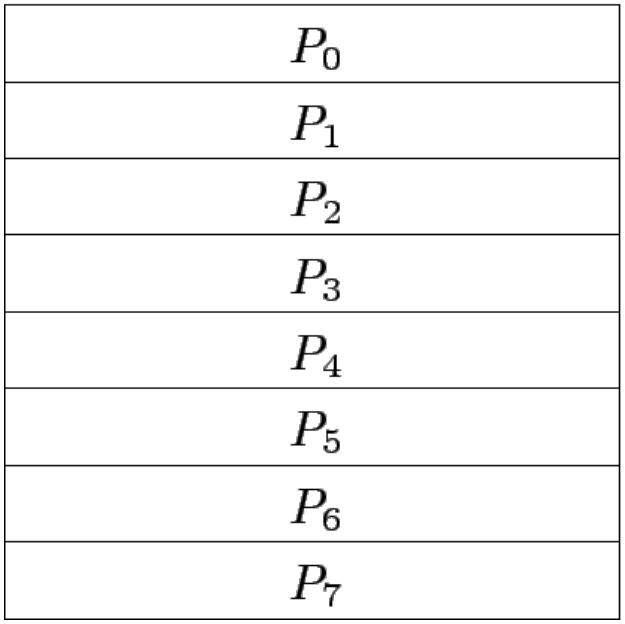
- **Static Mapping**: Tasks are mapped to processes a-priori. For this to work, we must have a good estimate of the size of each task. Even in these cases, the problem may be NP complete.
- **Dynamic Mapping**: Tasks are mapped to processes at runtime. This may be because the tasks are generated at runtime, 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.

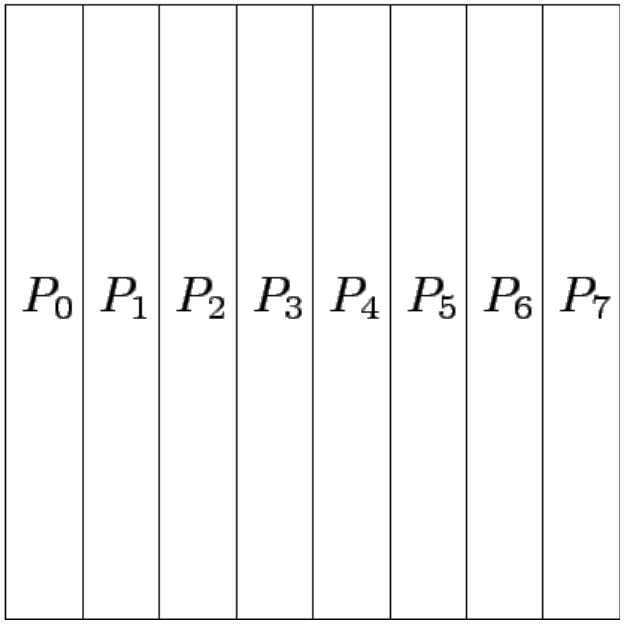
Mappings Based on Data Partitioning

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

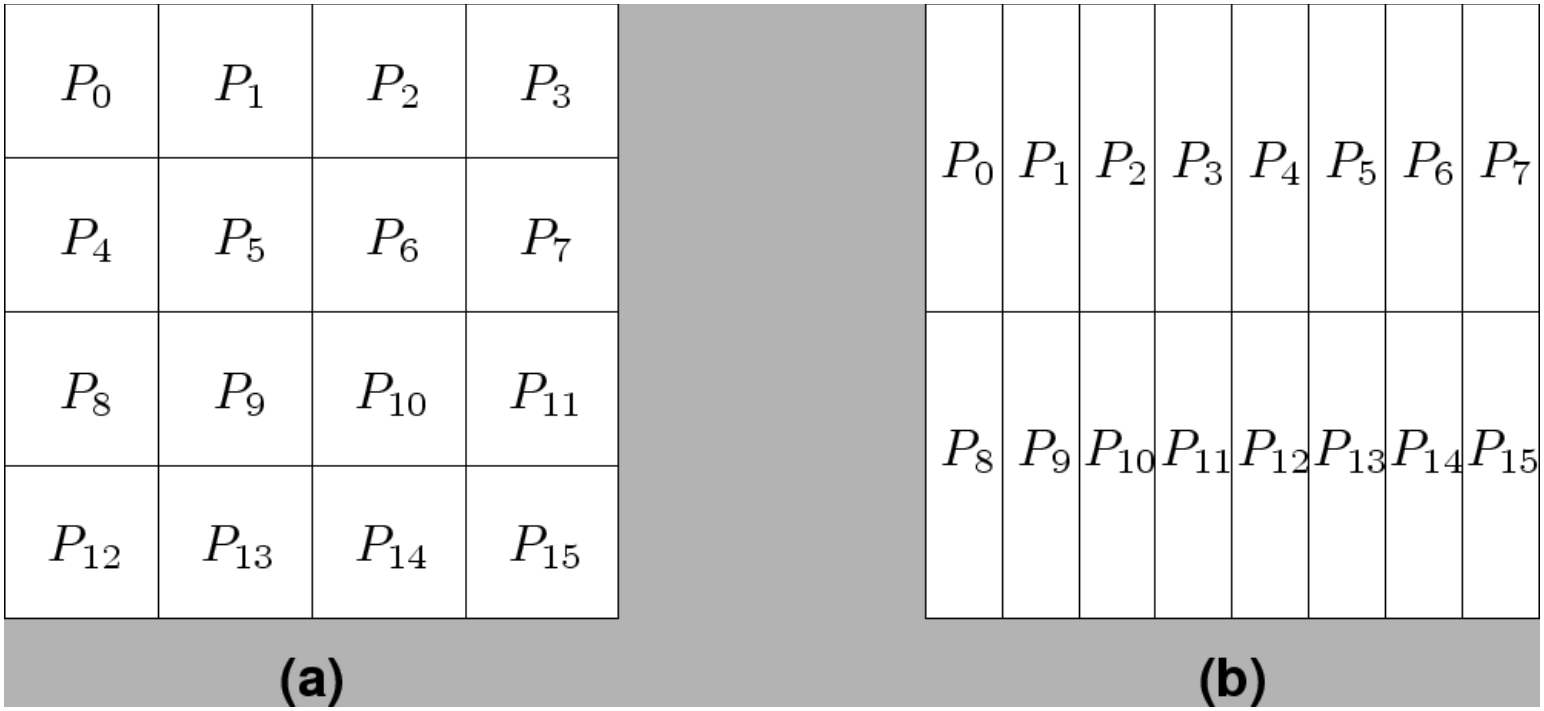


column-wise distribution



Block Array Distribution Schemes

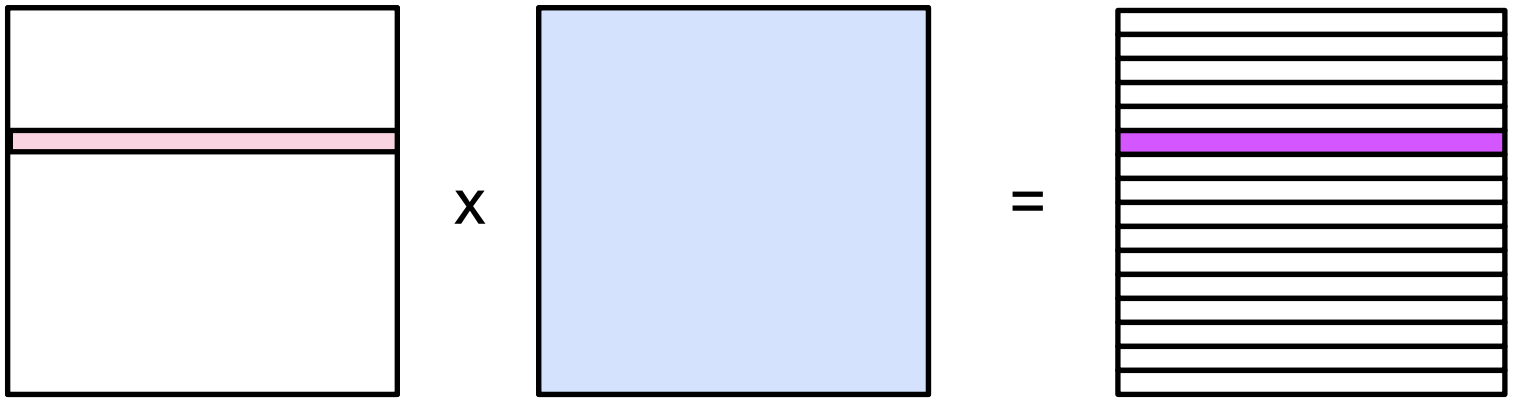
Block distribution schemes can be generalized to higher dimensions as well.

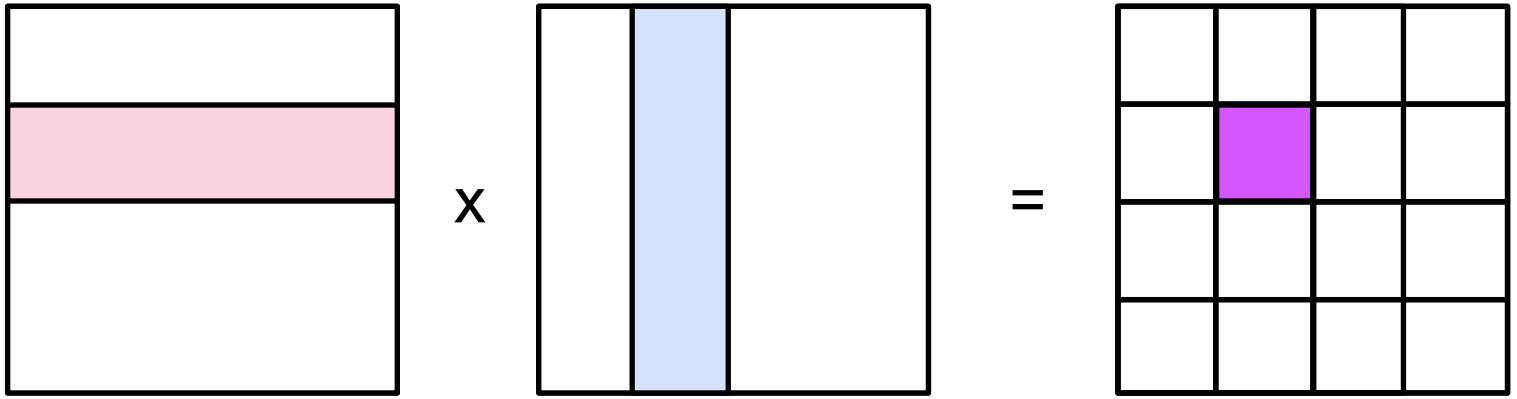


Block Array Distribution Schemes: Examples

- 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.

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

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: $A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$

10: $A_{2,2} \rightarrow L_{2,2}U_{2,2}$

11: $L_{3,2} = A_{3,2}U_{2,2}^{-1}$

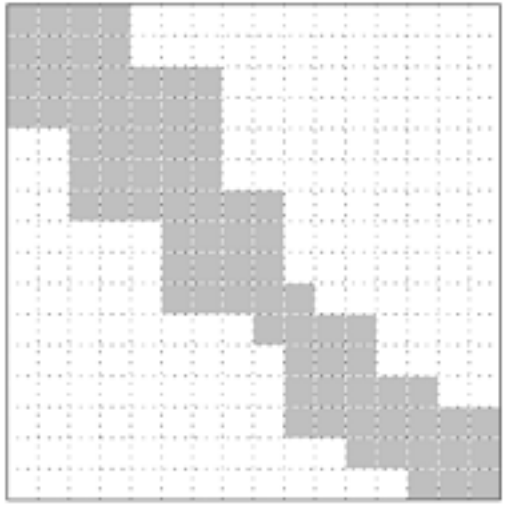
12: $U_{2,3} = L_{2,2}^{-1}A_{2,3}$

13: $A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$

14: $A_{3,3} \rightarrow L_{3,3}U_{3,3}$

Graph Partitioning Dased Data Decomposition

- In case of sparse matrices, block decompositions are more complex.
- Consider the problem of multiplying a sparse matrix with a vector.
- The graph of the matrix is a useful indicator of the work (number of nodes) and communication (the degree of each node).
- In this case, we would like to partition the graph so as to assign equal number of nodes to each process, while minimizing edge count of the graph partition.

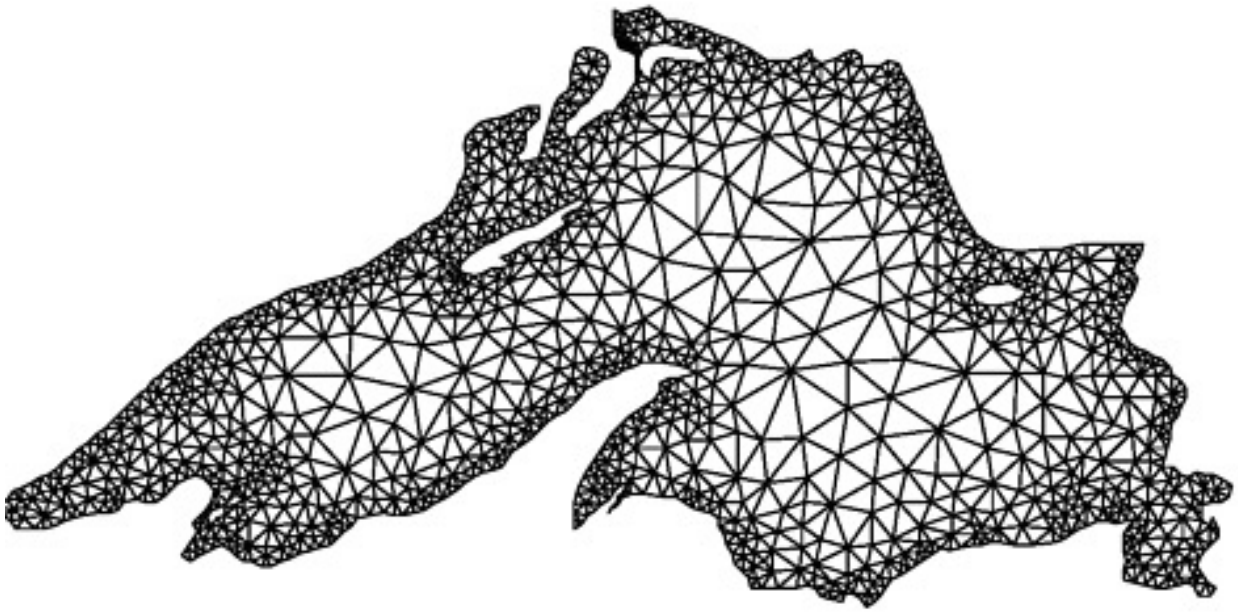


(a)

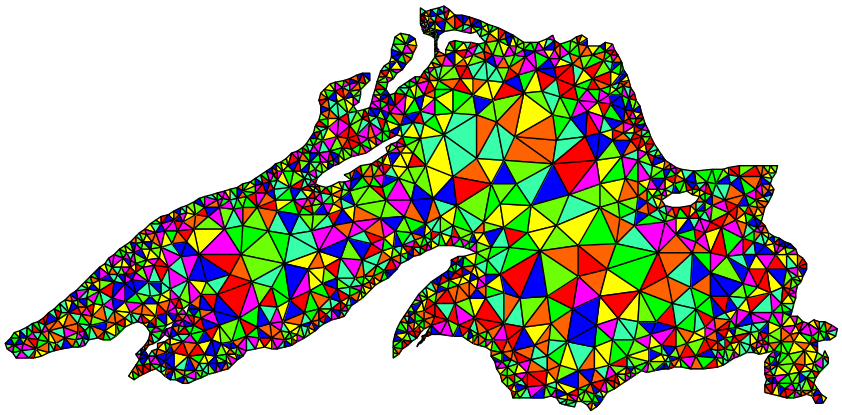
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}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

(b)

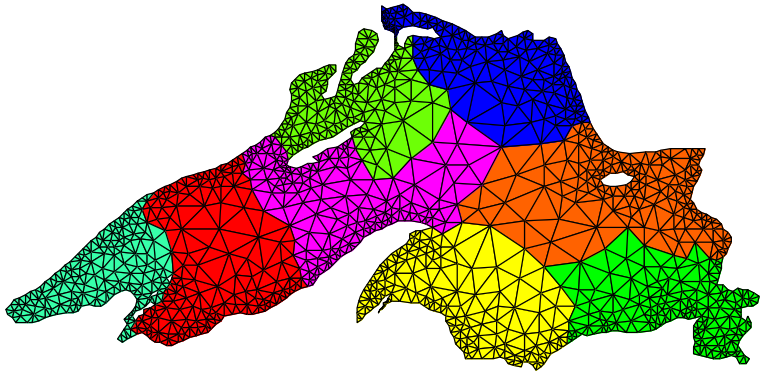
Partitioning the Graph of Lake Superior



Partitioning the Graph of Lake Superior



Random Partitioning



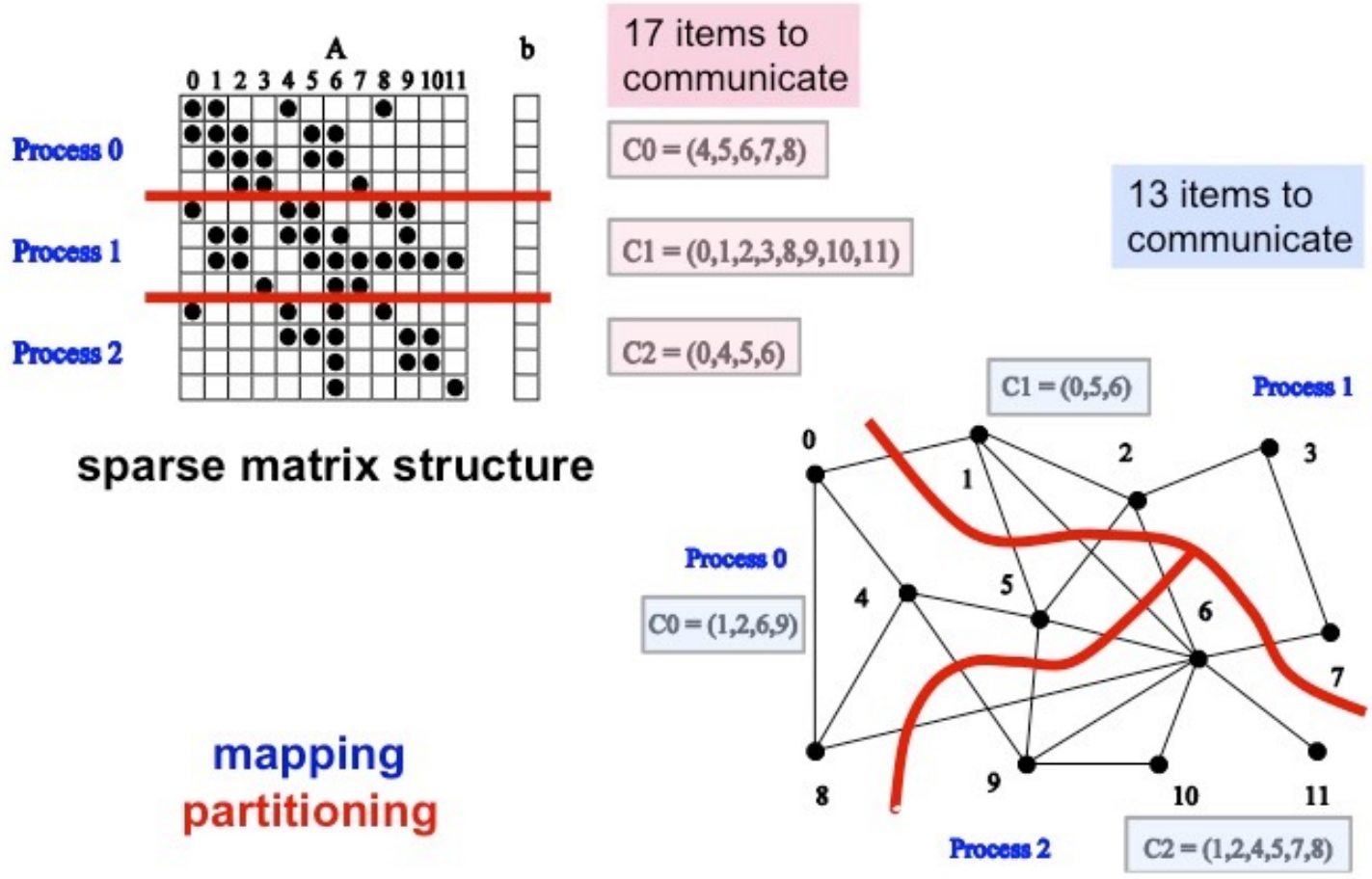
Partitioning for minimum edge-cut.

Mappings Based on Task Partitioning

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

Task Partitioning: Mapping a Sparse Graph

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

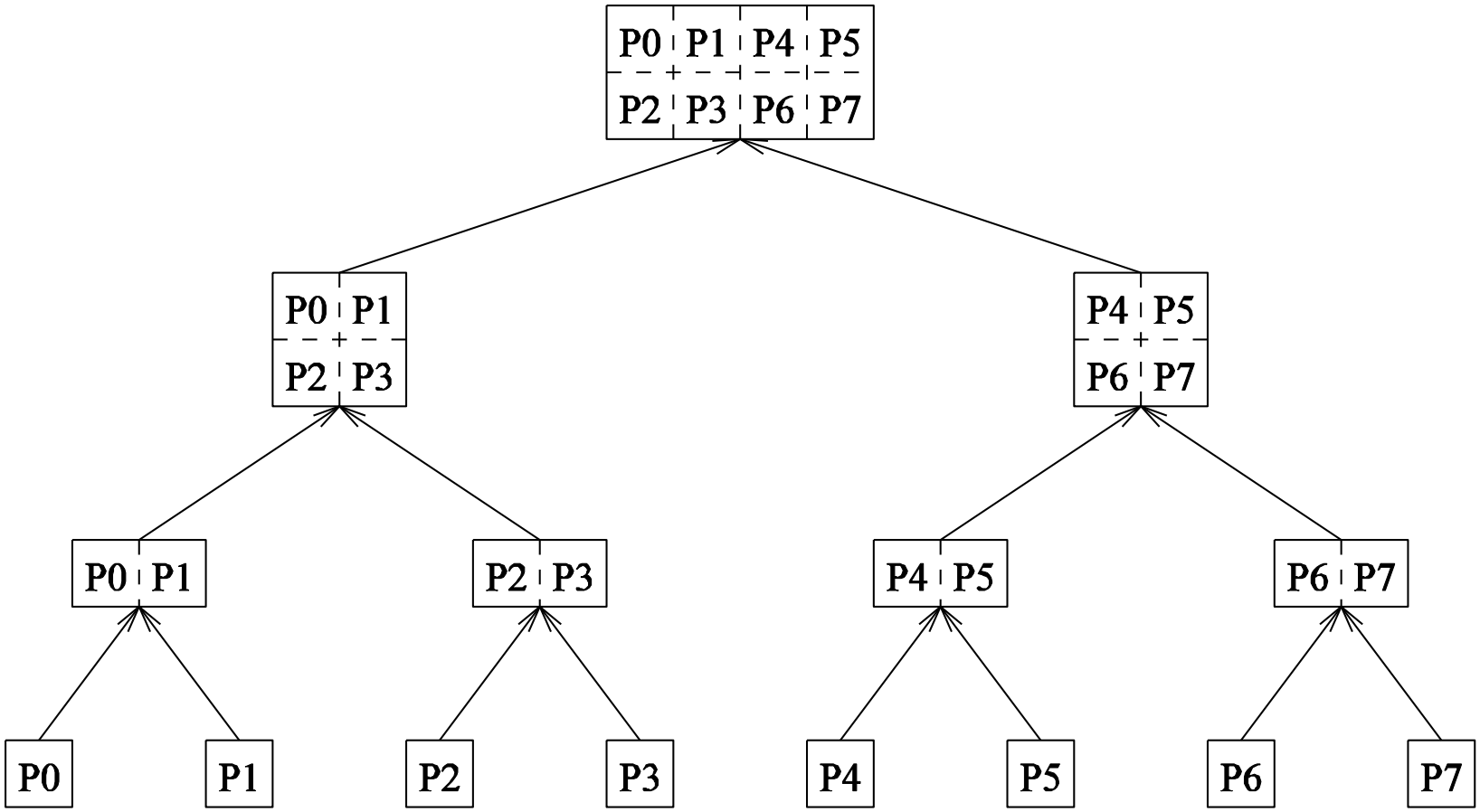


Hierarchical Mappings

- 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 Mappings

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 sending 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.

Minimizing Interaction Overheads

- **Maximize data locality**: Where possible, reuse intermediate data. Restructure computation so that data can be reused in smaller time windows.
- **Minimize volume of data exchange**: There is a cost associated with each word that is communicated. For this reason, we must minimize the volume of data communicated.
- **Minimize frequency of interactions**: There is a startup cost associated with each interaction. Therefore, try to merge multiple interactions to one, where possible.
- **Minimize contention and hot-spots**: Use decentralized techniques, replicate data where necessary.

Minimizing Interaction Overheads (continued)

- **Overlapping computations with interactions**: Use non-blocking communications, multithreading, and prefetching to hide latencies.
- **Replicating data or computations.**
- **Using group communications instead of point-to-point primitives**.

Parallel Algorithm Models

An algorithm model is a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

- **Data Parallel Model:** Tasks are statically (or semi-statically) mapped to processes and each task performs similar operations on different data.
- **Task Graph Model:** Starting from a task dependency graph, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs.

Parallel Algorithm Models (continued)

- **Master-Slave Model**: One or more processes generate work and allocate it to worker processes. This allocation may be static or dynamic.
- **Pipeline / Producer-Consumer Model**: A stream of data is passed through a succession of processes, each of which perform some task on it.