Module-3

Parallel Algorithm Design

Overview:

- *Tasks and Decomposition (concurrent work)*
- *Decomposition Techniques*
- *Processes and Mapping (work to parallel process mapping)*
- *Processes Versus Processors*
- *Mapping Techniques for Load Balancing*
- *Distribution : input, output, intermediate data*
- *Managing access to data*
- *Synchronizing the processors*
- *Characteristics of Tasks and Interactions*
- *Methods for Minimizing Interaction Overheads*
- *Parallel Algorithm Design Models*

Correct + Fast+ efficient → Performance.

Theory (complexity) vs. Practical (actual run-time)

Decomposition, Tasks, Dependency Graphs

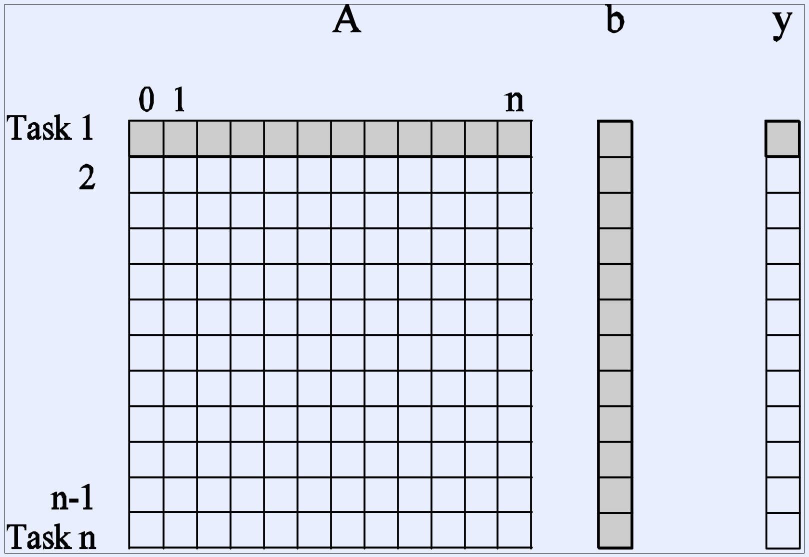
- Dividing the problem into tasks that can be executed concurrently/ in parallel → decomposition.
- Task may be small or large. (programmer defined)
- Tasks may be of same, different, or even intermediate sizes.

- A decomposition can be represented 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***.

- Reduction example. Matrix-vector multiplication.

Task dependency graph or DAG (directed acyclic graph).

Example: Multiplying a Dense Matrix with a Vector



Computing elements (n) of output vector ***y*** is independent of other elements. So, a matrix-vector product can be decomposed into ***n*** tasks.

Grey → portion of the matrix & vector accessed by Task 1.

observation

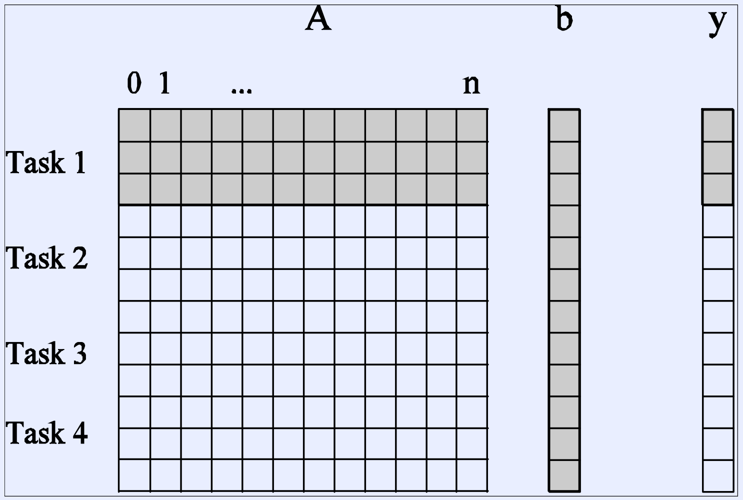
- While tasks share data (namely, the vector ***b***), they do not have any control dependencies - i.e., no task needs to wait for the (partial) completion of any other.

- 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?*

Different ways of arranging computation → 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 dense matrix-vector product example.
Each task → 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. (reduction example)
- The ***average degree of concurrency*** is the average number of tasks that can be processed in parallel over the execution of the program.
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.
- Degree of concurrency depends on shape of task dependency graph (can be different for same granurality).

Cost model

Each node → task (more !); edge → dependency (less !)

Complexity analysis - Cost model (PRAM – shared memory) :

•Each operation takes one unit of time

•Same speed of all processor

•No edge cost

•**No synchronization, communication overhead**

Summary: ignores practical issues of memory access time, neglects parallel overhead, however **problem size** and no of **processors** are there.

Serial cost/Time complexity (order of time)

Parallel cost/time complexity (order of time x no. Of processors)

Work span law

Work (w)=

1. Total no of nodes/taks/vertices if all node does same and unit amount of work. Or
2. Summation of work done by each node/task if all node does not do the same amount of work

Span/depth (D):
Longest path in the graph

W/D=average amount of parallelism (optimal number of processor to keep all processors busy on average)

Work span law

- Depth/span is lower bound (min possible time)
- When all processors(p) does same work → W/p is also lower bound
- Combine both to get work-span law for lower bound

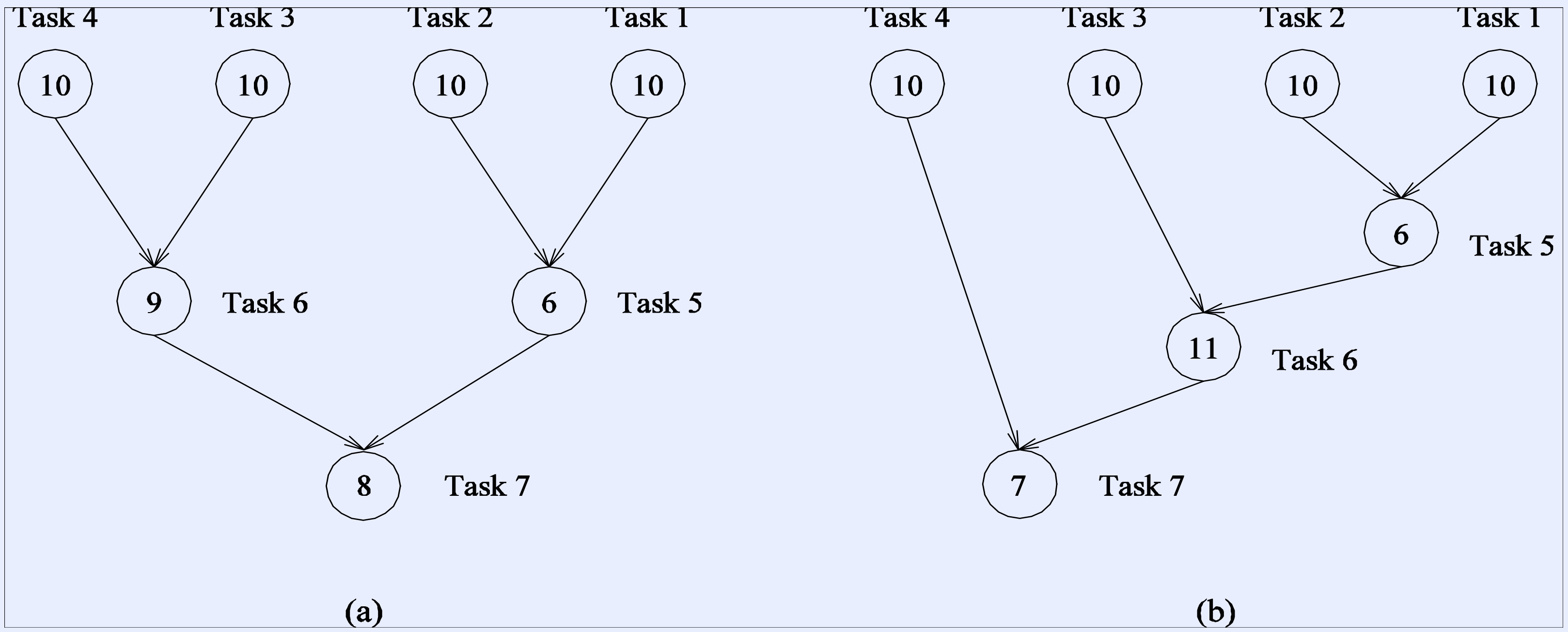
Parallel time >= max(depth, ceiling(W/p))

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** → **sum of the weights (size of work) of nodes along the critical path.**
- Ratio of total work to the critical path length → **average degree of concurrency.**

Critical Path Length

Consider the following 2 task dependency graphs (decompositions):



- What are the critical path lengths for the two task dependency graphs?
- If each task takes 10 time units, 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 degree of concurrency?
- 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 a dense matrix multiplication, there can be no more than* ***(n^2)*** *concurrent tasks.*
- Concurrent tasks may also have to exchange data with other tasks (distributed). 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, in the decomposition of a 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***.

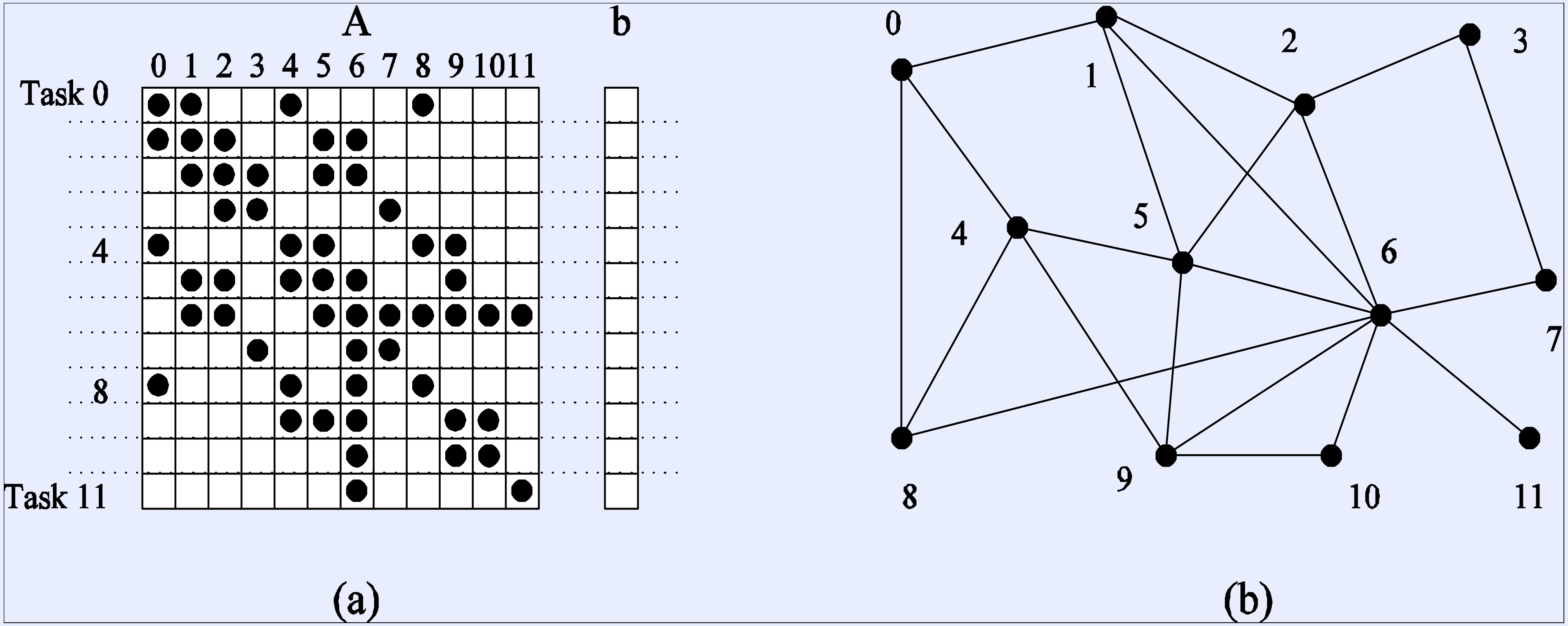
Task Interaction Graphs: An Example

Multiplication of a **sparse matrix *A*** with a vector ***b***. *Result R*

- computation of each element of the result vector is 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 we can make a task interaction graph.

Task Interaction Graph: An Example

Multiplication of a **sparse matrix *A*** with a vector ***b****. Result R*



Task interaction graph of the computation is identical to the graph of the matrix ***A.***

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.

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

Processes and Mapping

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

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

  tasks → processes →   physical processors.

Given the two graphs – what is the strategy ?

Processes and Mapping

- Appropriate mapping of tasks to processes is critical to the parallel performance of an algorithm.
- Mappings are determined by both the task dependency and task interaction graphs.
- **Task dependency graphs** can be used to ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance).
- **Task interaction graphs** can be used to make sure that processes need minimum interaction with other processes (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 as soon as they become available.
- ➢ Minimizing interaction between processes by mapping tasks with dense interactions to the same process.

Decomposition Techniques

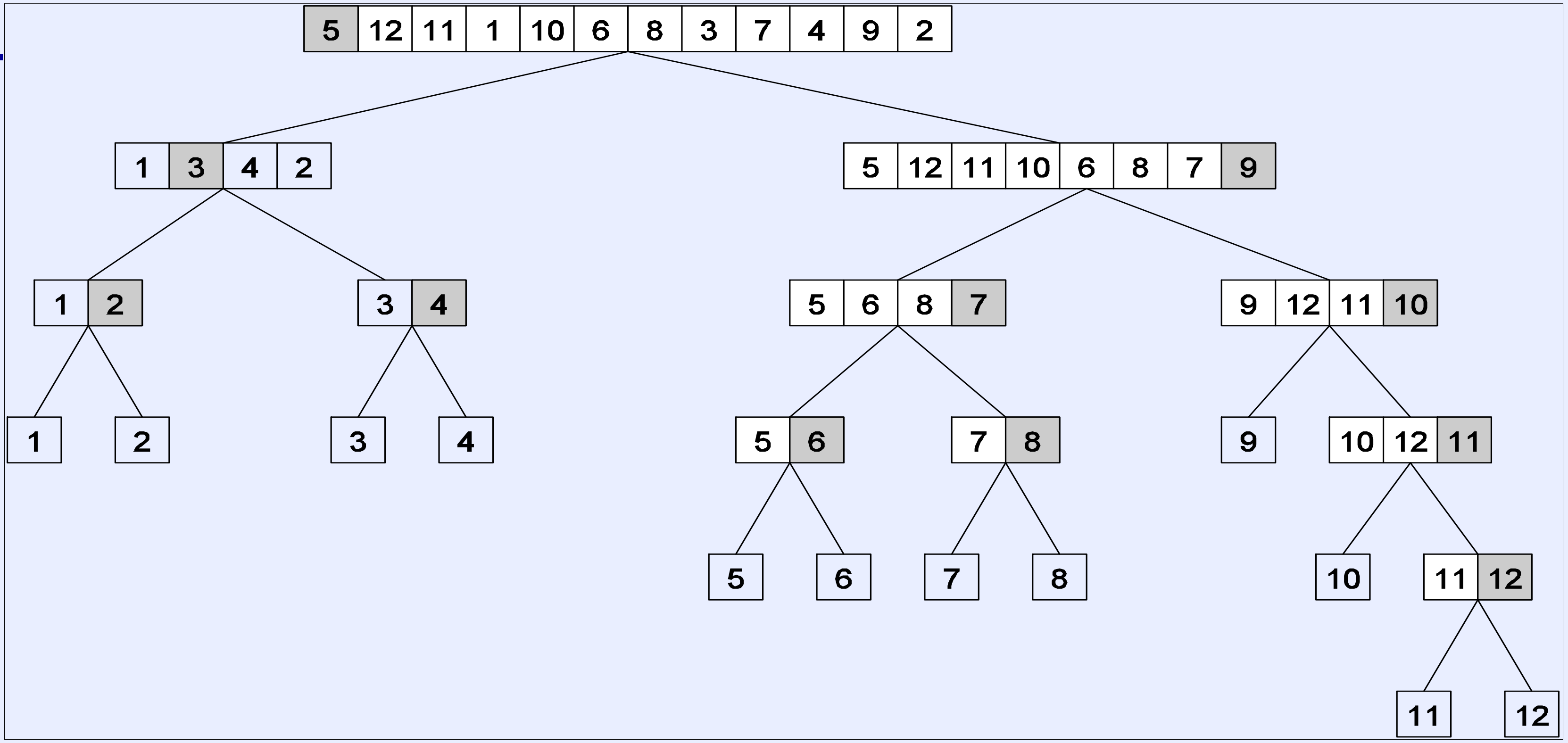
- recursive decomposition
- data decomposition
- exploratory decomposition
- speculative decomposition

Learning by example case studies.

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.
- Example - quicksort

Recursive Decomposition: Example - Quicksort



once the list has been partitioned around the pivot, each sub-list can be processed concurrently (each sublist - an independent subtask). This can be repeated recursively.

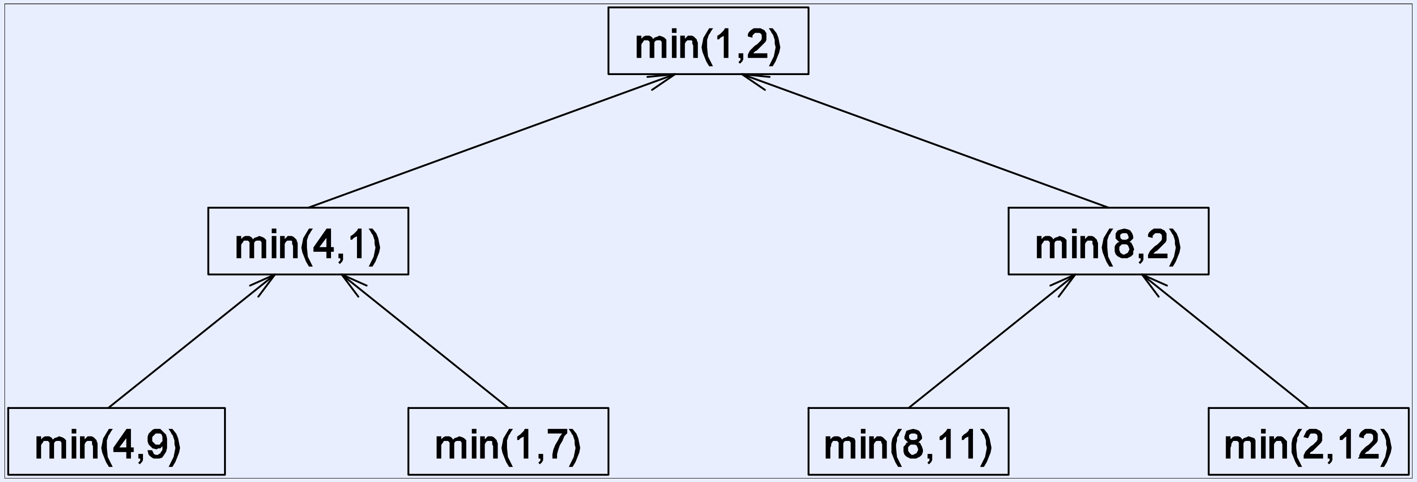
Recursive Decomposition: another Example

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.

Recursive Decomposition: Example

recursive decomposition strategy for min.

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 - this 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. Example?

- Example of input data partition?

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, with identical output data distribution, we can derive the following two (other) 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: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

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.

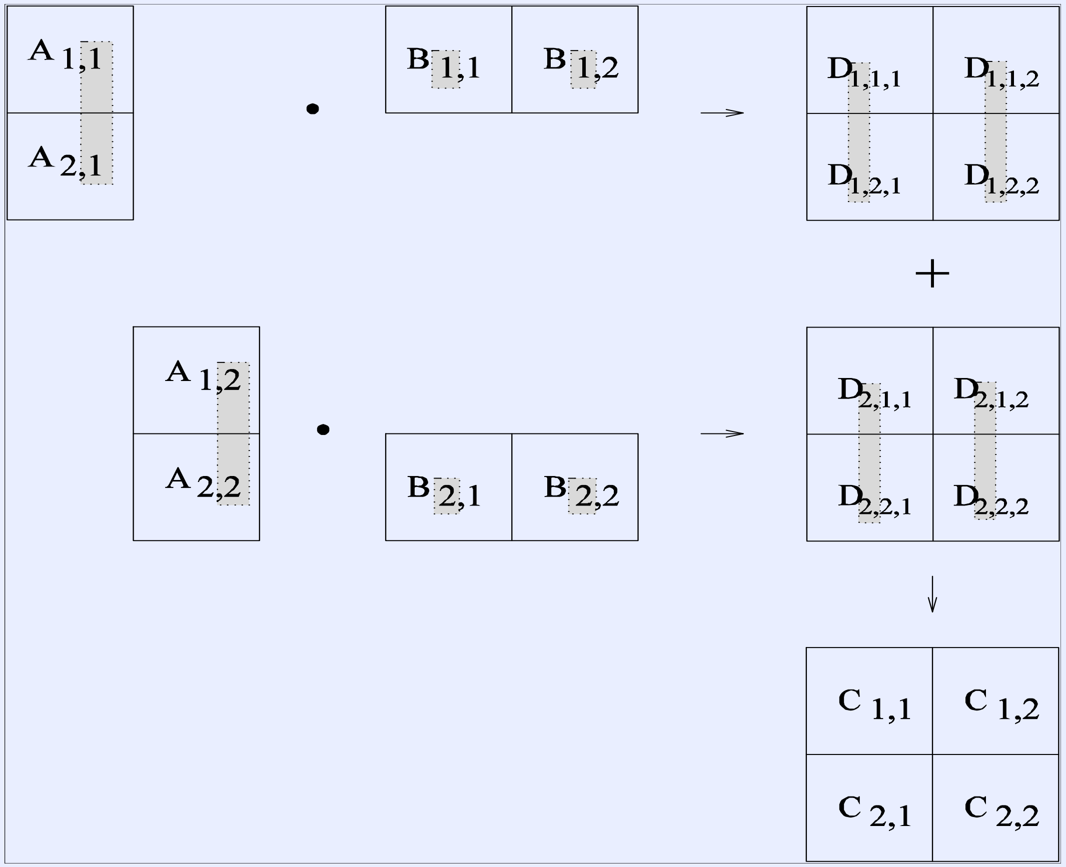
Intermediate Data Partitioning

- 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

example :dense matrix multiplication.

visualize this computation in terms of intermediate matrices ***D***.



Shape of the task dependency graph?

Intermediate Data Partitioning: Example

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

Stage I

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

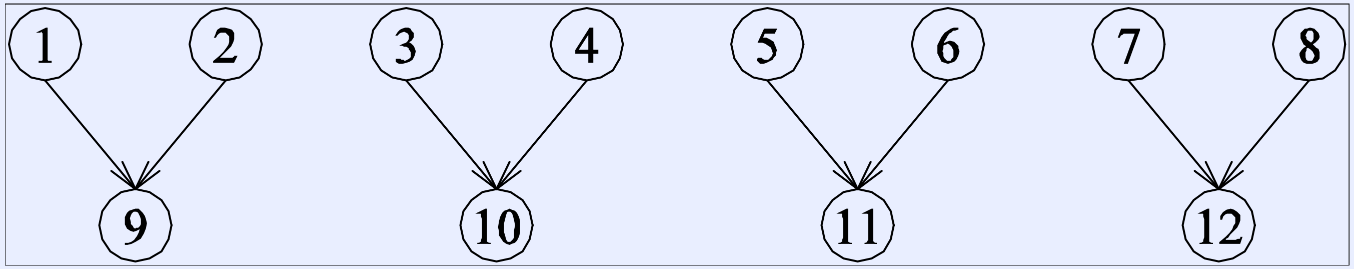
$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Stage II

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:



The Owner Computes Rule

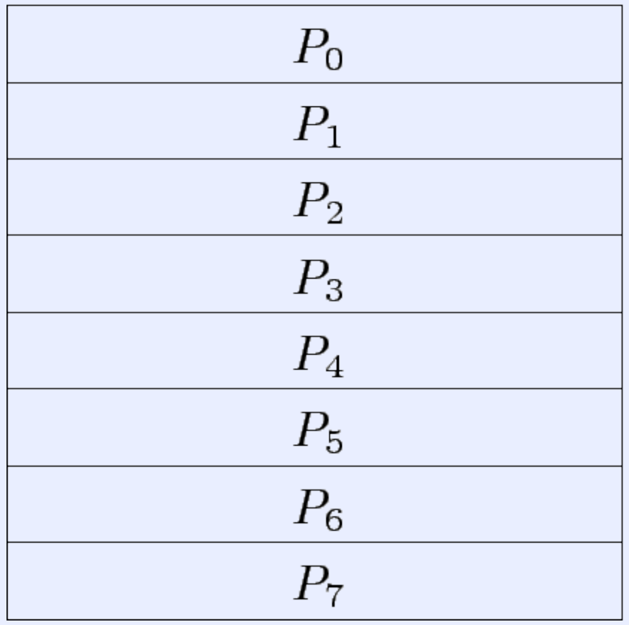
- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.
-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

Static Mapping

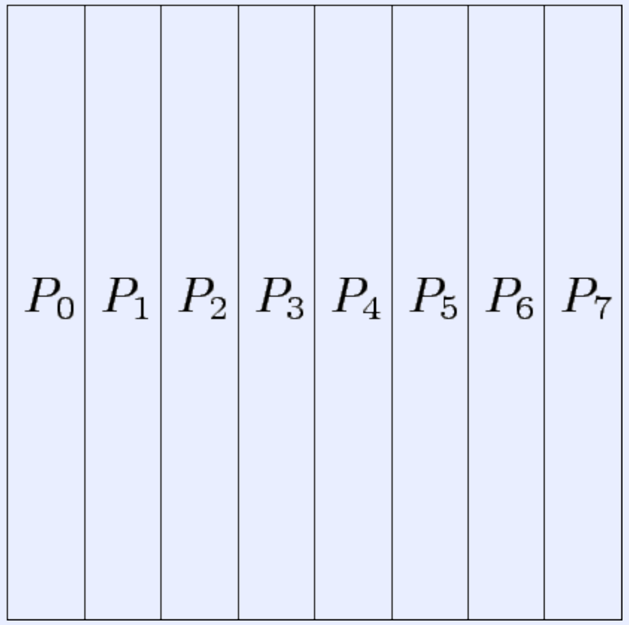
➢ Mappings based on data partitioning.

The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

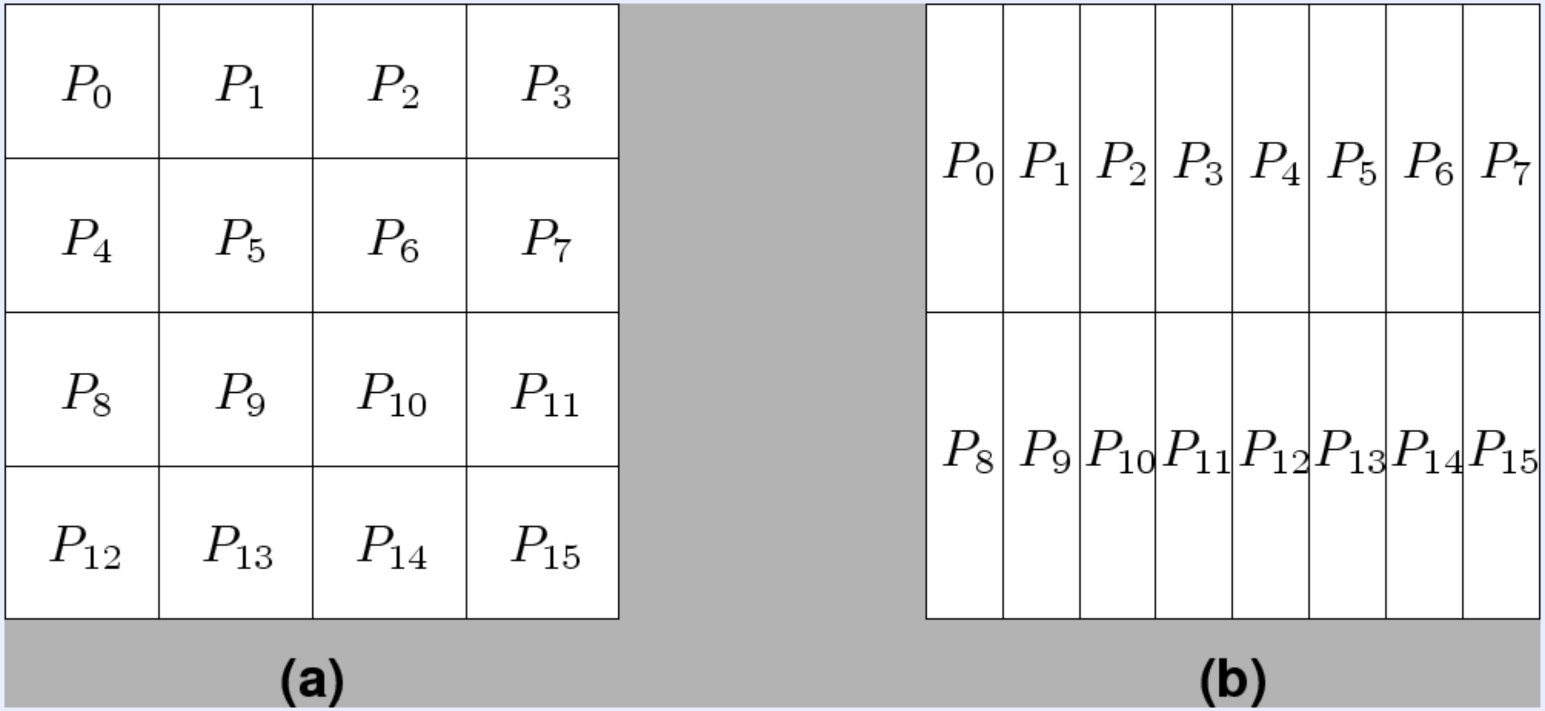
row-wise distribution



column-wise distribution



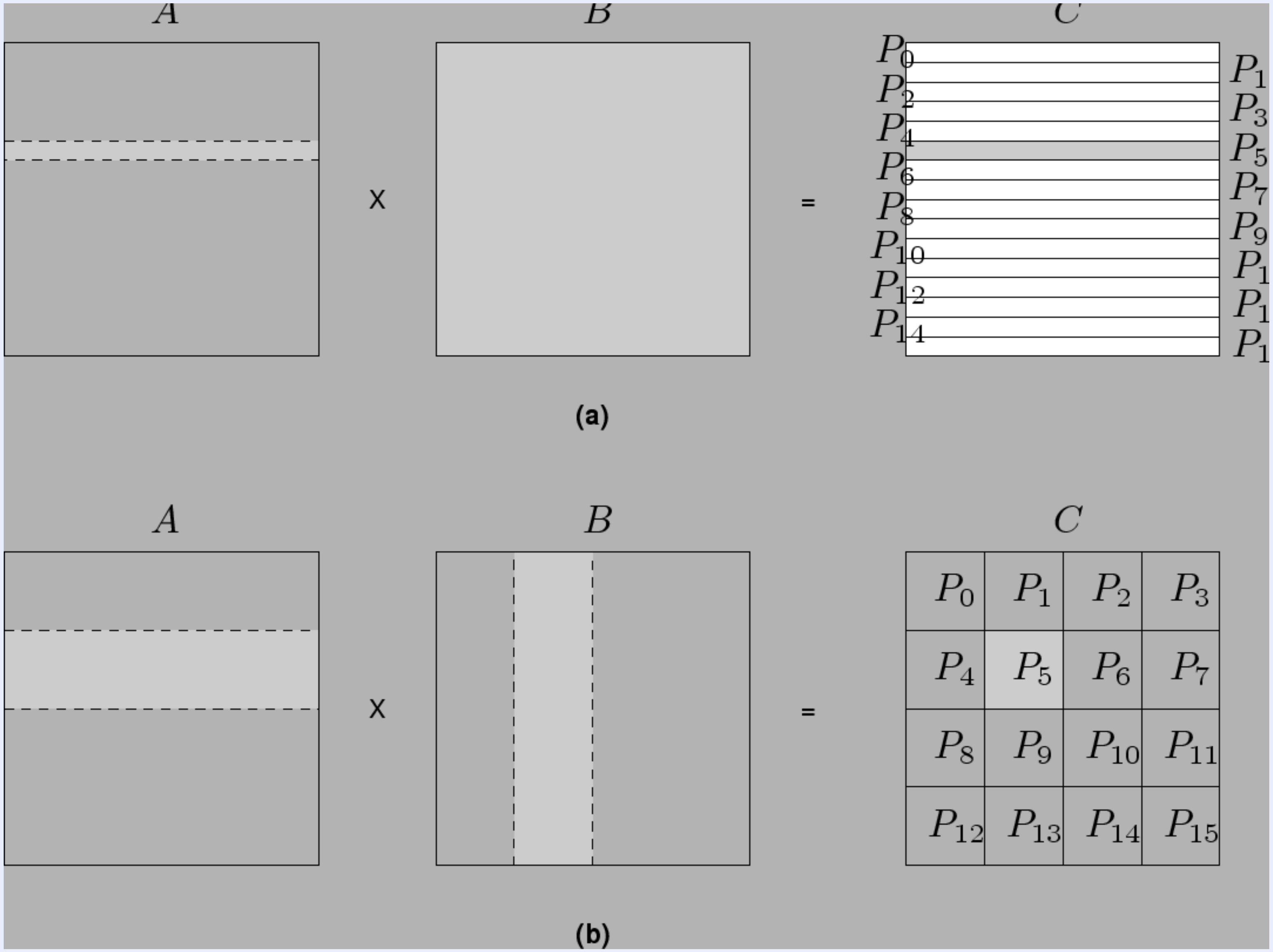
2D Block Array Distribution Schemes


P_0 P_1 P_2 P_3
P_4 P_5 P_6 P_7
P_8 P_9 P_{10} P_{11}
P_{12} P_{13} P_{14} P_{15}
(a)
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7
P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15}
(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***.
- The choice of precise decomposition scheme determines the associated communication overhead.
- In general, higher dimension decomposition allows the use of larger number of processes.

Data Sharing in Dense Matrix Multiplication

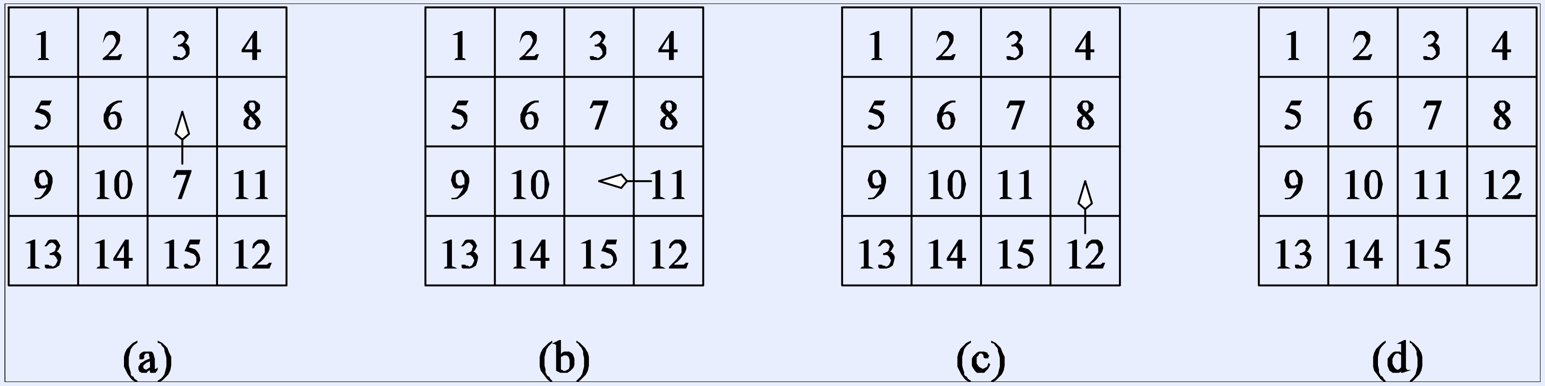


Exploratory Decomposition

- 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, theorem proving, game playing, etc.

Exploratory Decomposition: Example

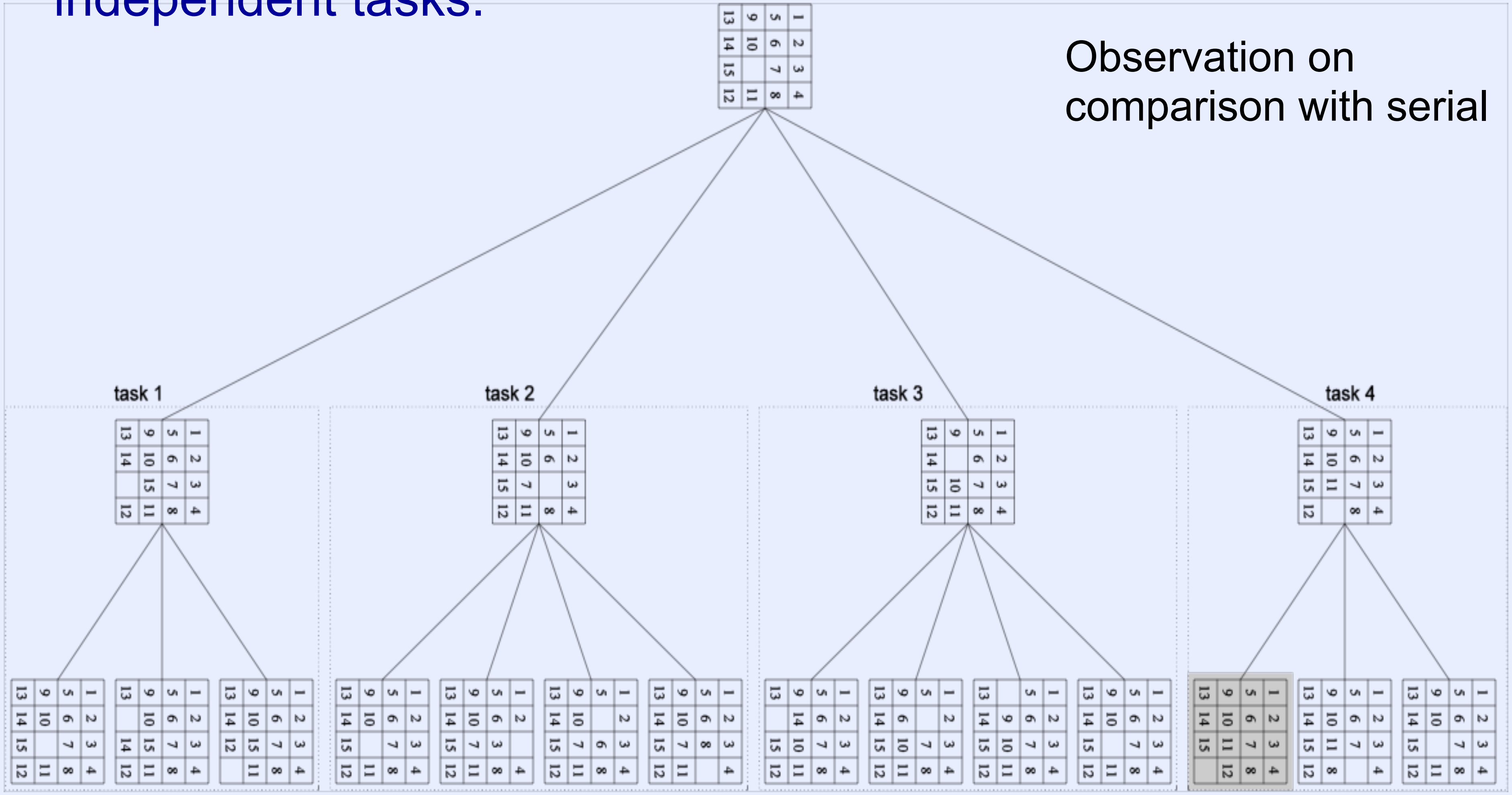
A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). Fig. show a sequence of three moves that transform a given initial state (a) to desired final state (d).



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

Exploratory Decomposition: Example

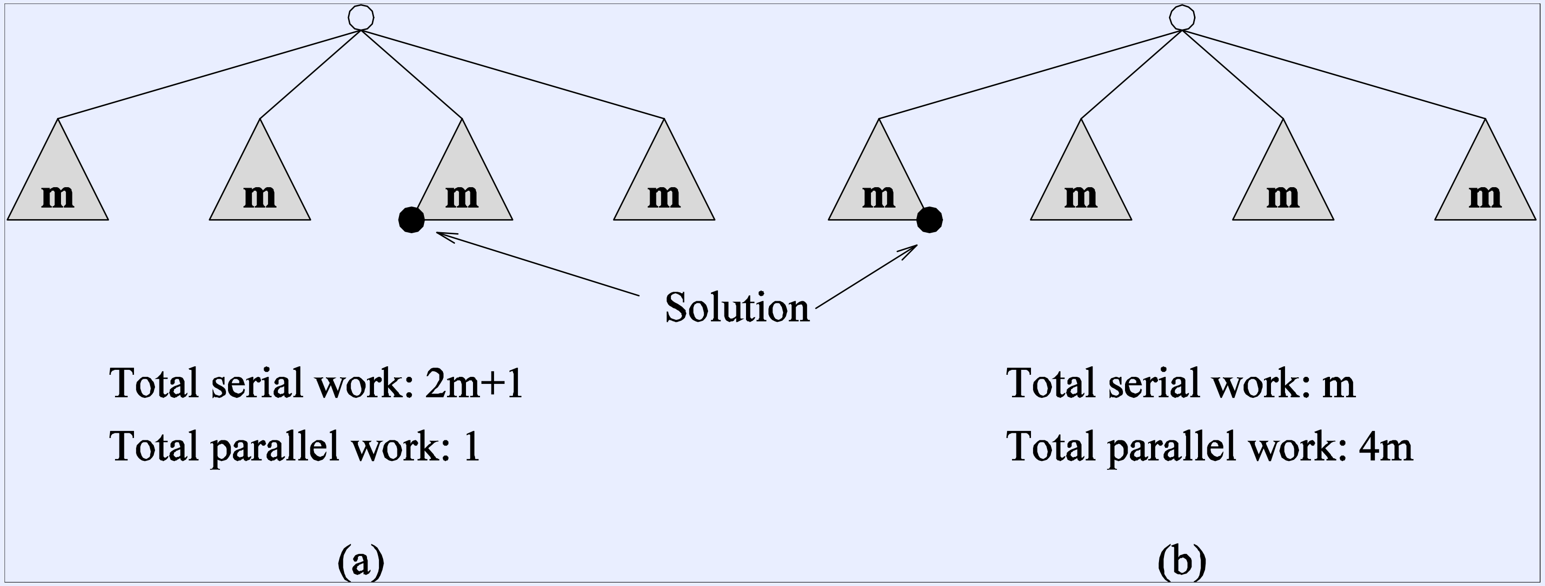
The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



Observation on comparison with serial

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.



Speculative Decomposition

- ➢ Applications where dependencies between tasks are not known a-priori and it is impossible to identify independent tasks.

- ➢ Generally two approaches to dealing with such applications: **conservative approaches**, which identify independent tasks only when they are guaranteed to not have dependencies, and, **optimistic approaches**, which 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.

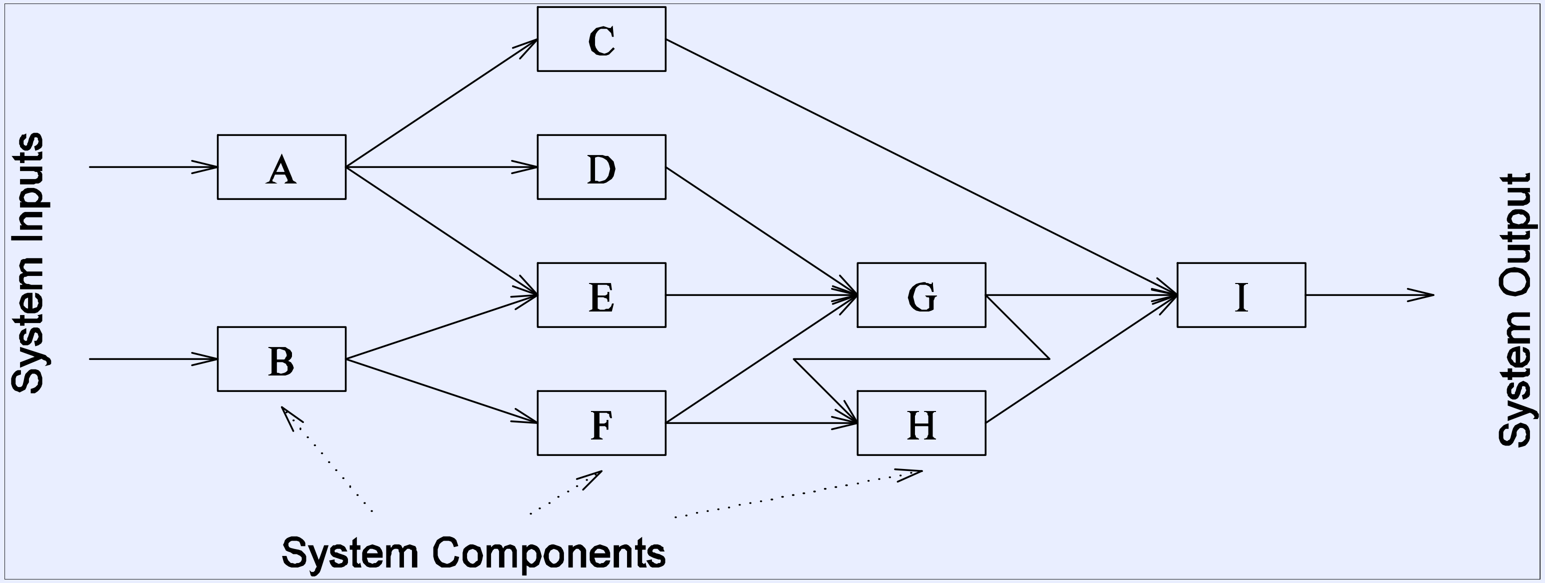
Speculative Decomposition: Example

Discrete event simulation

- ➢ The central data structure in a discrete event simulation is a time-ordered event list.
- ➢ Events are extracted precisely in time order, processed, and if required, resulting events are inserted back into the event list.
- ➢ Consider your day today as a discrete event system - you get up, get ready, drive to work, work, eat lunch, work some more, drive back, eat dinner, and sleep.
- ➢ Each of these events may be processed independently, however, in driving to work, you might meet with an unfortunate accident and not get to work at all.
- ➢ Therefore, an optimistic scheduling of other events will have to be rolled back.

Speculative Decomposition: Example

The task is to simulate the behavior of this network for various inputs and node delay parameters.



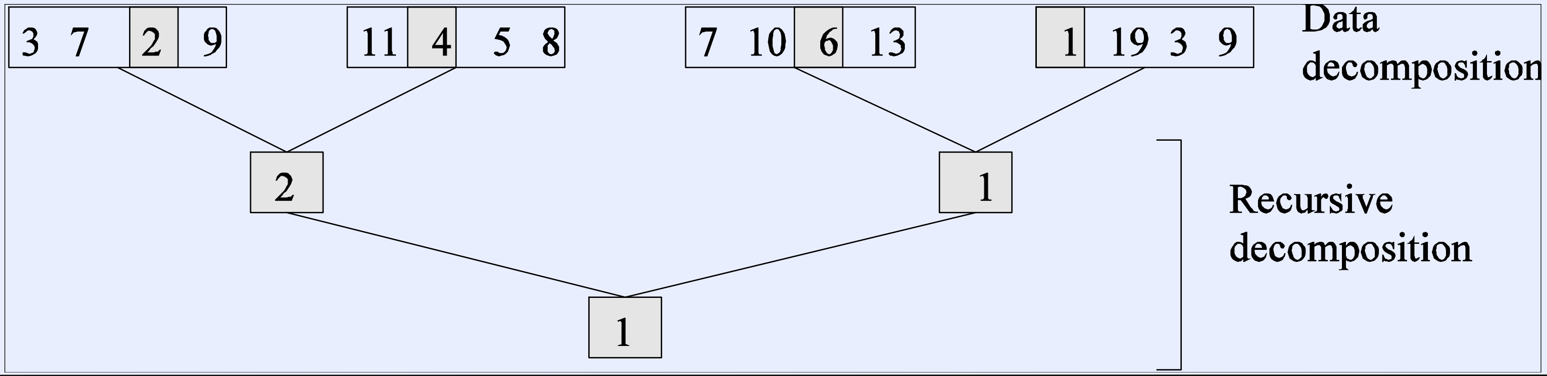
Hybrid Decompositions

mix of decomposition for decomposing a problem:

In quicksort, recursive decomposition alone limits concurrency (Why?).

A mix of data and recursive decompositions is more desirable.

- In discrete event simulation, there might be concurrency in task processing. A mix of speculative decomposition and data decomposition may work well.
- 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.

Example: discrete optimization problems, in which it is difficult to estimate the effective size of a state space.

Size of Data Associated with Tasks

- ➢ The size of data associated with a task may be small or large when viewed in the context of the size of the task.
- ➢ A small context of a task implies that an algorithm can easily communicate this task to other processes dynamically (e.g., the 15 puzzle).
- ➢ A large context ties the task to a process, or alternately, an algorithm may attempt to reconstruct the context at another processes as opposed to communicating the context of the task.

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.

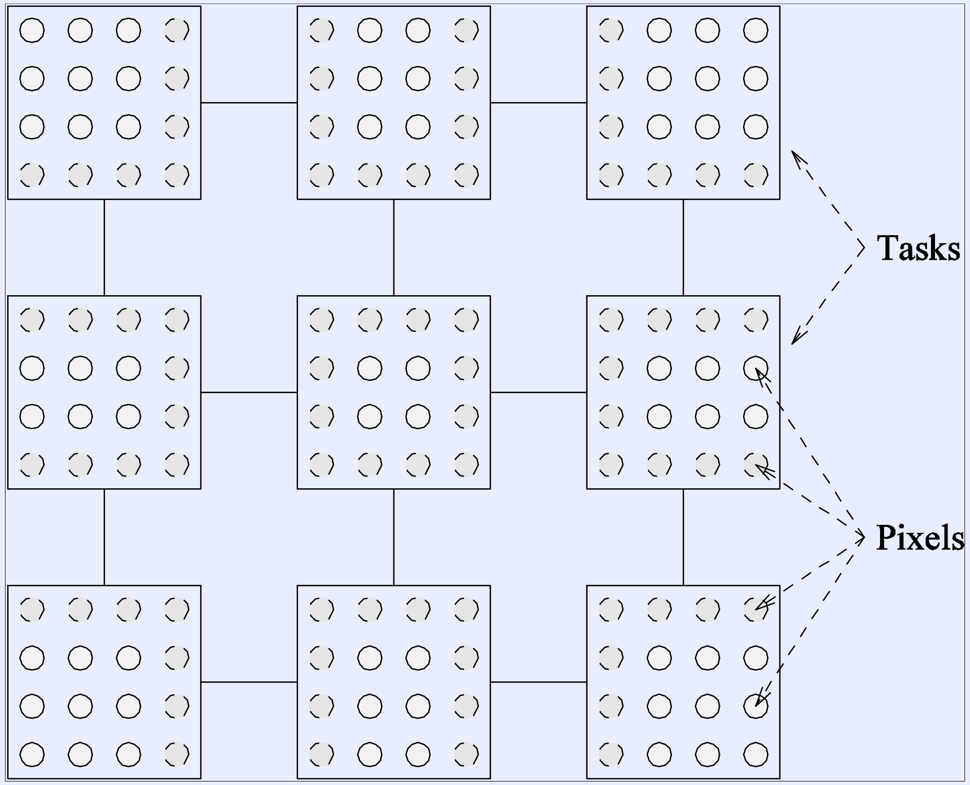
- **Dynamic interactions:** The timing or interacting tasks cannot be determined a-priori. These interactions are harder to code, especially using message passing APIs.

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.

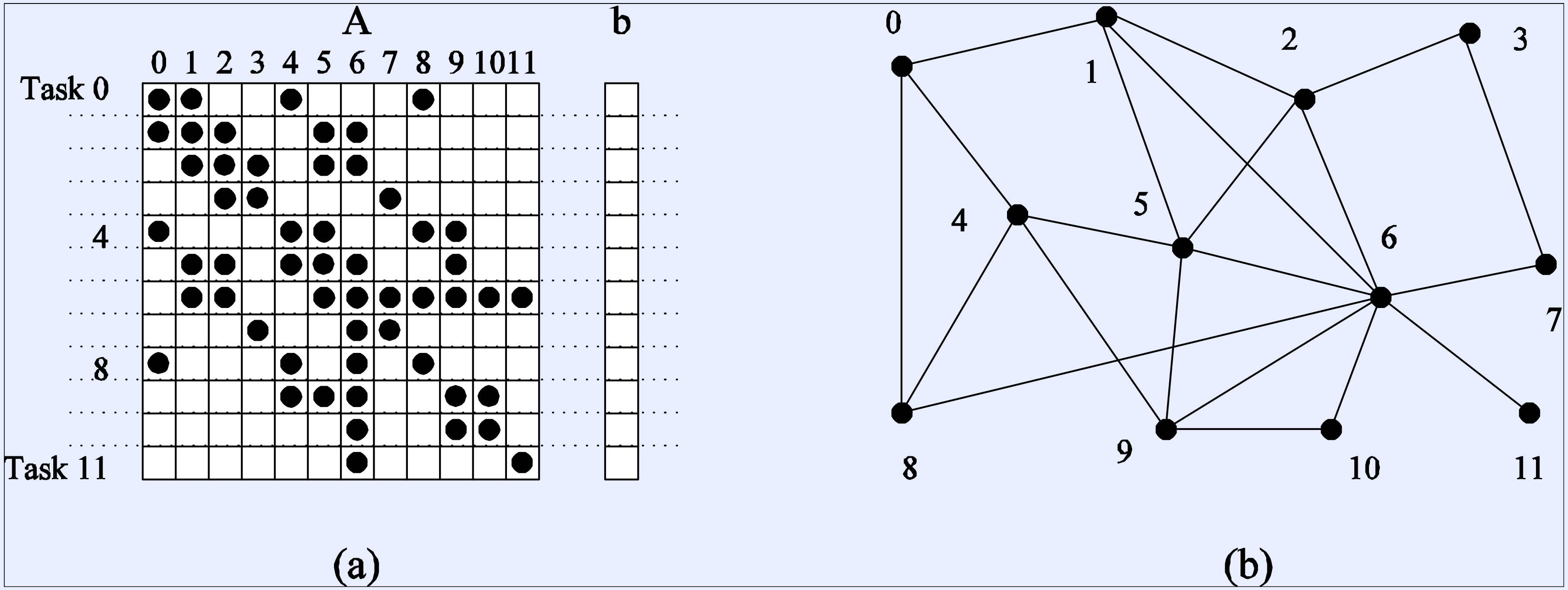
Characteristics of Task Interactions: Example

A simple example of a regular static 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.



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, since they require additional synchronization primitives.

Characteristics of Task Interactions

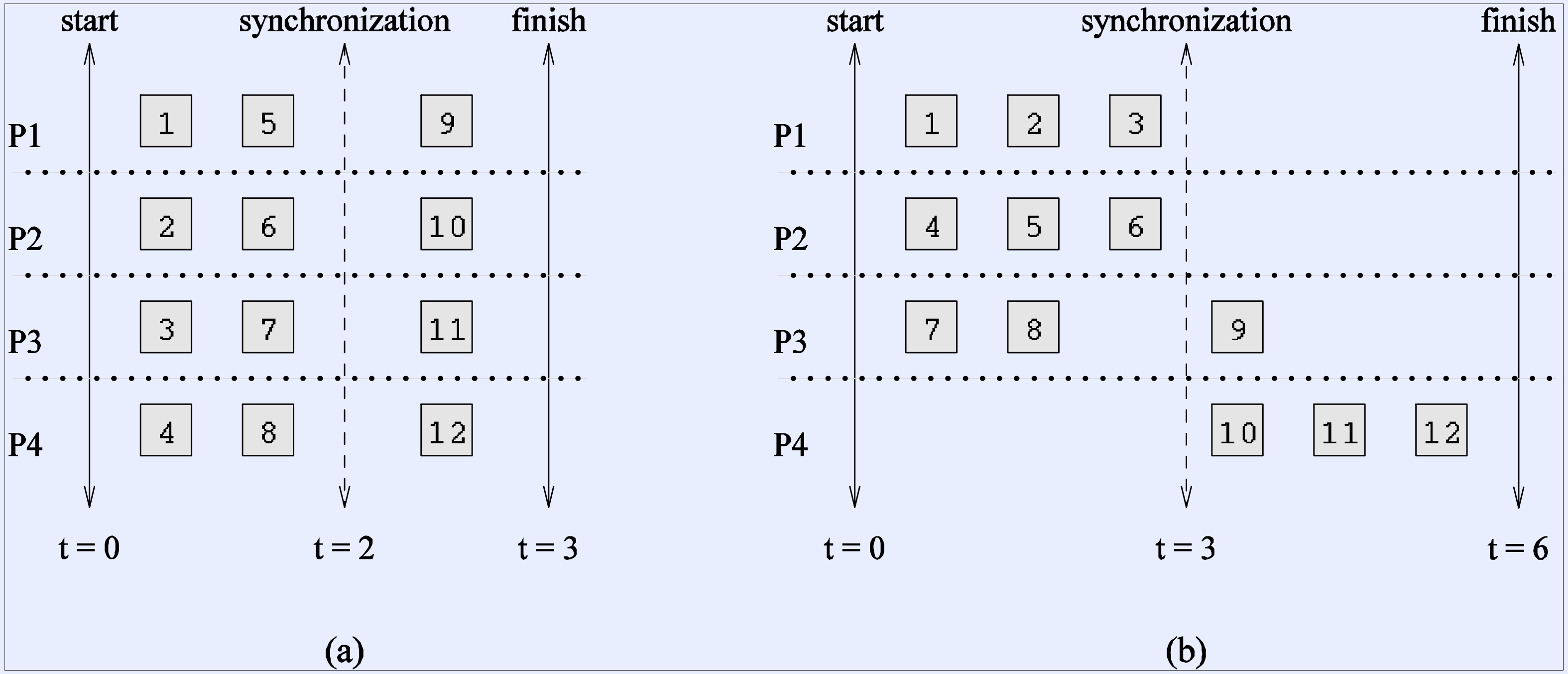
- Interactions may be one-way or two-way.
- A one-way interaction can be initiated and accomplished by one of the two interacting tasks.
- A two-way interaction requires participation from both tasks involved in an interaction.
- One way interactions are somewhat harder to code in message passing APIs.

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.

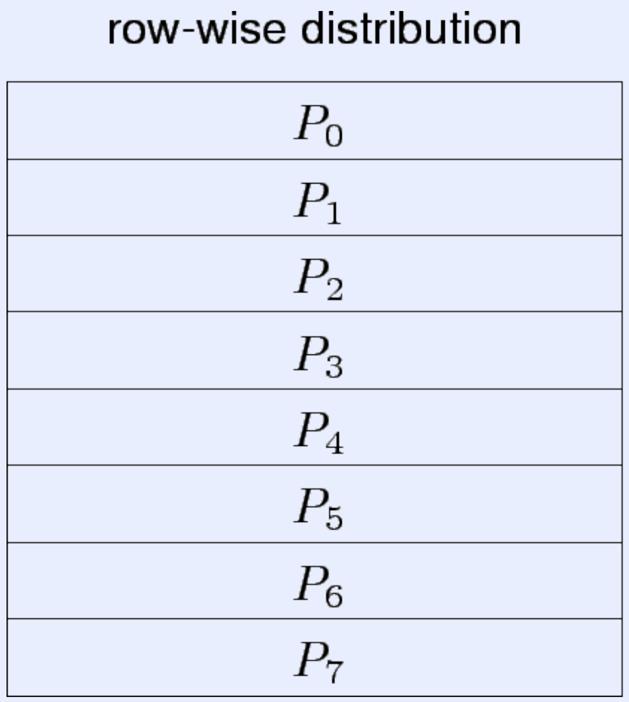
Other factors that determine the choice of techniques include the size of data associated with a task and the nature of underlying domain.

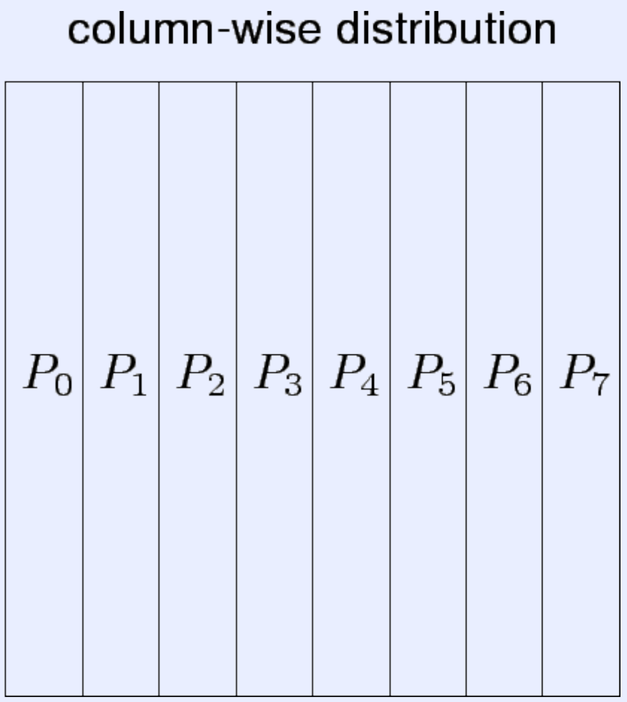
Schemes for Static Mapping

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

Mappings Based on Data Partitioning

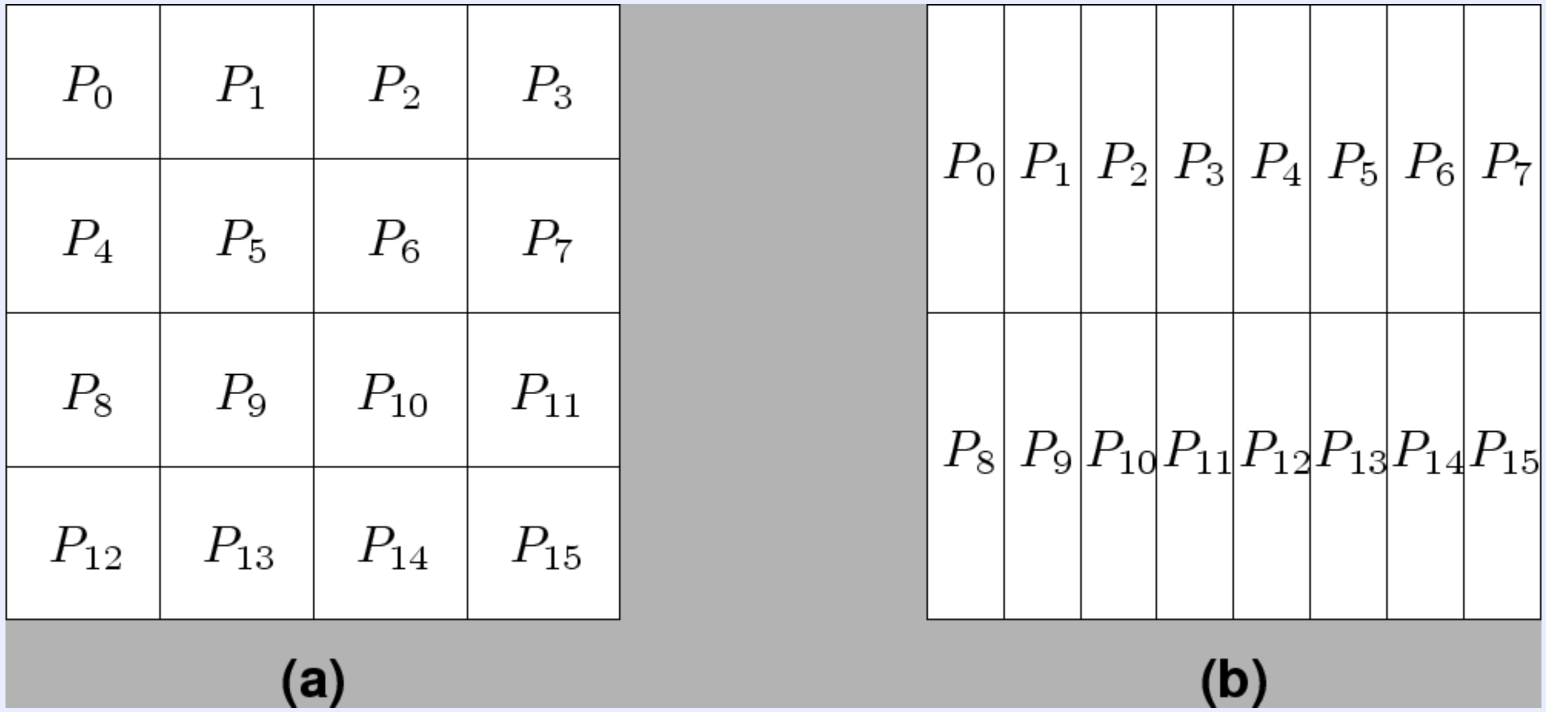
We can combine data partitioning with the ``owner-computes'' rule to partition the computation into subtasks. The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.





Block Array Distribution Schemes

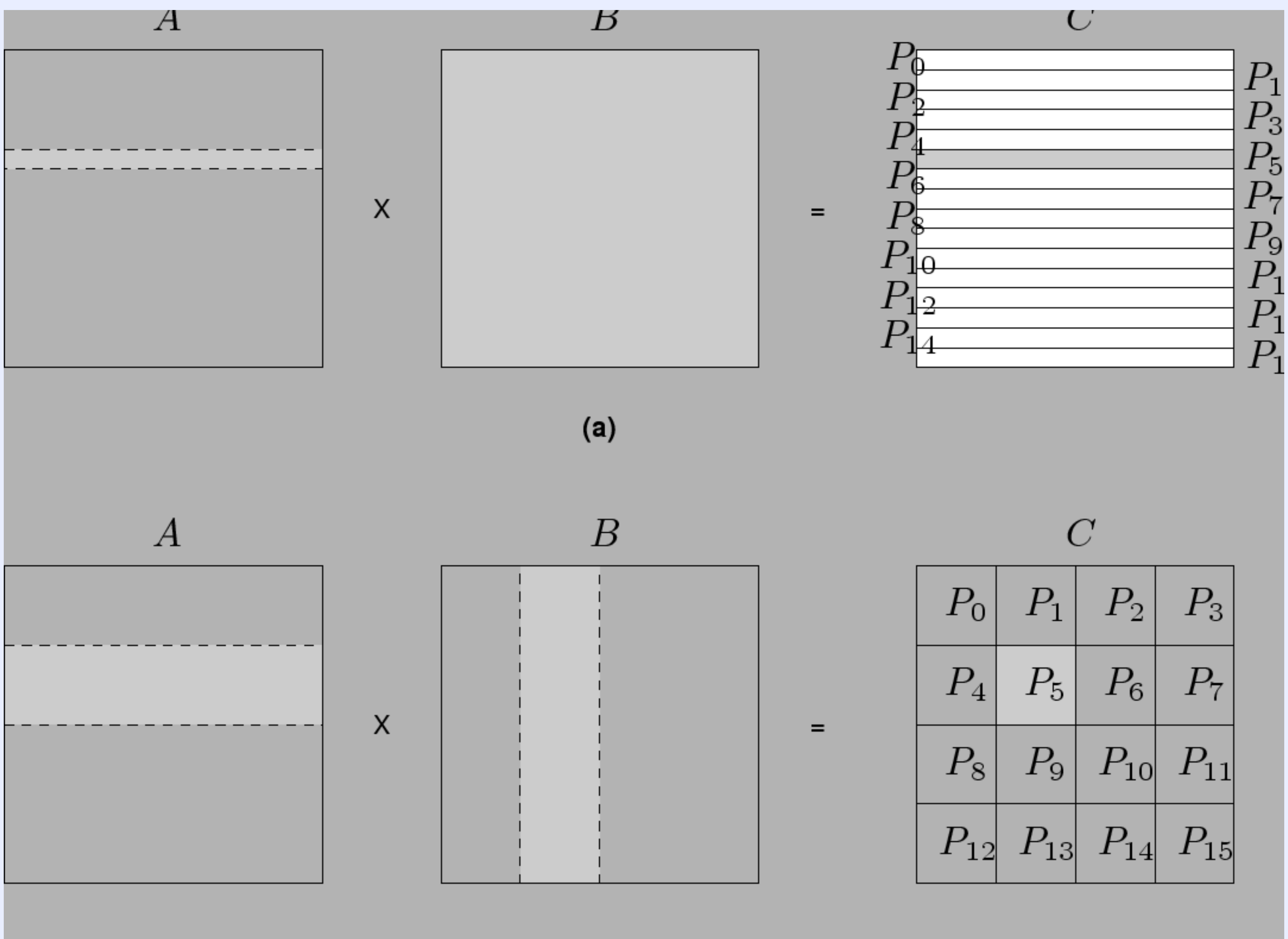
Block distribution schemes can be generalized to higher dimensions as well.


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


A
B
C
X
=
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{12} P_{14}
X
=
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15}

(a)

(b)

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: $A_{1,1} \rightarrow L_{1,1}U_{1,1}$

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

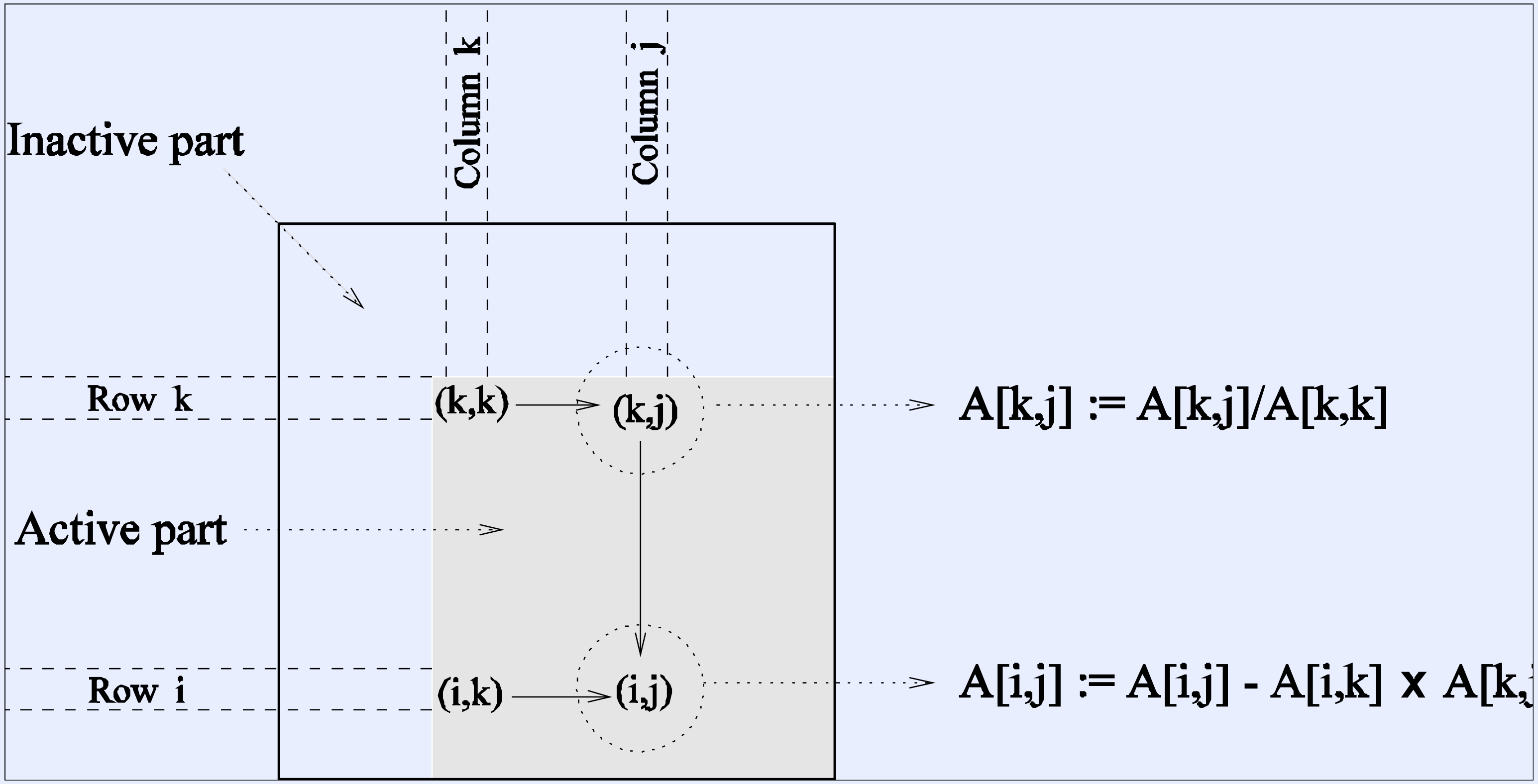
14: $A_{3,3} \rightarrow L_{3,3}U_{3,3}$

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 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.



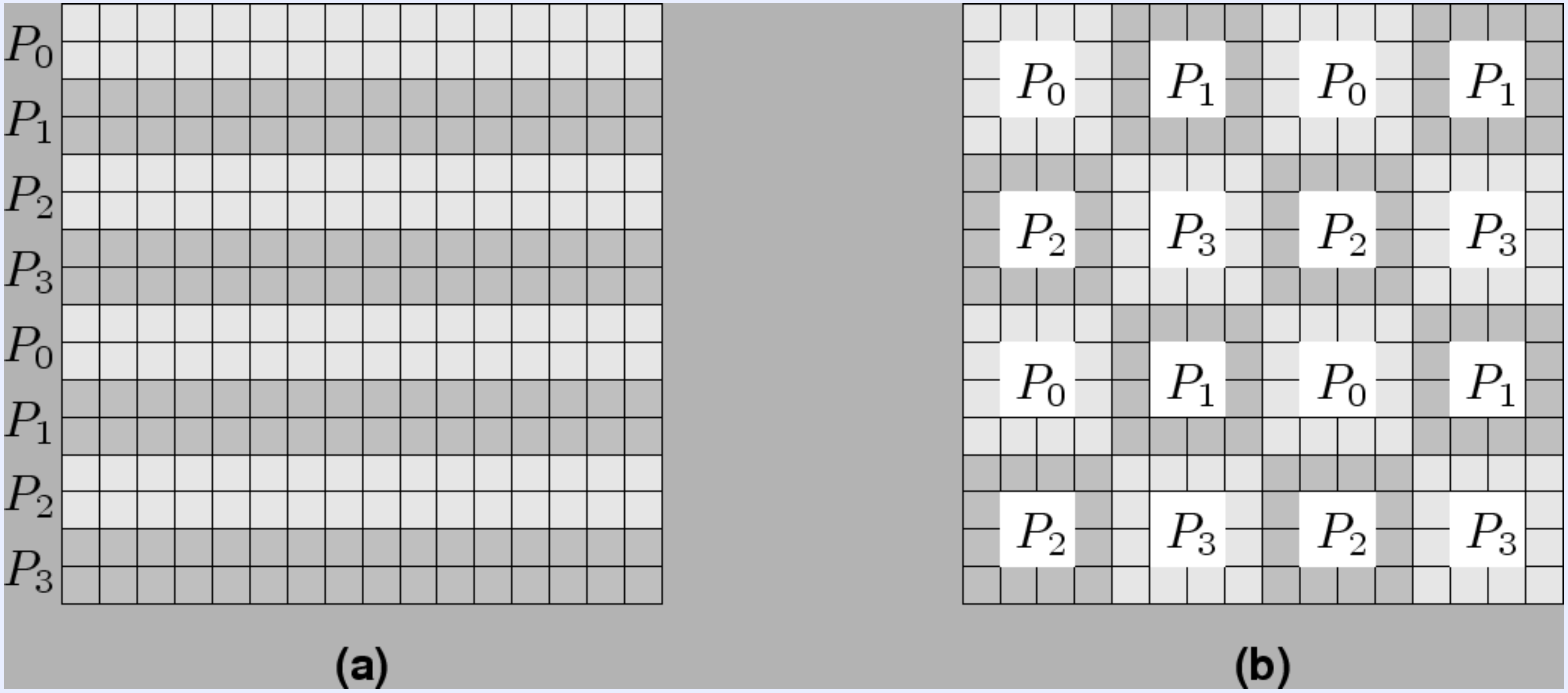
Block-Cyclic Distribution: Examples

One- and two-dimensional block-cyclic distributions among 4 processes.

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

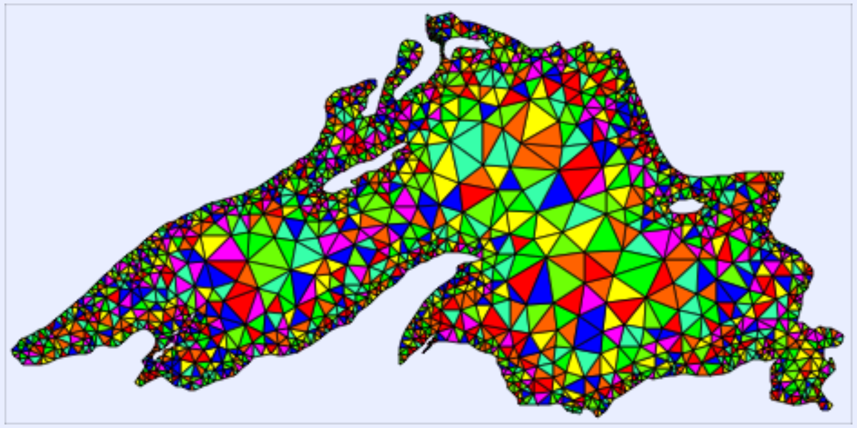
- A cyclic distribution is a special case in which block size is one.
- A block distribution is a special case in which block size is ***n/p*** , where ***n*** is the dimension of the matrix and ***p*** is the number of processes.



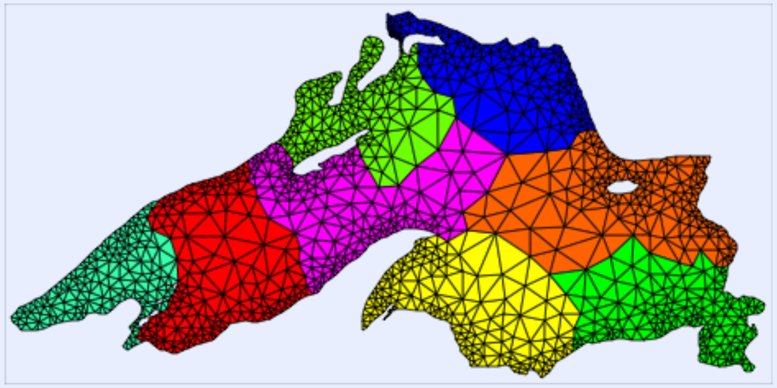
Graph Partitioning Dased Data Decomposition

- 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.

Partitioning the Graph of Lake Superior



Random Partitioning



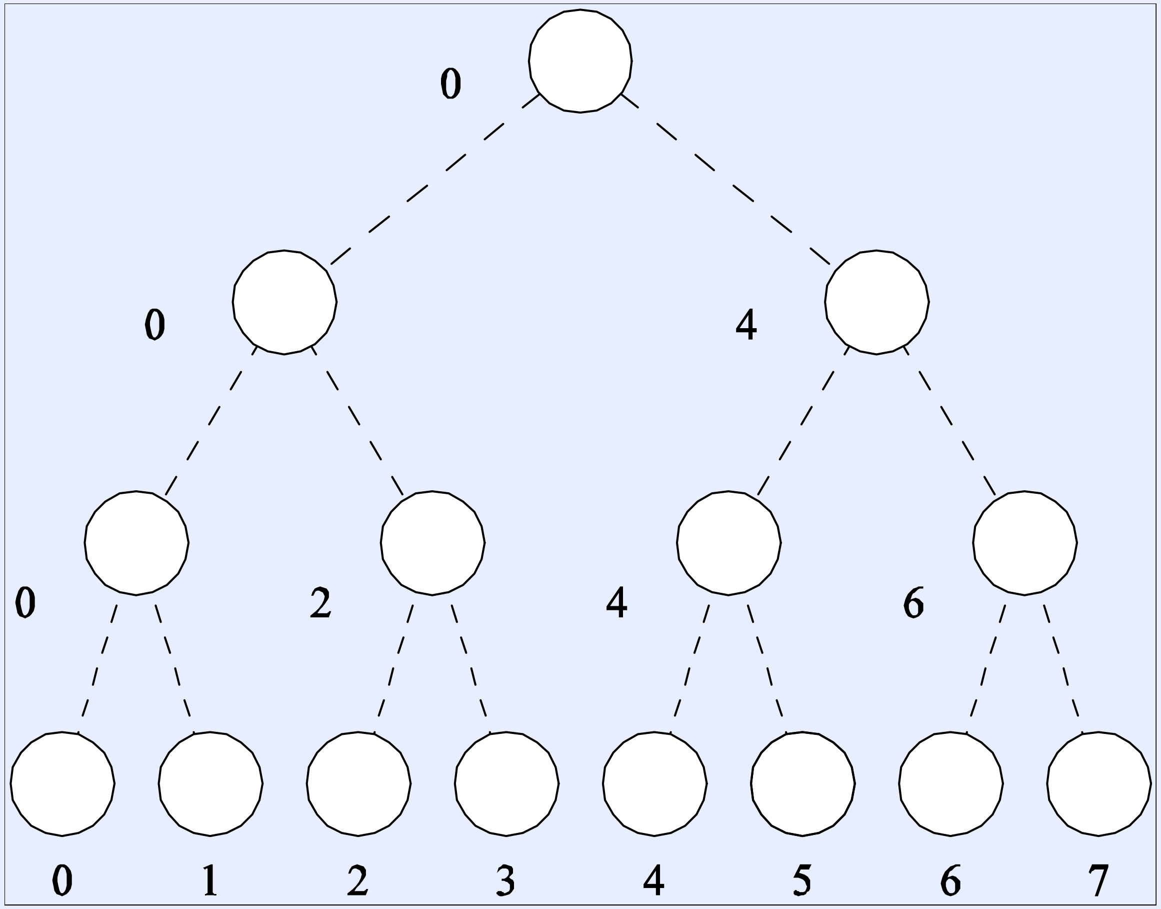
Partitioning for minimum edge-cut.

Mappings Based on Task Paritioning

- 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.

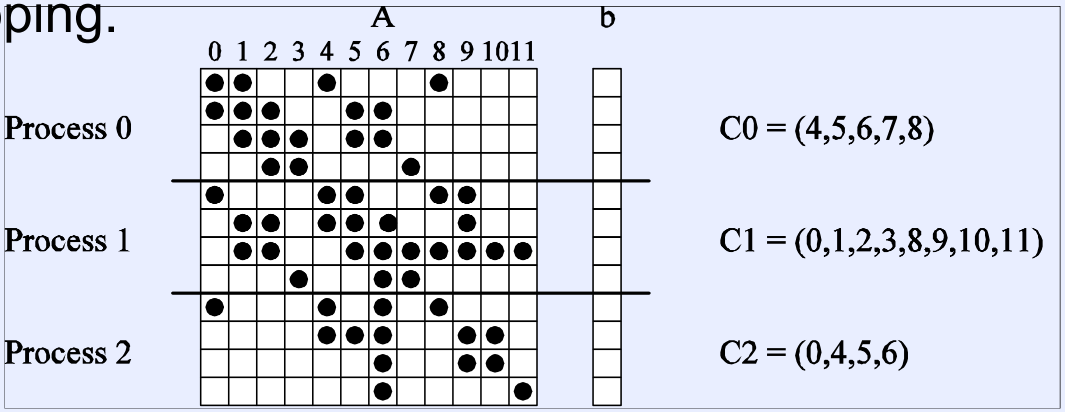
Task Paritioning: Mapping a Binary Tree Dependency Graph

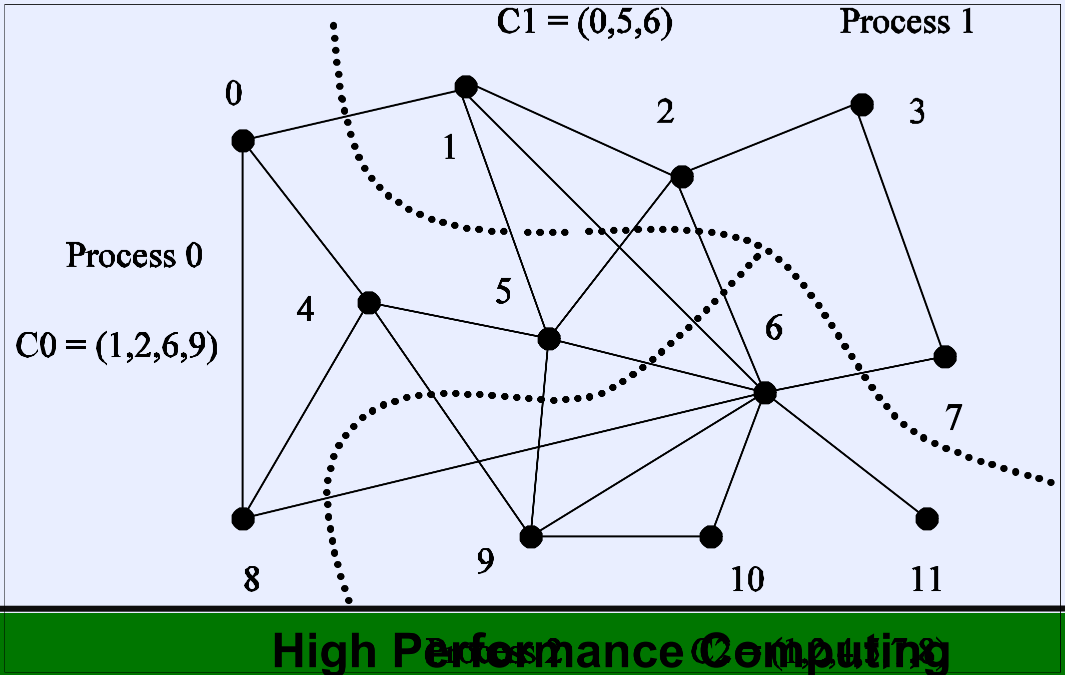
Example illustrates the dependency graph of one view of quick-sort and how it can be assigned to processes in a hypercube.



Task Paritioning: Mapping a Sparse Graph

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

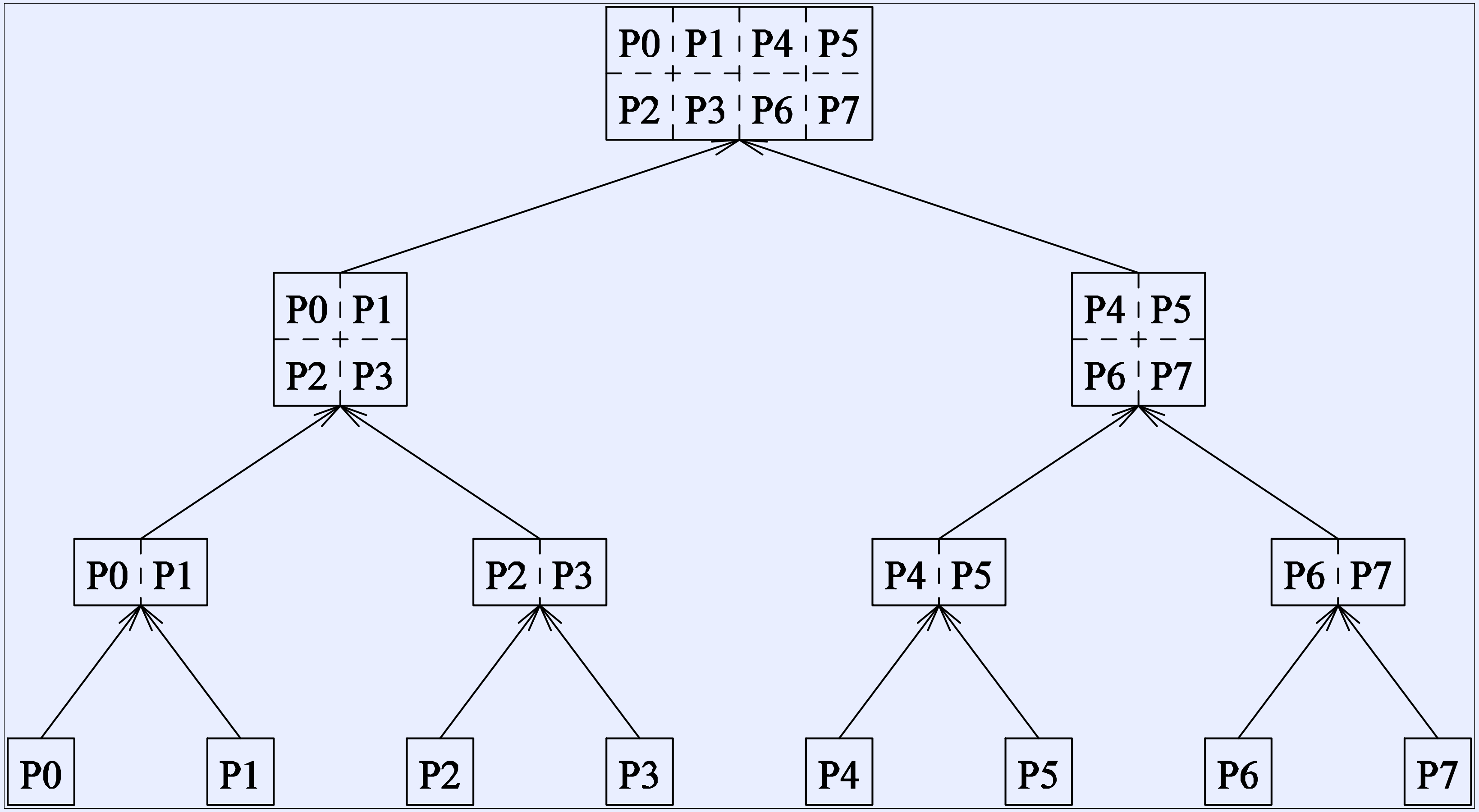




Hierarchical Mappings

- 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.

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.

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.
- Overlap interactions with other interactions.

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-Comsumer Model: A stream of data is passed through a succession of processes, each of which perform some task on it.
- Hybrid Models: A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.

Case Studies

Vector Dot Product

➢ Multiplication of 2 vectors followed by Summation

A[i]		B[i]		A[i] * B[i]
X_1		Y_1		X_1* Y_1
X_2		Y_2		X_2* Y_2
X_3	●	Y_3	=	X_3* Y_3
X_4		Y_4		X_4* Y_4
X_5		Y_5		X_5* Y_5
… …		… …		… …
X_n		Y_n		X_n* Y_n

OpenMP Dot Product

```
#include <omp.h>
main ()  {
int   i, n, chunk;
float a[16], b[16], result;
n = 16;
chunk = 4;
result = 0.0;
for (i=0; i < n; i++)
  {
    a[i] = i * 1.0;
    b[i] = i * 2.0;
  }
#pragma omp parallel for default(shared) private(i)  \
    schedule(static,chunk) reduction(+:result)
  for (i=0; i < n; i++)
      result = result + (a[i] * b[i]);
printf("Final result= %f\n",result);
}
```

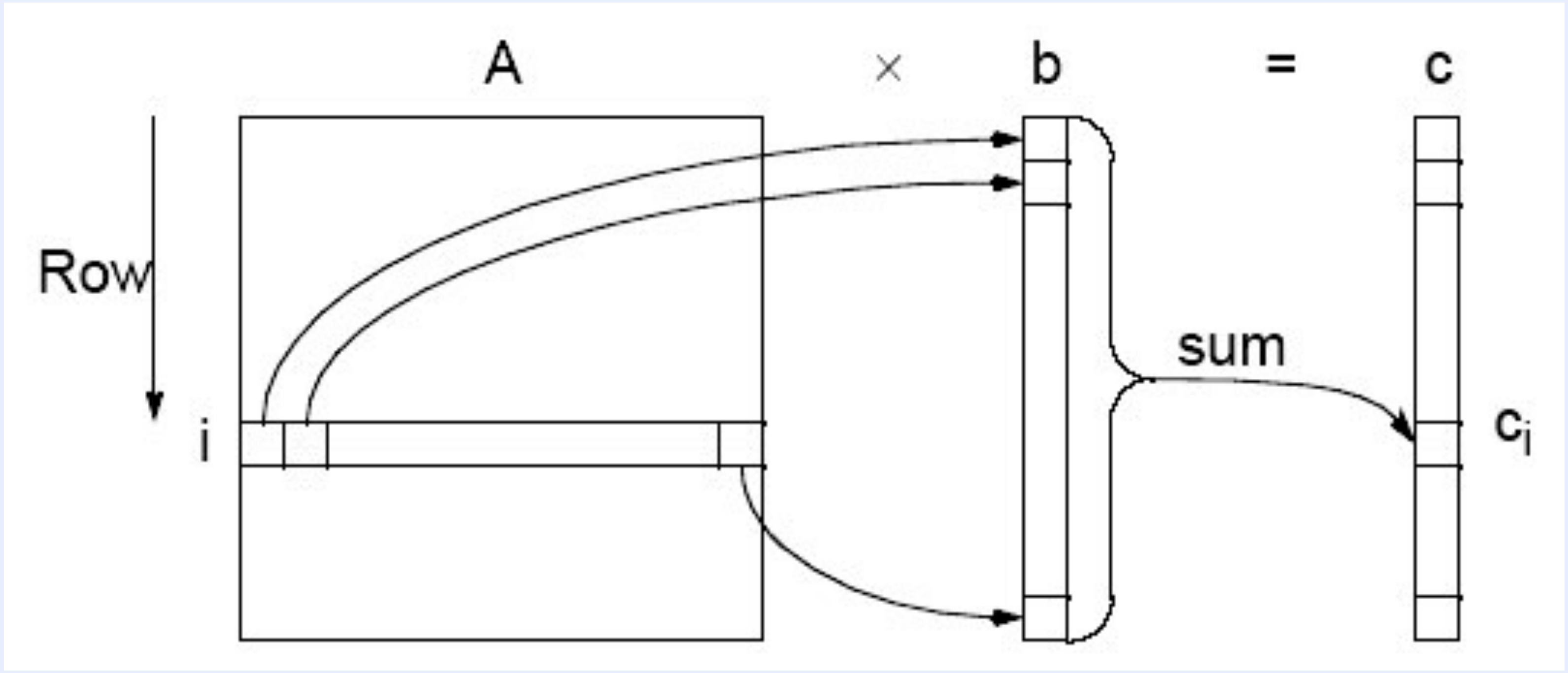
Reduction example with summation where the result of the reduction operation stores the dot product of two vectors

$\sum a[i]*b[i]$

Matrix Vector Multiplication

$$\begin{array}{c|c}
 & \text{Column} \xrightarrow{\hspace{10em}} \\
\hline
\text{Row}\downarrow & \begin{bmatrix}
a_{0,0} & a_{0,1} & \cdots & a_{0,m-2} & a_{0,m-1} \\
a_{1,0} & a_{1,1} & \cdots & a_{1,m-2} & a_{1,m-1} \\
\vdots & \vdots & & \vdots & \vdots \\
a_{n-2,0} & a_{n-2,1} & \cdots & a_{n-2,m-2} & a_{n-2,m-1} \\
a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,m-2} & a_{n-1,m-1}
\end{bmatrix}
\end{array}$$

Matrix-Vector Multiplication **c = A xb**



Implementing Matrix Multiplication- Sequential Code

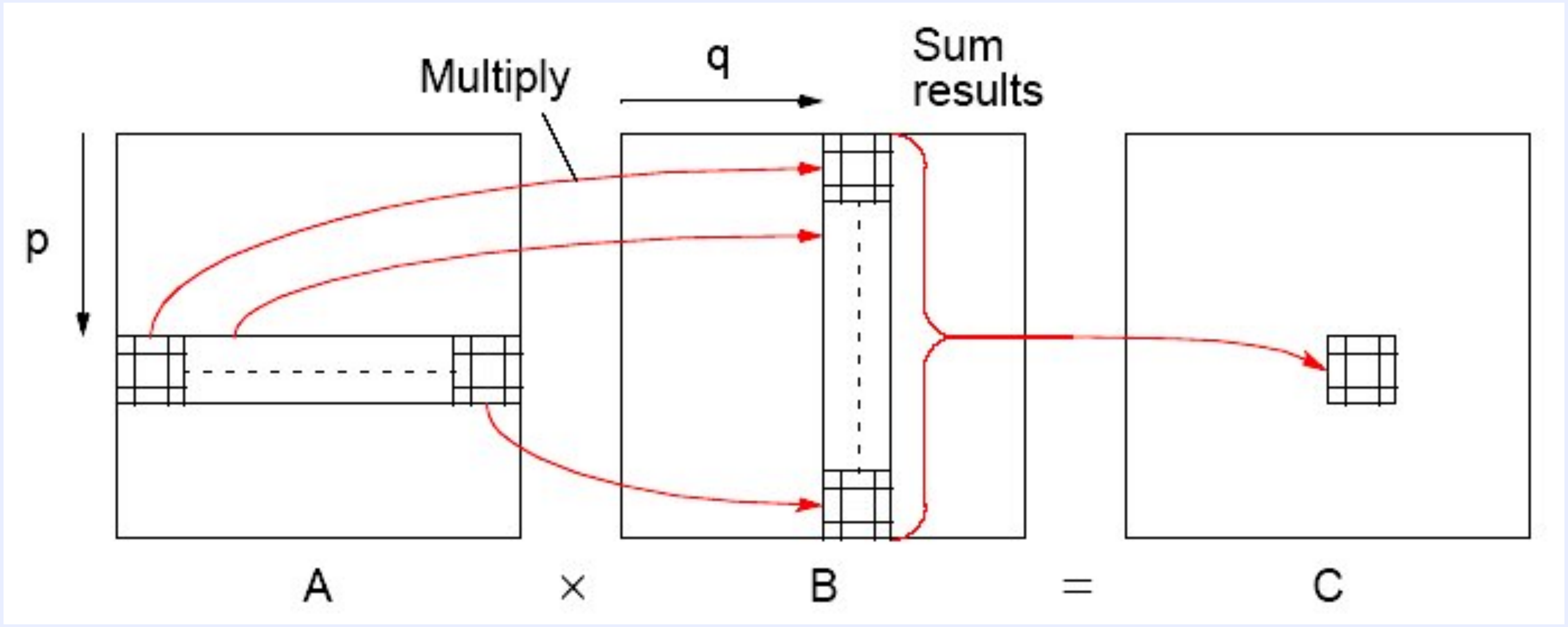
Assume throughout that the matrices are square (*n* x *n* matrices).
The sequential code to compute **A** x **B** could simply be

```
        for (i = 0; i < n; i++)
                for (j = 0; j < n; j++) {
                        c[i][j] = 0;
                        for (k = 0; k < n; k++)
                                c[i][j] = c[i][j] + a[i][k] * b[k][j];
                  }
```

This algorithm requires n^3 multiplications and n^3 additions, leading to a sequential time complexity of $O(n^3)$.
Very easy to parallelize. (**Problem ?**)

Block Matrix Multiplication

Partitioning into sub-matricies



$$\left[\begin{array}{cc|cc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array}\right]$$

$$\overset{A_{0,0}}{\begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}} \times \overset{B_{0,0}}{\begin{bmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{bmatrix}} + \overset{A_{0,1}}{\begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}} \times \overset{B_{1,0}}{\begin{bmatrix} b_{2,0} & b_{2,1} \\ b_{3,0} & b_{3,1} \end{bmatrix}}$$

$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1} \end{bmatrix} + \begin{bmatrix} a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

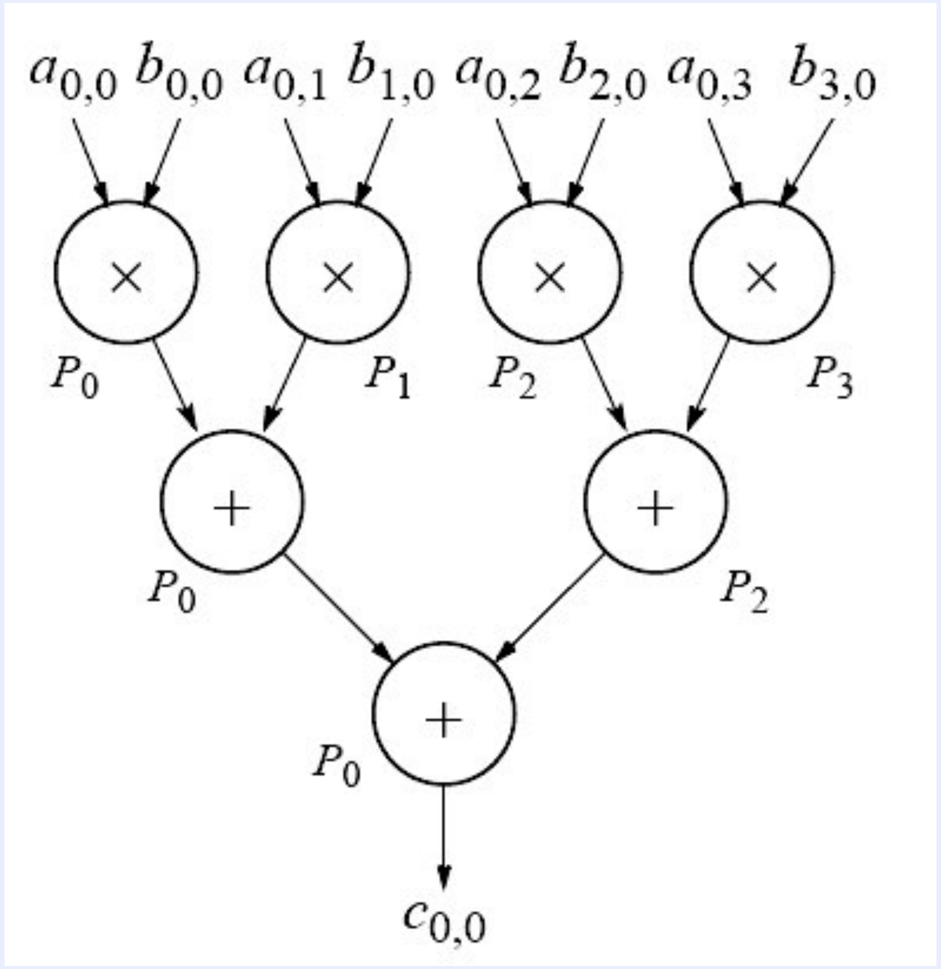
$$= \begin{bmatrix} a_{0,0}b_{0,0}+a_{0,1}b_{1,0}+a_{0,2}b_{2,0}+a_{0,3}b_{3,0} & a_{0,0}b_{0,1}+a_{0,1}b_{1,1}+a_{0,2}b_{2,1}+a_{0,3}b_{3,1} \\ a_{1,0}b_{0,0}+a_{1,1}b_{1,0}+a_{1,2}b_{2,0}+a_{1,3}b_{3,0} & a_{1,0}b_{0,1}+a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1} \end{bmatrix}$$

$$= C_{0,0}$$

Performance Improvement

Using tree construction *n* numbers can be added in O(log *n)* steps (using *n*³ *processors)*:

Final Assignment ?


$a_{0,0}$ $b_{0,0}$ $a_{0,1}$ $b_{1,0}$ $a_{0,2}$ $b_{2,0}$ $a_{0,3}$ $b_{3,0}$
×
P_0 P_1 P_2 P_3
+
P_0 P_2
P_0
$c_{0,0}$

OpenMP Matrix Multiplication

```
#include <stdio.h>
#include <omp.h>

/* Main Program */

main()
{
     int   NoofRows_A, NoofCols_A, NoofRows_B, NoofCols_B, i, j, k;
    NoofRows_A = NoofCols_A = NoofRows_B = NoofCols_B = 4;
    float Matrix_A[NoofRows_A][NoofCols_A];
    float Matrix_B[NoofRows_B][NoofCols_B];
    float Result[NoofRows_A][NoofCols_B];

        for (i = 0; i < NoofRows_A; i++) {
                for (j = 0; j < NoofCols_A; j++)
                        Matrix_A[i][j] = i + j;
        }
        /* Matrix_B Elements */
        for (i = 0; i < NoofRows_B; i++) {
                for (j = 0; j < NoofCols_B; j++)
                        Matrix_B[i][j] = i + j;
        }
```

Initialize the two Matrices A[][] & B[][] with sum of their index values

OpenMP Matrix Multiplication

```
for (i = 0; i < NoofRows_A; i++) {
          for (j = 0; j < NoofCols_A; j++)
                    printf("%f \t", Matrix_A[i][j]);
          printf("\n");
 }
 printf("The Matrix_B Is \n");
 for (i = 0; i < NoofRows_B; i++) {
          for (j = 0; j < NoofCols_B; j++)
                    printf("%f \t", Matrix_B[i][j]);
          printf("\n");
}
 for (i = 0; i < NoofRows_A; i++) {
          for (j = 0; j < NoofCols_B; j++) {
                    Result[i][j] = 0.0;
          }
 }
#pragma omp parallel for private(j,k)
 for (i = 0; i < NoofRows_A; i = i + 1)
          for (j = 0; j < NoofCols_B; j = j + 1)
                    for (k = 0; k < NoofCols_A; k = k + 1)
                              Result[i][j] = Result[i][j] + Matrix_A[i][k] * Matrix_B[k][j];
printf("\nThe Matrix Computation Result Is \n");
```

Print the Matrices for debugging purposes

Initialize the results matrix with 0.0

Using OpenMP parallel For directive:
Calculate the product of the two matrices
Loadbalancing is done based on the values of OpenMP environment variables and the number of threads

OpenMP Matrix Multiplicaton

```
 for (i = 0; i < NoofRows_A; i = i + 1) {
                  for (j = 0; j < NoofCols_B; j = j +
1)
                           printf("%f ", Result[i][j]);
                  printf("\n");
         }
}
```

Implementing Matrix Multiplication

- With n processors (and n x n matrices),
  - Time complexity with n^2 processors
  - Time complexity with n^3 processors
- O(log n) lower bound for parallel matrix multiplication.
- **Final Assignment?**