Chapter 3
Principles of Parallel Algorithm Design

(Selected slides)

A. Grama, A. Gupta, G. Karypis, and V. Kumar

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

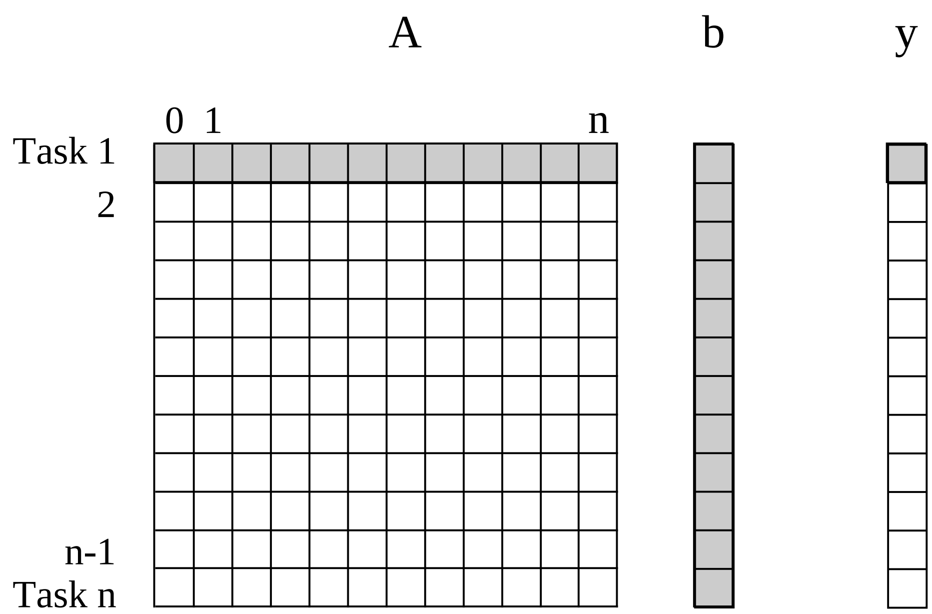
Chapter Overview: Algorithms and Concurrency

- Preliminaries
- Decomposition Techniques
- Characteristics of Tasks and Interactions
- Mapping Techniques for Load Balancing
- Methods for Minimizing Interaction Overheads
- Parallel Algorithm Design Models

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 docomposed into tasks in many different ways.
- Tasks may be of same or different sizes.
- A decomposition can be represented by a directed graph, where nodes correspond to tasks and edges indicate 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.

Example: Database Query Processing

Consider the execution of the query:

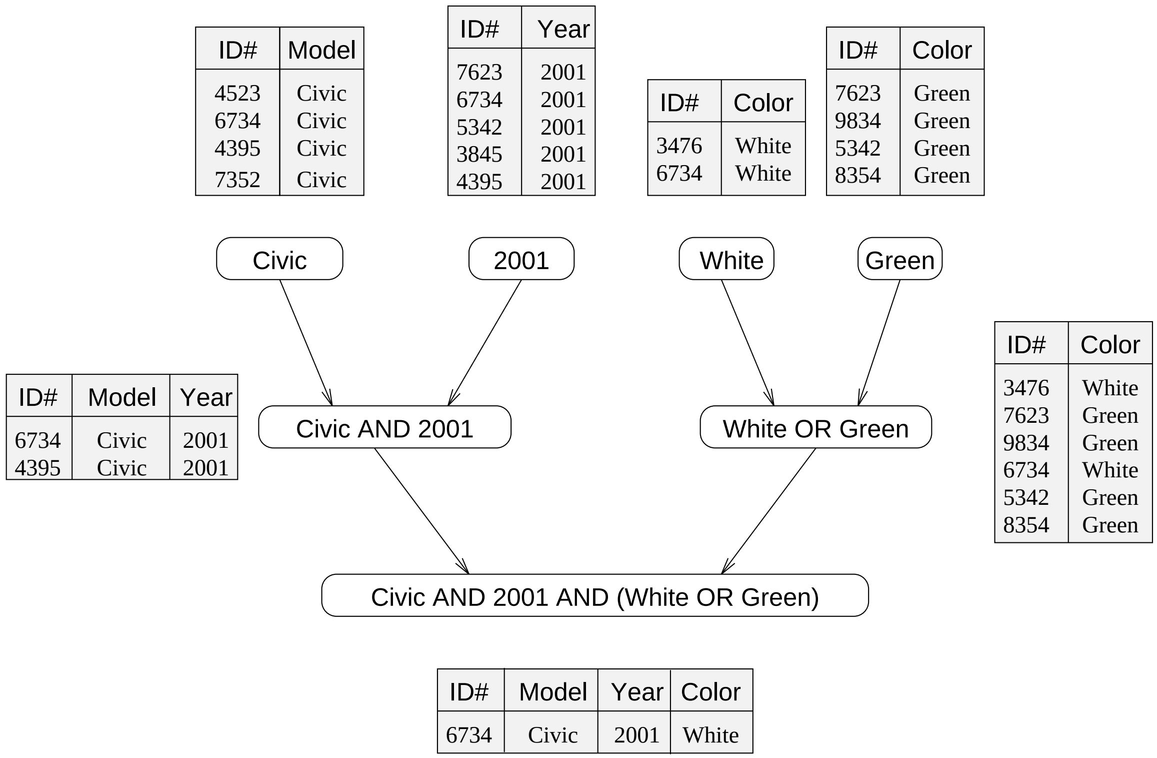
```
MODEL = ''CIVIC'' AND YEAR = 2001 AND
(COLOR = ''GREEN'' OR COLOR = ''WHITE)
```

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.



ID#	Model
4523	Civic
6734	Civic
4395	Civic
7352	Civic

ID#	Year
7623	2001
6734	2001
5342	2001
3845	2001
4395	2001

ID#	Color
3476	White
6734	White

ID#	Color
7623	Green
9834	Green
5342	Green
8354	Green

ID#	Model	Year
6734	Civic	2001
4395	Civic	2001

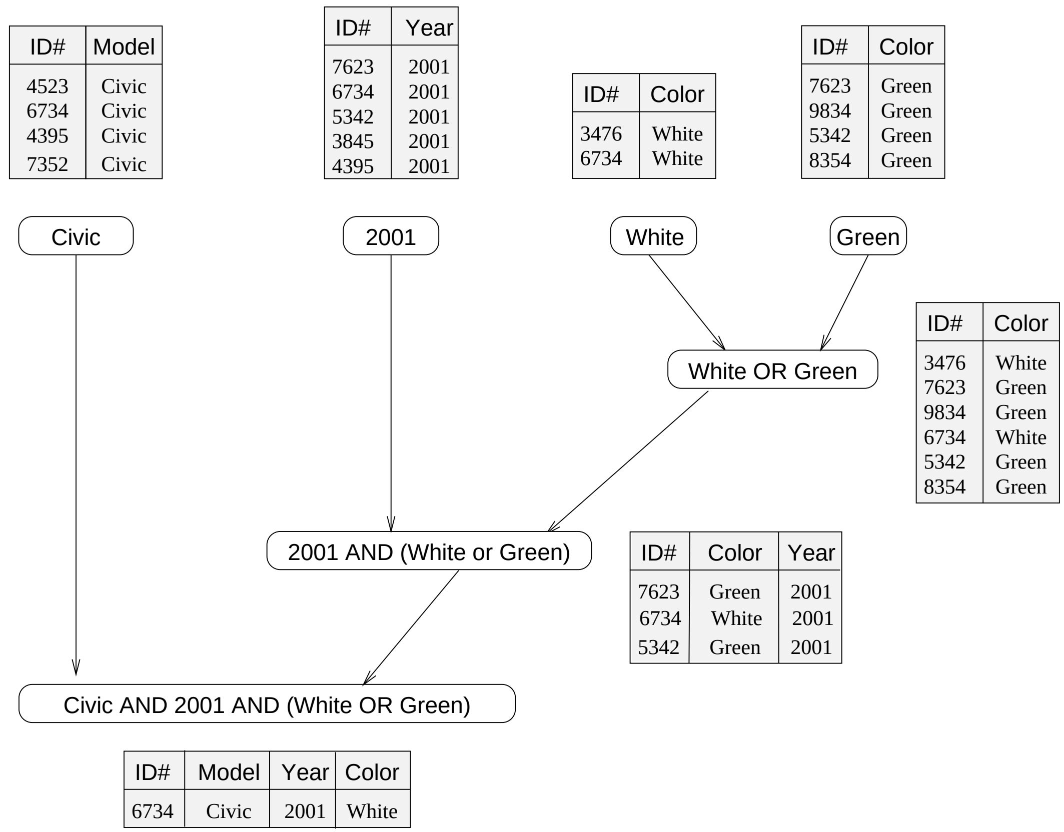
ID#	Color
3476	White
7623	Green
9834	Green
6734	White
5342	Green
8354	Green

ID#	Model	Year	Color
6734	Civic	2001	White

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.



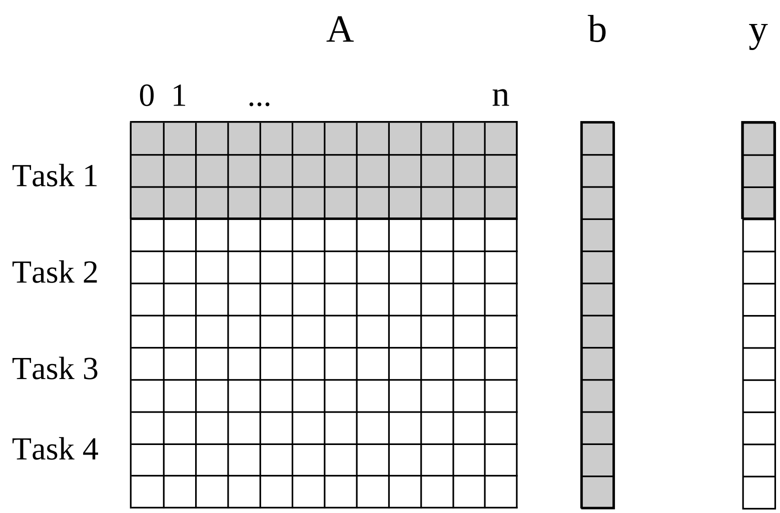
An alternate decomposition of the given problem into subtasks, along with their data dependencies.

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.

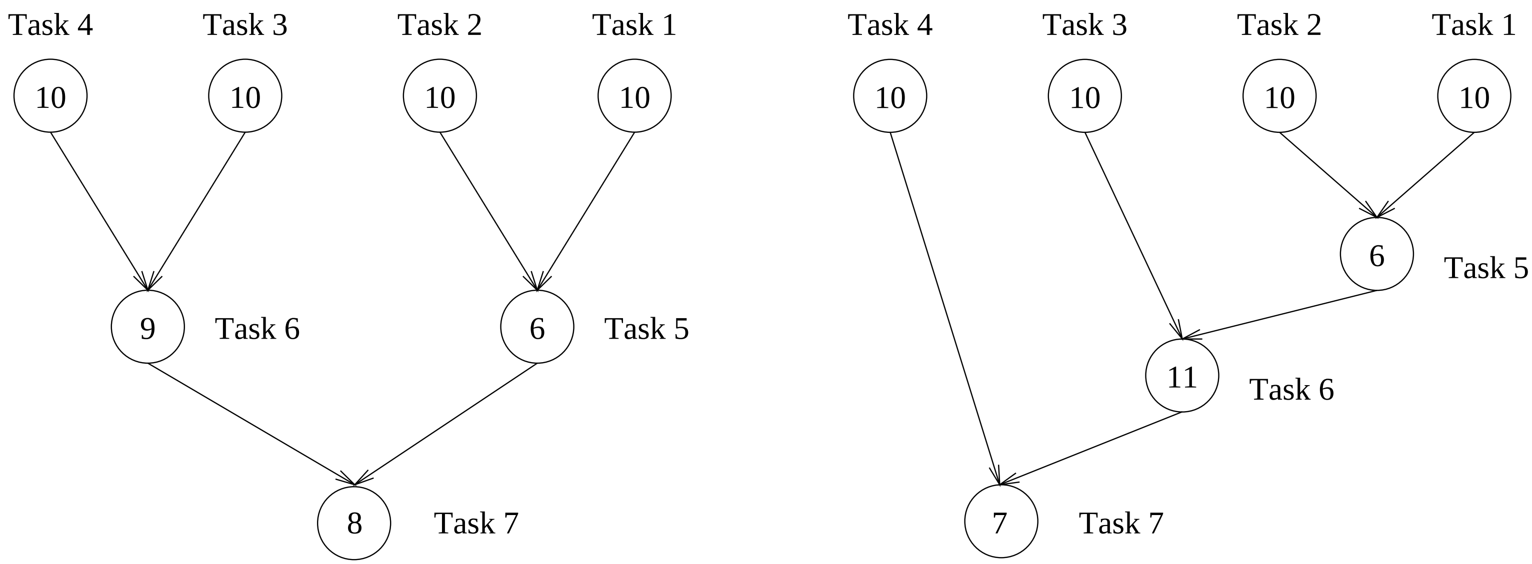
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.

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 time needed by a parallel execution.
- 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:



(a) (b)

What are the critical path lengths for the two task dependency graphs?

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* (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

- Tasks generally exchange data with others in a decomposition. For example, even in the trivial decomposition of the dense matrix-vector product, if the vector b is not replicated across all tasks, they will have to communicate elements of b.

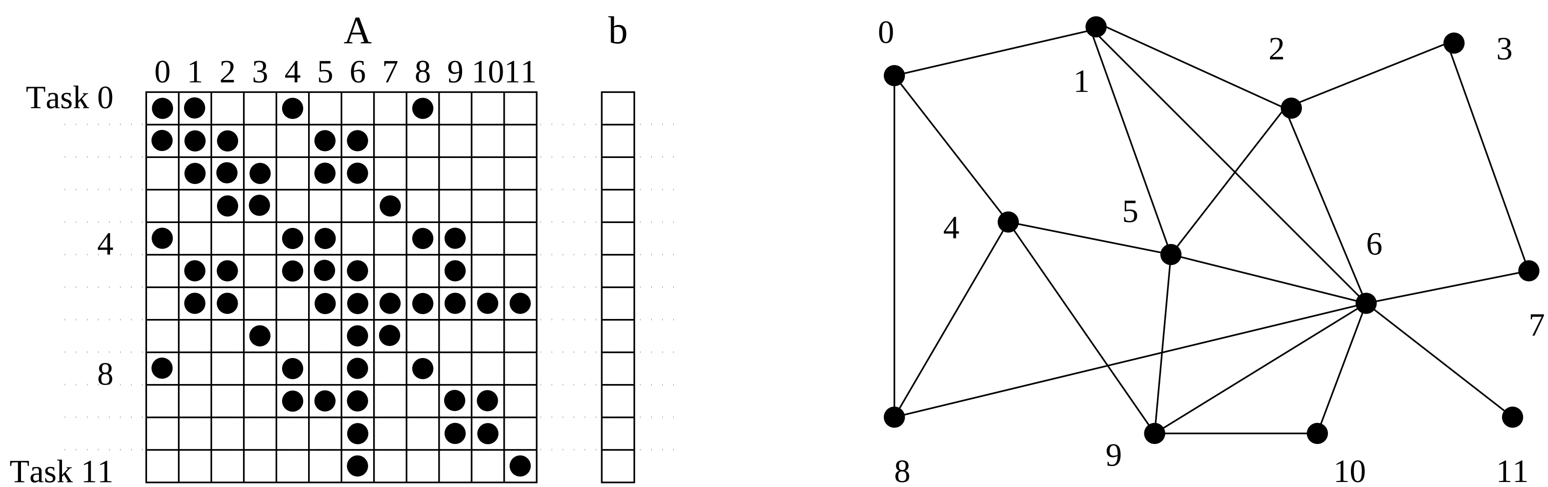
- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a **task interaction graph**.

- 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 assocaited 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.

Viewing node 0 as an independent task involves a useful computation of one time unit and overhead (communication) of three time units.

Now, if we consider nodes 0, 4, and 5 as one task, then the task has useful computation totaling to three time units and communication corresponding to four time units (four edges).

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 must also provide a mapping of tasks to processes.

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).

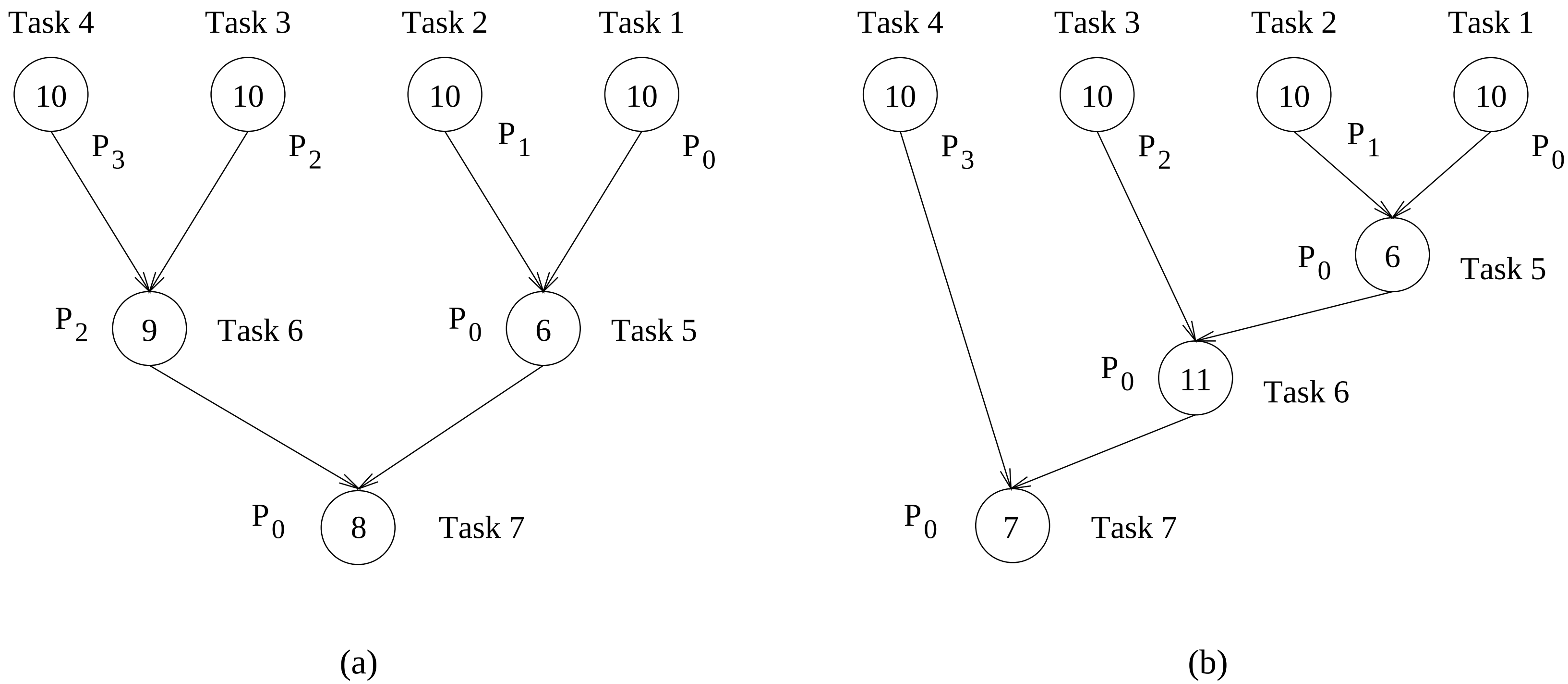
Processes and Mapping

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.

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!

Processes and Mapping: Example



Mapping tasks in the database query decomposition to processes. These mappings were arrived at by viewing the dependency graph in terms of levels (no two nodes in a level have dependencies). Tasks within a single level are then assigned to different processes.

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:

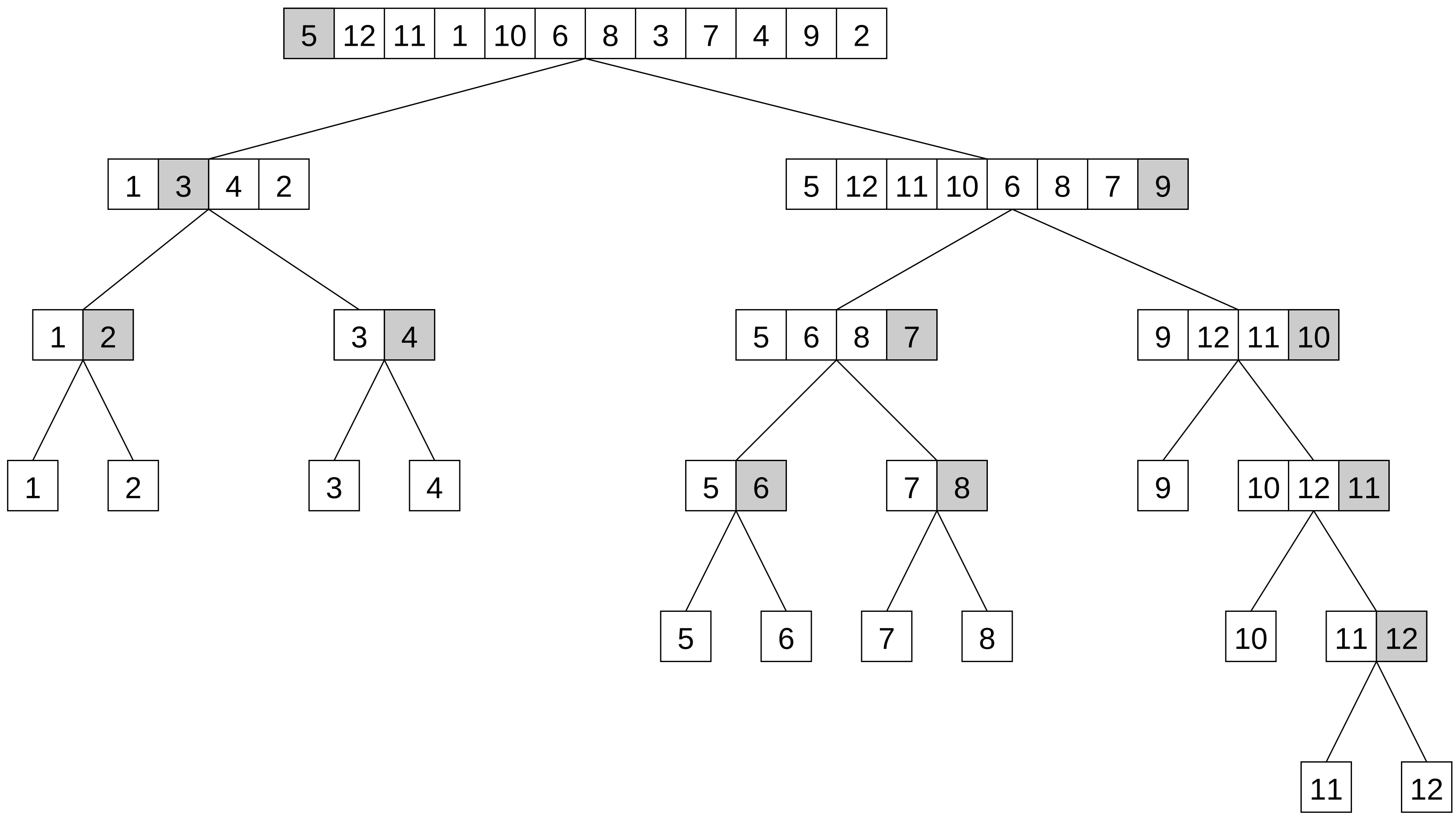
- recursive decomposition
- data decomposition
- exploratory decomposition
- speculative decomposition

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:

```
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
```

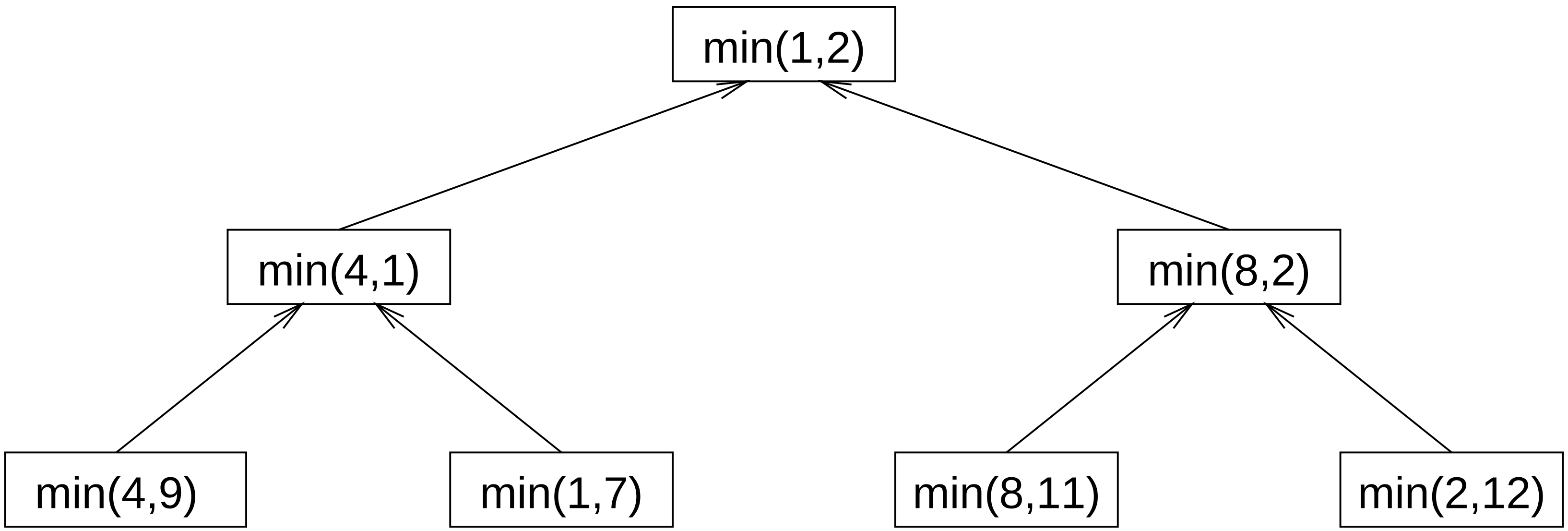
Recursive Decomposition: Example

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 – 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.

Output Data Decomposition: Example

Consider the problem of multiplying two $n \times n$ 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
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
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

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

(b) Partitioning the frequencies (and itemsets) among the tasks

task 1

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
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

Itemsets	Itemset Frequency
A, B, C	1
D, E	3
C, F, G	0
A, E	2

task 2

Database Transactions
A, B, C, E, G, H
B, D, E, F, K, L
A, B, F, H, L
D, E, F, H
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

Itemsets	Itemset Frequency
C, D	1
D, K	2
B, C, F	0
C, D, K	0

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.

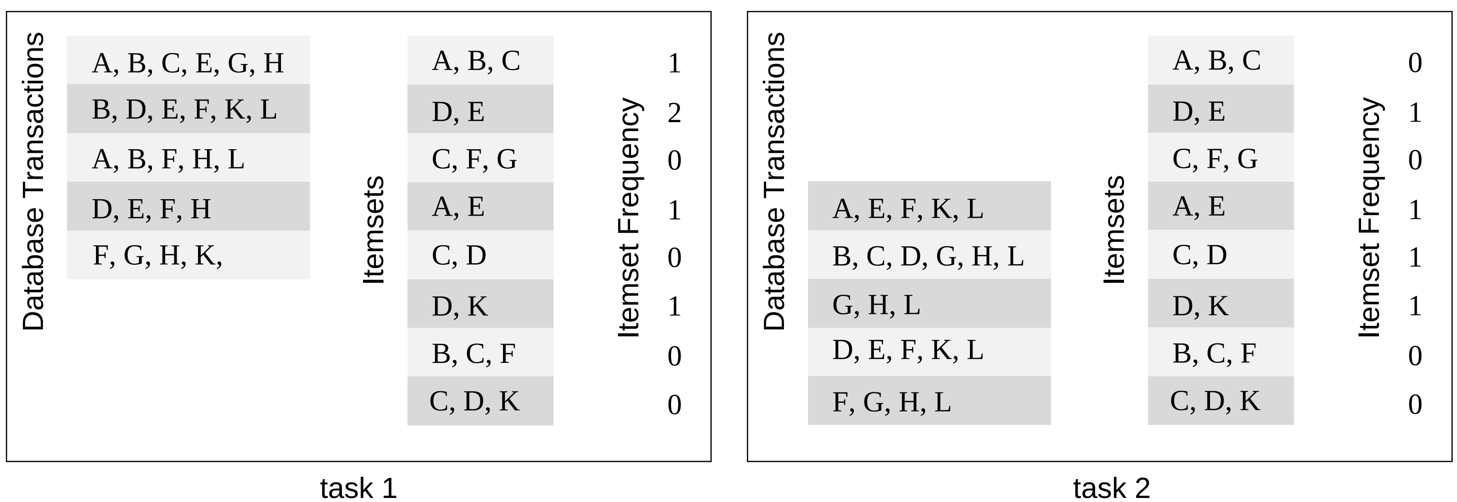
- In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).

- 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.

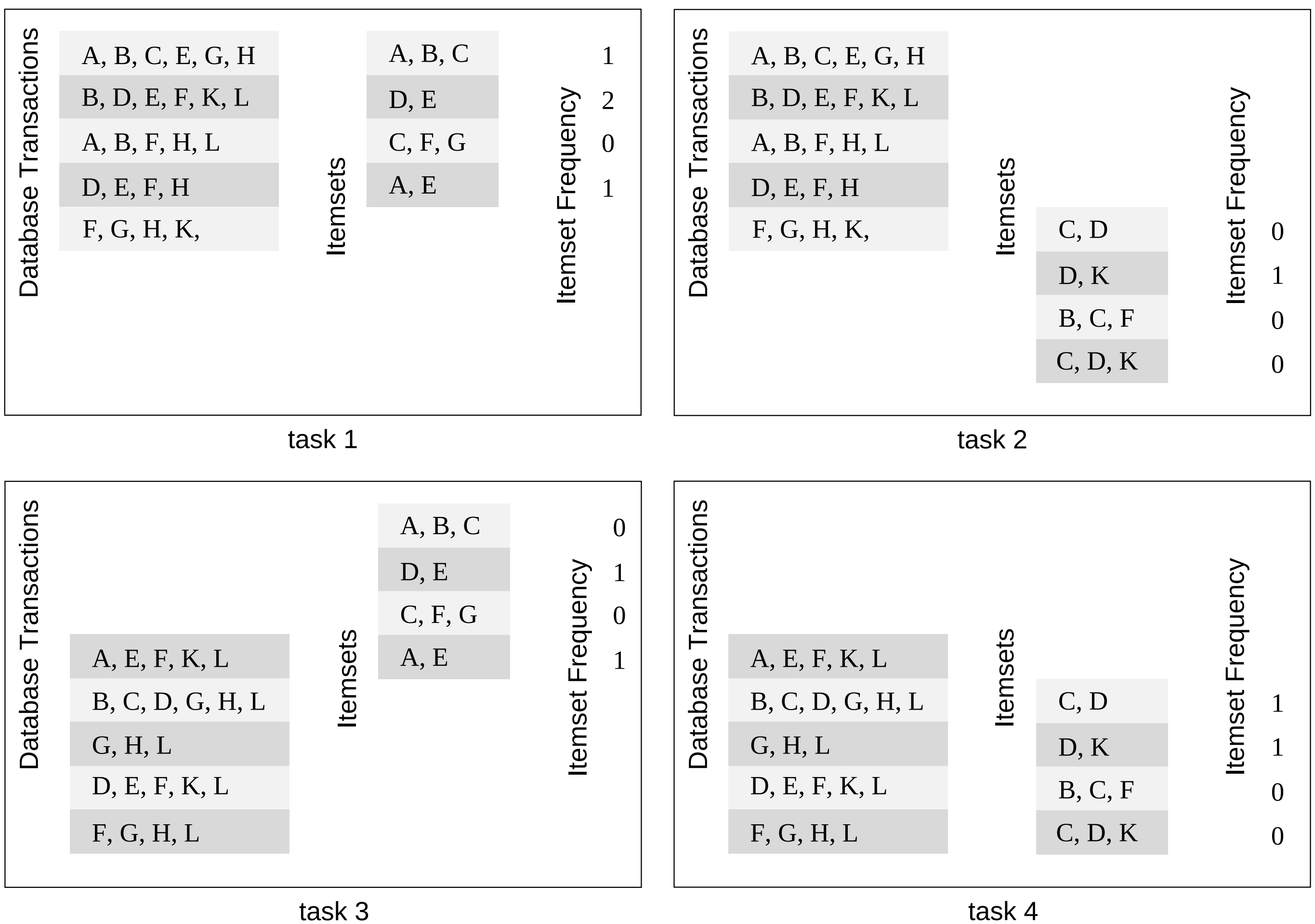
Partitioning the transactions among the tasks



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:

Partitioning both transactions and frequencies among the tasks



The Owner Computes Rule

- The *Owner Computes Rule* generally states that the process assined a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule imples 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.

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.

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 (e.g., 0/1 integer programming).

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, especitally, as we shall see, 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

- 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 modily 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

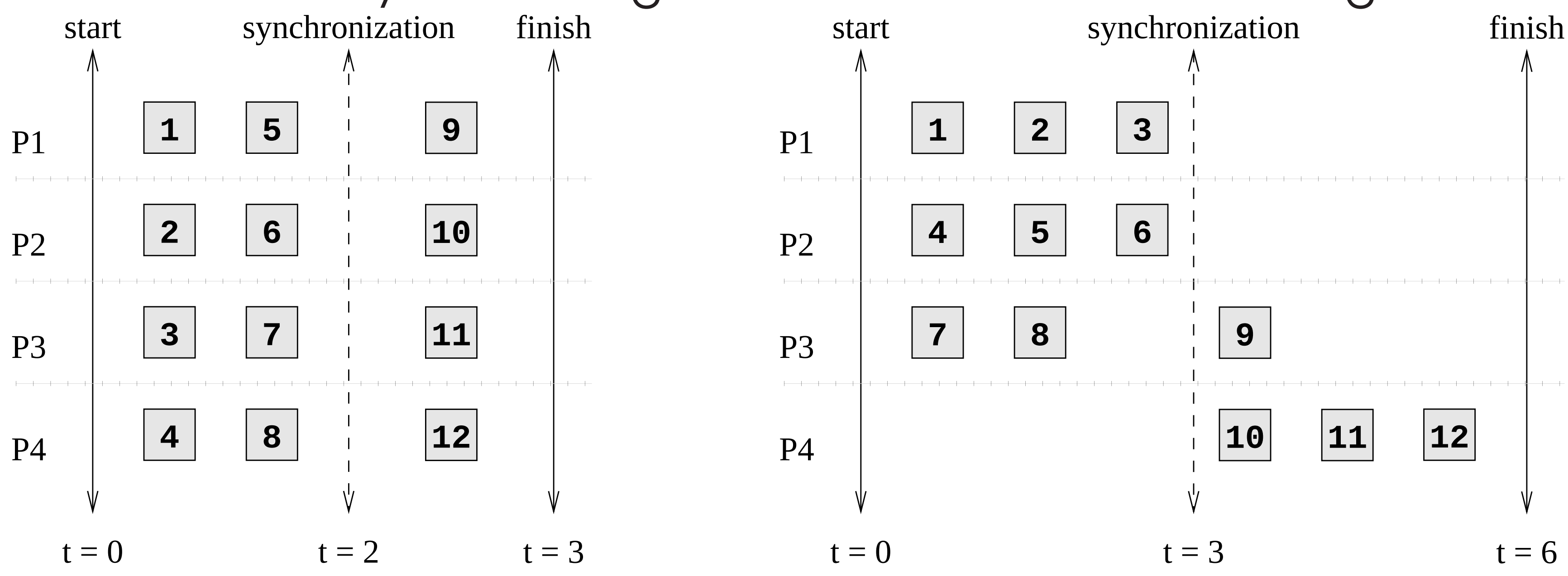
- Interactions may be one-way or two-way.
- A one-way interaction can be initiated and accomplished by one of the two interacting tasks.
- A two-way interaction requires participation from both tasks involved in an interaction.

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 for Minimum Idling

Mapping must simultaneously minimize idling and load balance. Merely balancing load does not minimize idling.



(a) (b)

Mapping Techniques

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.

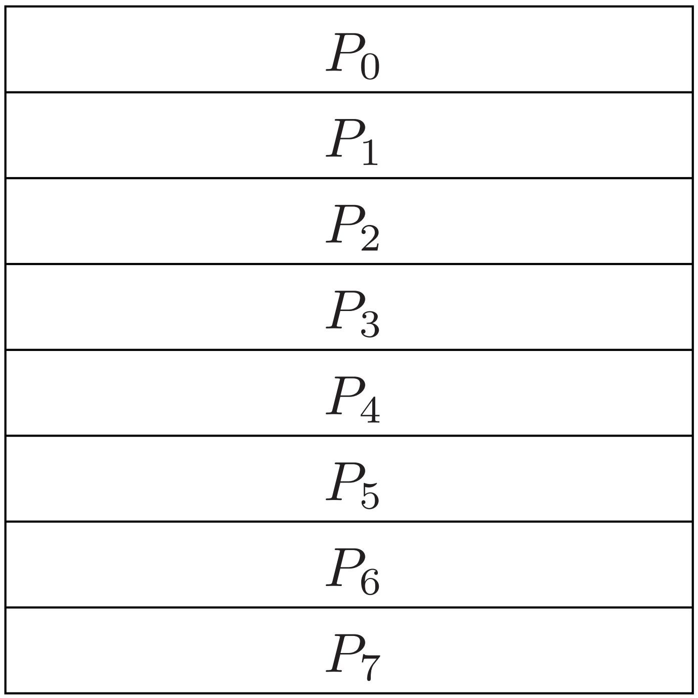
Schemes for Static Mapping

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

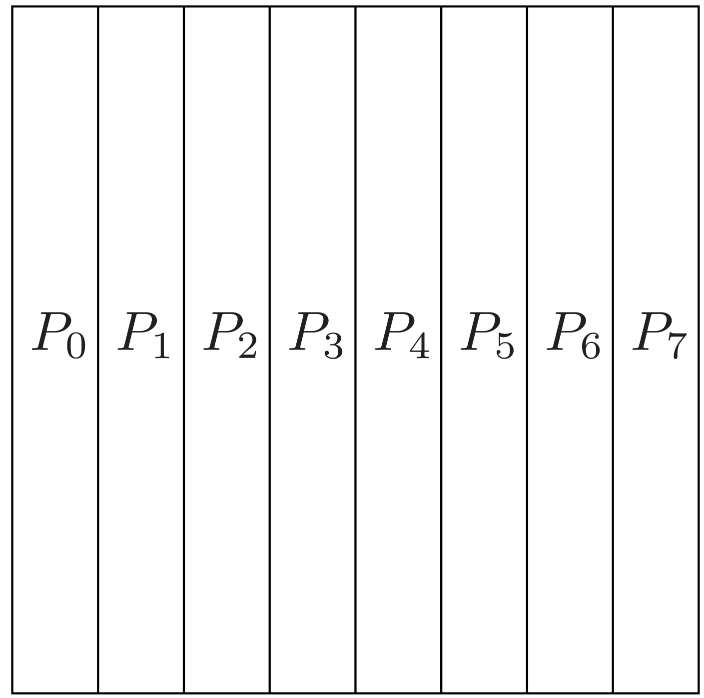
Mappings Based on Data Partitioning

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.

row-wise distribution



column-wise distribution



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

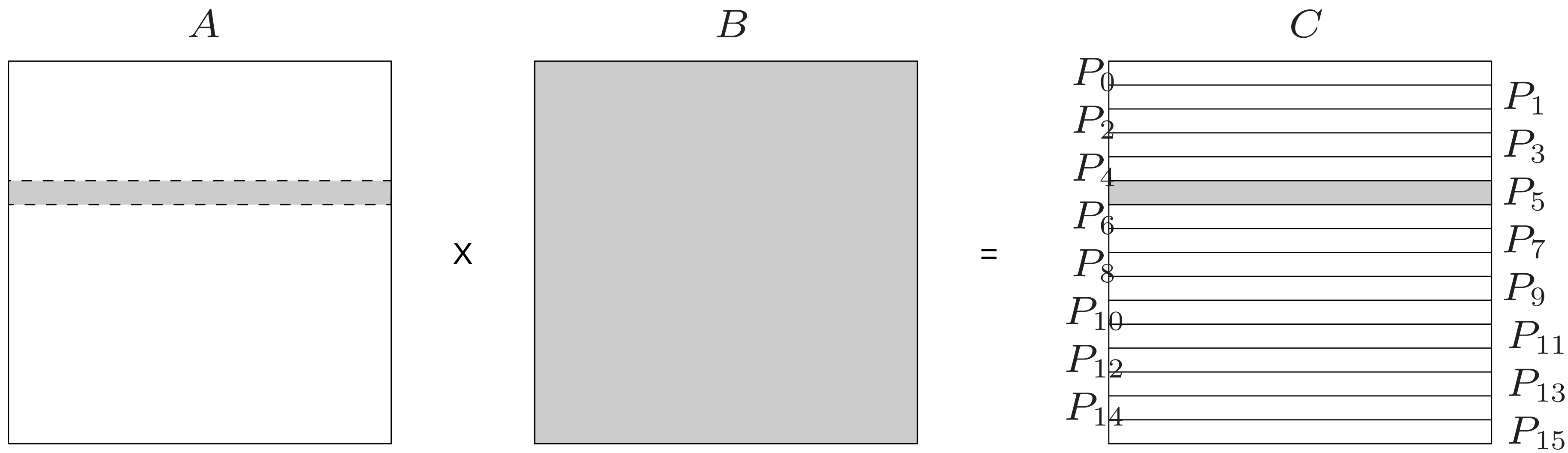
- For multiplying two dense matrices A and B, we can partition the output matrix C using a block decomposition.

- 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.)

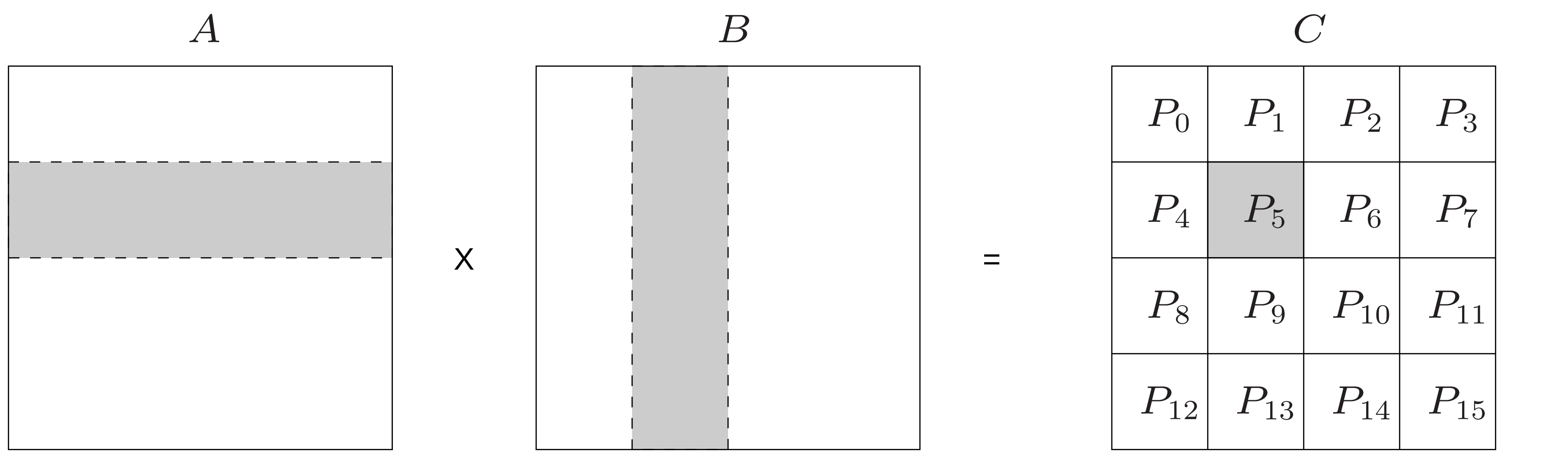
- The choice of precise decomposition (1-D or 2-D) is determined by the associated communication overhead.

- In general, higher dimension decomposition allows the use of larger number of processes.

Data Sharing in Dense Matrix Multiplication



(a)



(b)

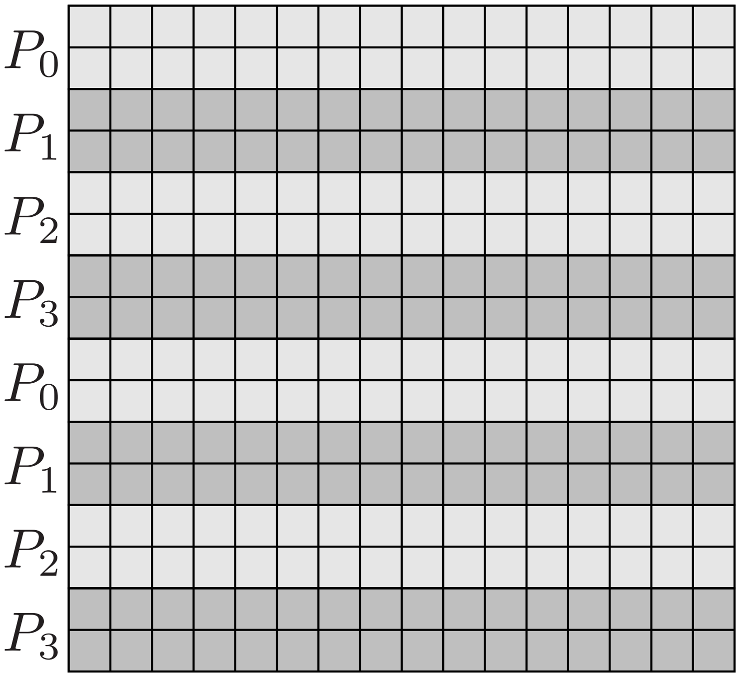
Cyclic and Block Cyclic Distributions

- If the amount of computation associated with data items varies, a block decomposition may lead to significant load imbalances.

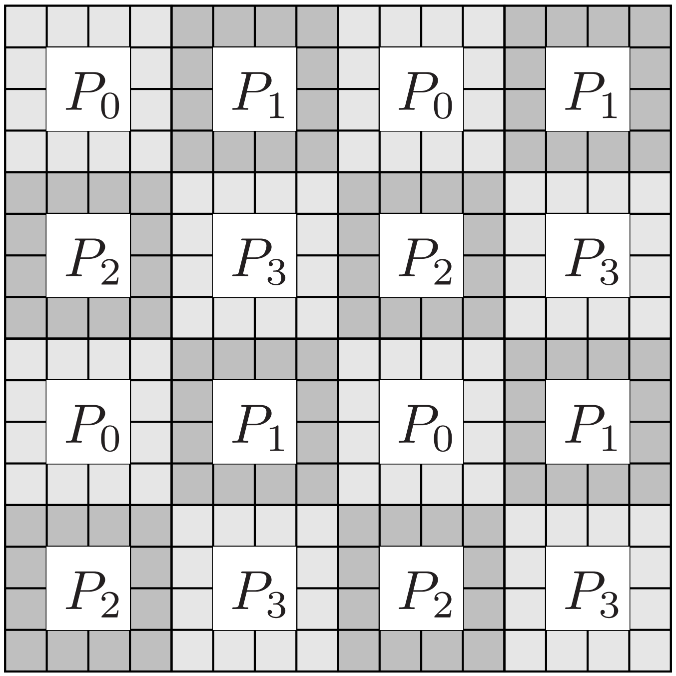
- A simple example of this is in LU decomposition (or Gaussian Elimination) of dense matrices.

Block-Cyclic Distribution: Examples

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



(a)

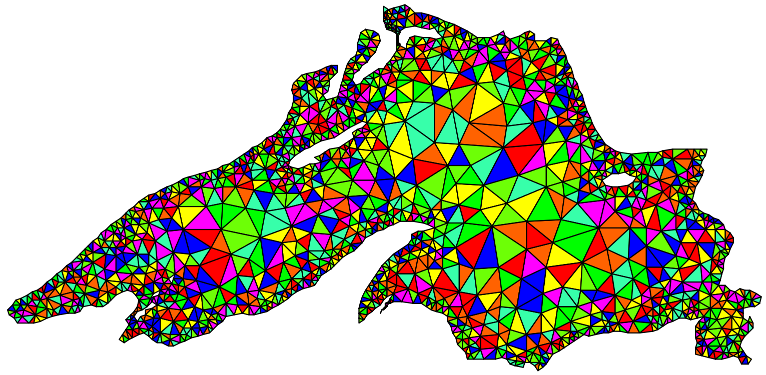


(b)

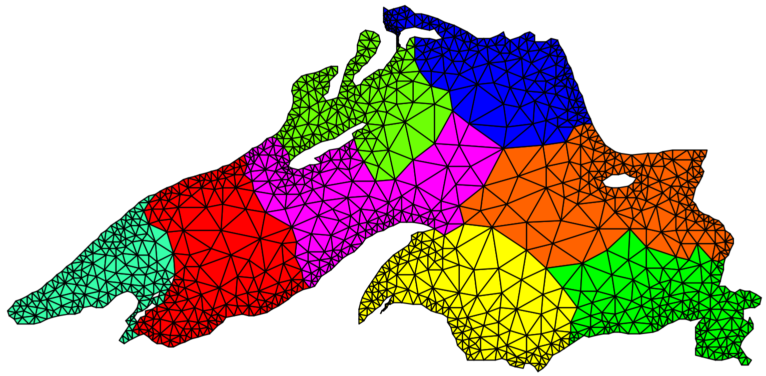
Graph Partitioning Based Data Decomposition

- 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



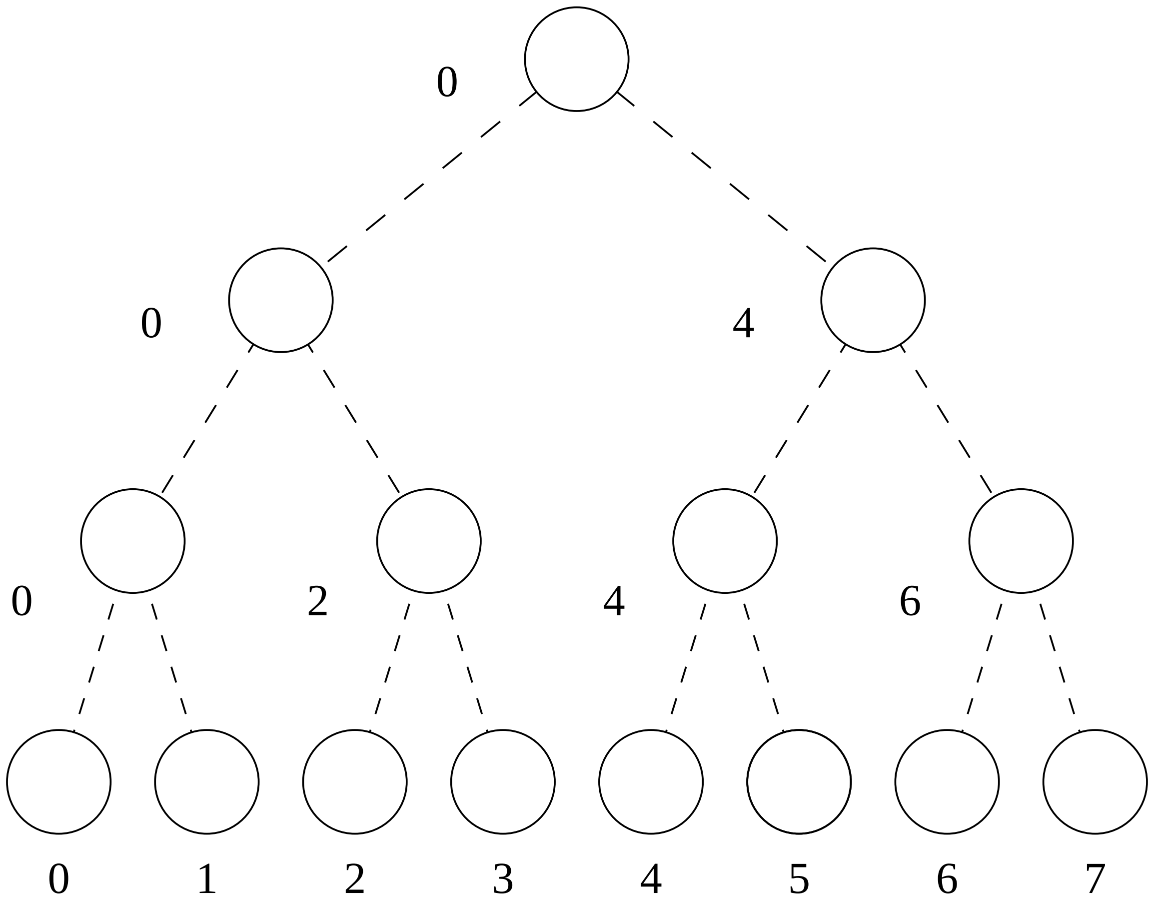
Partitioning for minimum edge-cut.

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.

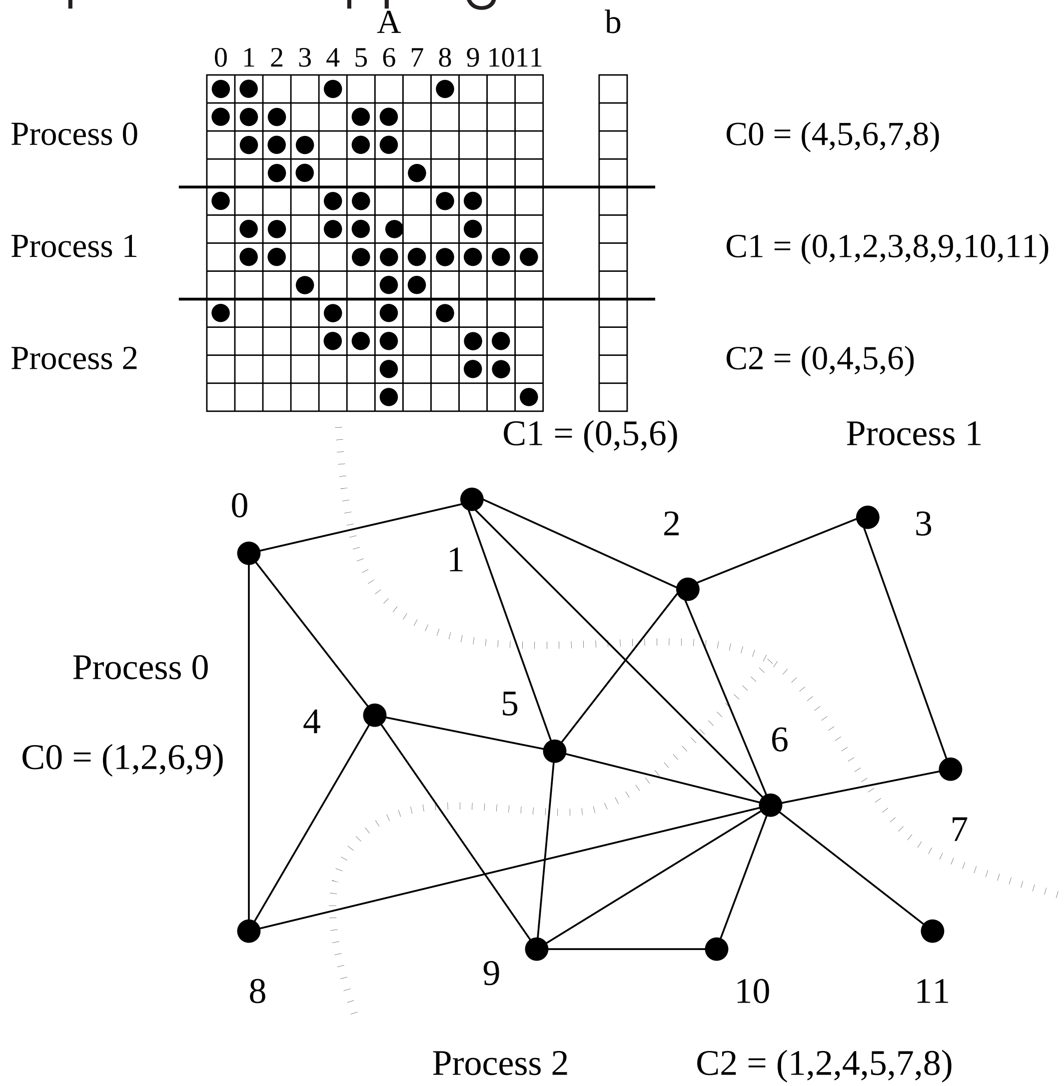
Task Paritioning: Mapping a Binary Tree Dependency Graph

Example: Assigning the tasks of a bindary-tree dependency graph to 8 processes in a hypercube.



Task Paritioning: Mapping a Sparse Graph

Sparse graph for computing a sparse matrix-vector product and two examples of 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.

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.