



# Parallel and Distributed Computing

Dr. Ali Sayyed  
Department of Computer Science  
National University of Computer & Emerging Sciences



# Storage and the Memory Hierarchy





# Performance–Memory Relation



- Although designing and implementing **efficient algorithms** is **typically the most critical aspect of writing programs that perform well**, there's another, often overlooked factor that can have a major impact on performance: **memory**.
- Perhaps surprisingly, **two algorithms** with the same performance run on the same inputs **might perform very differently** in practice due to the organization of the hardware on which they execute.
- To achieve the best performance, **a program's access patterns need to align well with the hardware's memory arrangement**

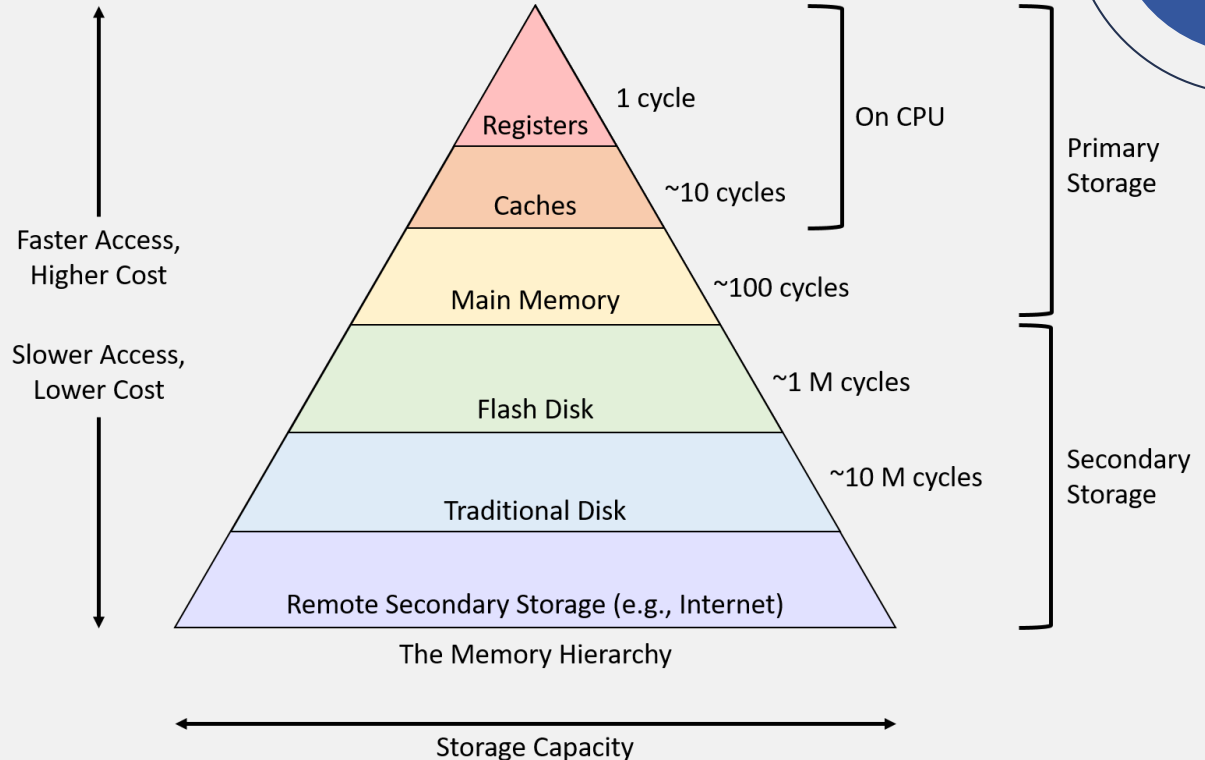
# Comparison of two codes

```
float averageMat_v1(int **mat, int n) {  
    int i, j, total = 0;  
  
    for (i = 0; i < n; i++) {  
        for (j = 0; j < n; j++) {  
            total += mat[i][j];  
        }  
    }  
  
    return (float) total / (n * n);  
}
```

```
float averageMat_v2(int **mat, int n) {  
    int i, j, total = 0;  
  
    for (i = 0; i < n; i++) {  
        for (j = 0; j < n; j++) {  
            total += mat[j][i];  
        }  
    }  
  
    return (float) total / (n * n);  
}
```

# The Memory Hierarchy

Here the cache is shown as single entity, but most systems contain **multiple levels of caches** that form their own smaller hierarchy.





# Storage Devices



- **Primary storage** devices can be **accessed directly by a program on the CPU**.
  - That is, the CPU's assembly instructions encode the exact location of the data that the instructions should retrieve.
- In contrast, CPU instructions cannot directly refer to **secondary storage devices**.
  - To access the contents of a secondary storage device, **a program must first request that the device copy its data into primary storage**



# Characteristics of Memory Hierarchy



- **Capacity:** The amount of data a device can store.
  - Measured in bytes.
- **Latency:** **The amount of time it takes for a device to respond** with data after it has been instructed to perform a data retrieval operation.
  - Measured in milliseconds or nanoseconds or CPU cycles.
- **Transfer rate:** The amount of data that can be moved between the device and main memory per unit time.
  - Measured in bytes per second.



# Common Primary Storage Devices



Device	Capacity	Approx. latency
Register	4 - 8 bytes	< 1 ns
CPU cache	1 - 32 megabytes	5 ns
Main memory	4 - 64 gigabytes	100 ns





# Common Secondary Storage Devices



Device	Capacity	Latency
Flash disk	0.5 - 2 terabytes	0.1 - 1 ms
Traditional hard disk	0.5 - 10 terabytes	5 - 10 ms
Remote net- work server	Varies considerably	20 - 200 ms



# Locality



- Because memory devices vary considerably in their performance, modern systems integrate several forms of storage.
- Luckily, most programs exhibit common memory **access patterns, known as locality**, and designers build hardware that exploits good locality to automatically move data into an appropriate storage location.
- **A system improves performance by moving the subset of data** that a program is actively using into storage that lives close to the CPU (register or CPU cache).
- As necessary **data moves up the hierarchy toward the CPU, unused data moves farther away** to slower storage.



# Forms of Locality



- Systems primarily exploit two forms of locality:
- **Temporal locality:** Programs tend to **access the same data** repeatedly over time.
  - If a particular memory location (or data) is accessed, it is likely to be accessed again in the near future.
- **Spatial locality:** Programs tend to **access data that is nearby** other, previously accessed data. "Nearby" here refers to the data's memory address.
  - For example, if a program accesses data at addresses  $N$  and  $N+1$ , it's likely to access  $N+2$  soon.



# Locality Examples



- Fortunately, common programming patterns exhibit both forms of locality quite frequently. Take the following function, for example:

```
/* Sum up the elements in an integer array of length len. */
int sum_array(int *array, int len) {
    int i;
    int sum = 0;

    for (i = 0; i < len; i++) {
        sum += array[i];
    }

    return sum;
}
```

The variable `sum` and `i` are accessed **repeatedly** in each iteration of the loop and has **temporal locality** because it is accessed repeatedly in a short period.

**What about Spatial locality?**

# Locality Examples

- In many cases, a programmer can help a system by **intentionally writing code that exhibits good locality patterns.**

```
float averageMat_v1(int **mat, int n) {  
    int i, j, total = 0;  
  
    for (i = 0; i < n; i++) {  
        for (j = 0; j < n; j++) {  
            // Note indexing: [i][j]  
            total += mat[i][j];  
        }  
    }  
  
    return (float) total / (n * n);  
}
```

```
float averageMat_v2(int **mat, int n) {  
    int i, j, total = 0;  
  
    for (i = 0; i < n; i++) {  
        for (j = 0; j < n; j++) {  
            // Note indexing: [j][i]  
            total += mat[j][i];  
        }  
    }  
  
    return (float) total / (n * n);  
}
```

- Due to the **row-major order organization of a matrix in memory**, the code (left) **executes about 5 times faster** than the code (right). The first version accesses the matrix's values **sequentially in memory**. The second version accesses the matrix's values by **repeatedly jumping between rows** across nonsequential memory addresses.



# Locality Examples

```
float averageMat_v1(int **mat, int n) {
    int i, j, total = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [i][j]
            total += mat[i][j];
        }
    }

    return (float) total / (n * n);
}
```

```
float averageMat_v2(int **mat, int n) {
    int i, j, total = 0;

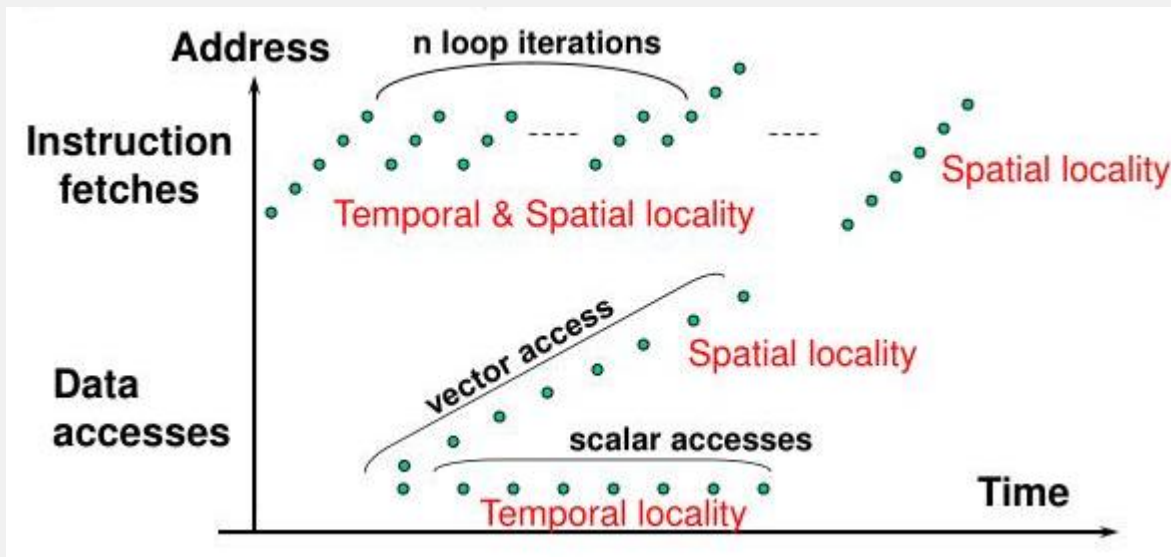
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [j][i]
            total += mat[j][i];
        }
    }

    return (float) total / (n * n);
}
```

- In terms of cache efficiency, v1 (left) might perform better in most cases because it exhibits better spatial locality. Since it accesses elements in a contiguous manner within each row, it tends to utilize cache more effectively, resulting in potentially faster execution compared to v2 (right), which accesses elements column-wise and might cause more cache misses.



# Typical Memory Reference Patterns





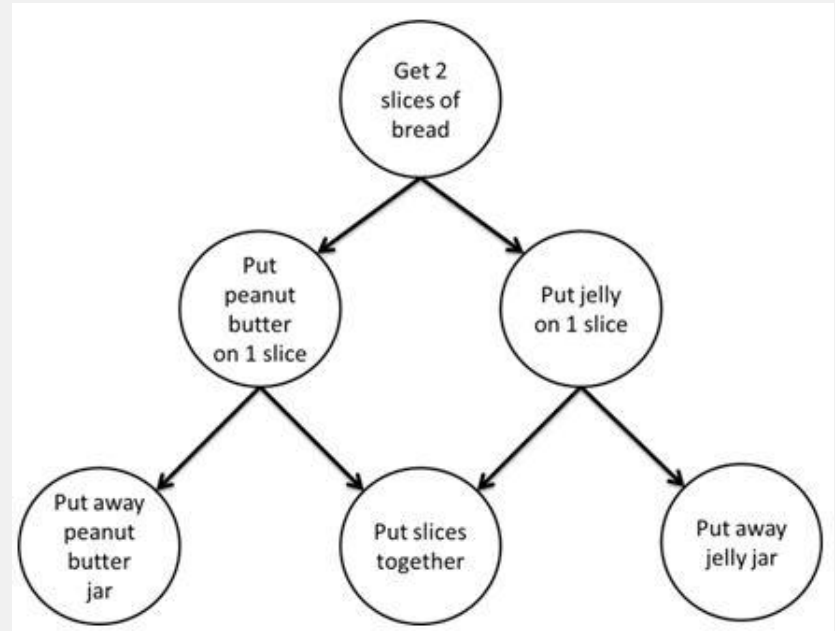
# Dependency Graphs





# Dependency Graphs

- 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: Database Query Processing

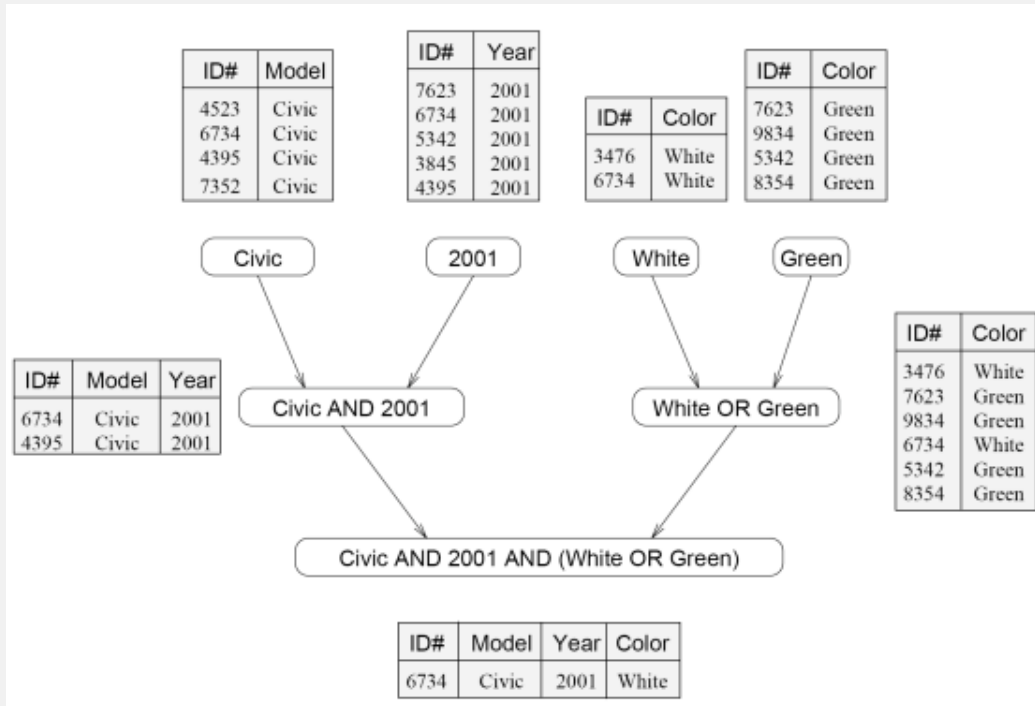
Consider the execution of the query: **MODEL = "CIVIC" AND YEAR = 2001 AND (COLOR = "GREEN" OR COLOR = "WHITE")** on the following database:

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

**Table 3.1** A database storing information about used vehicles.

# Example: Database Query Processing

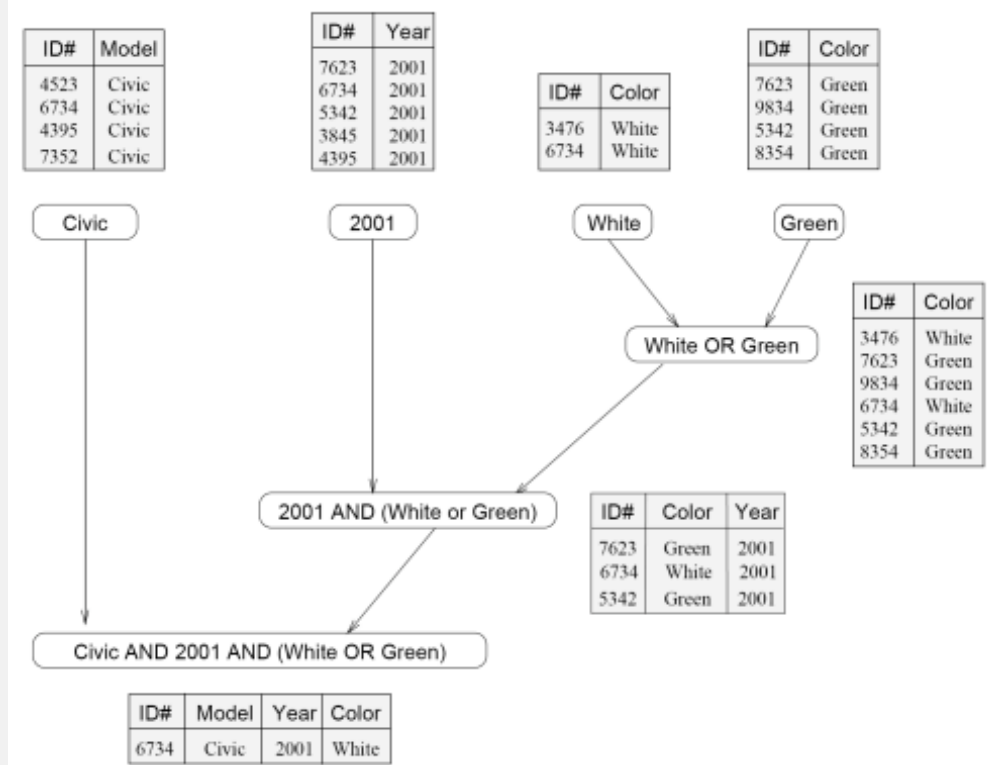
- The execution of the query can be **divided into subtasks in various ways**.
- **Each task** can be thought of as **generating an intermediate table** of entries that satisfy a particular clause.



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

# Example: Database Query Processing

- An **Alternate Task Decomposition** and Dependency Graph



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



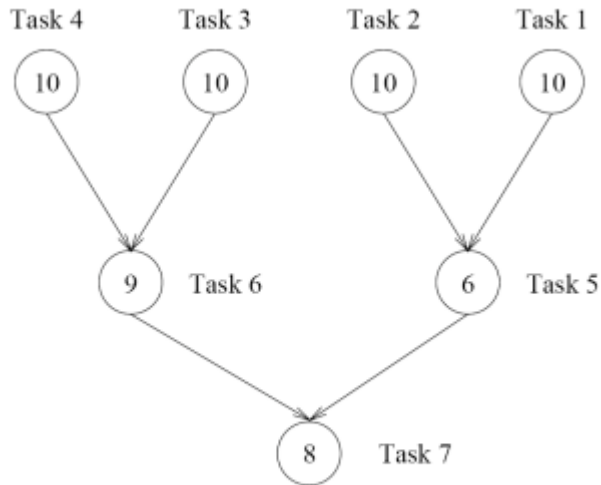
# Critical Path Length



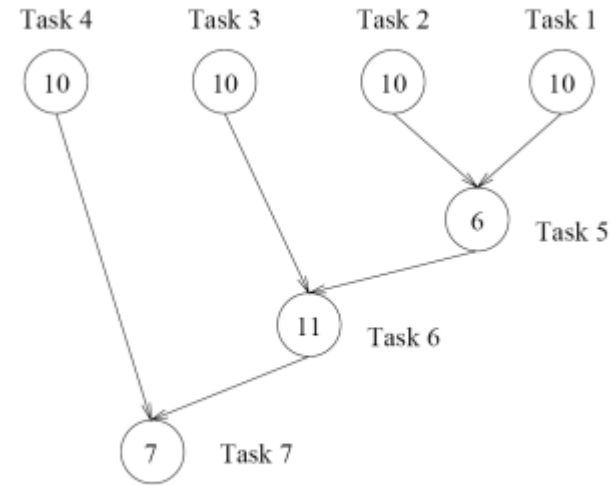
- **Critical path:** The longest directed path between any pair of start node (node with no incoming edge) and finish node (node with no outgoing edges).
- **Critical path length:** The sum of weights of nodes along critical path.
  - The weights of a node is the size or the amount of work associated with the corresponding task
- **Average degree of concurrency** = total amount of work / critical path length

# Examples of Critical Path Length

- Consider the task dependency graphs of the two database query decompositions



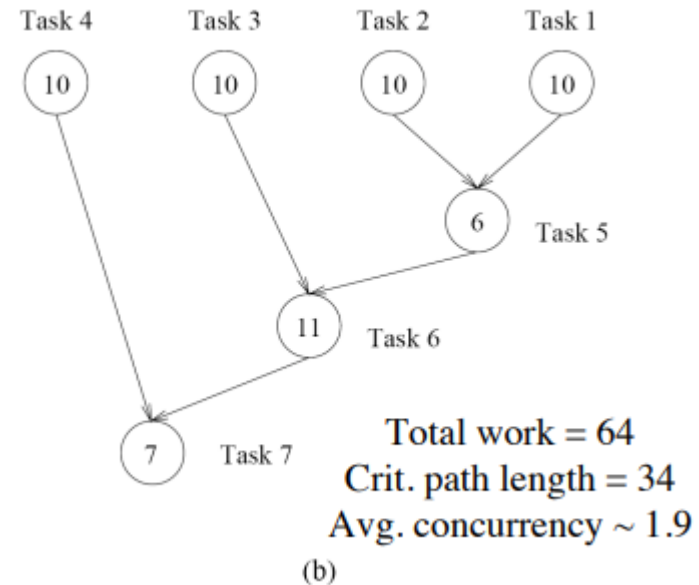
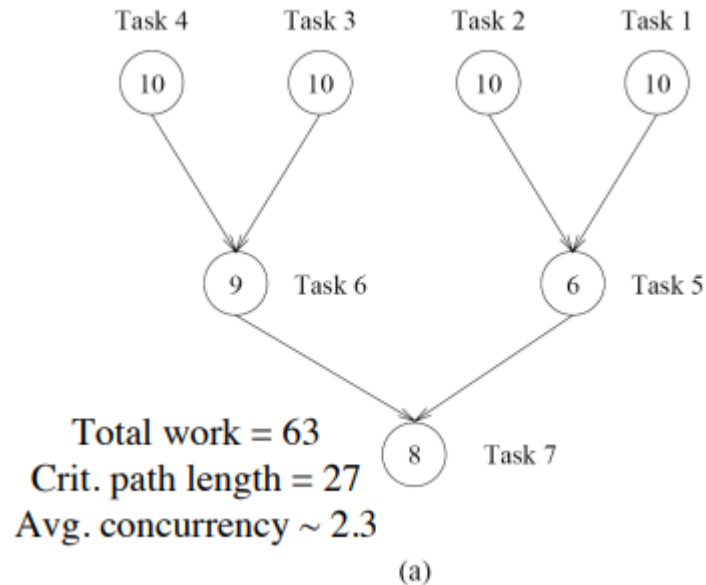
(a)



(b)

# Examples of Critical Path Length

- Consider the task dependency graphs of the two database query decompositions





# Decomposition Techniques








# 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



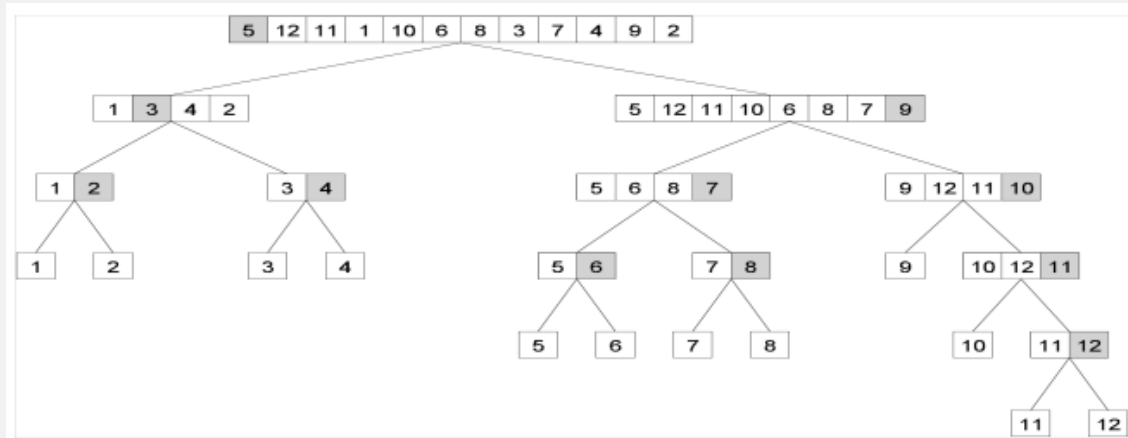
Ideal for problems to be solved by divide-and conquer method.

## Steps

- Decompose a problem into a set of independent sub-problems
  - Recursively decompose each sub-problem
  - Stop decomposition when minimum desired granularity is achieved or (partial) result is obtained
- 

# Recursive Decomposition: Example


- A classic example is to apply recursive decomposition to **Quicksort**.
- It works basically on the basis of the “Divide and Conquer” principle.
- In this example, a task represents the work of partitioning a (sub)array. Note that each subarray represents an independent subtask. This can be repeated recursively.





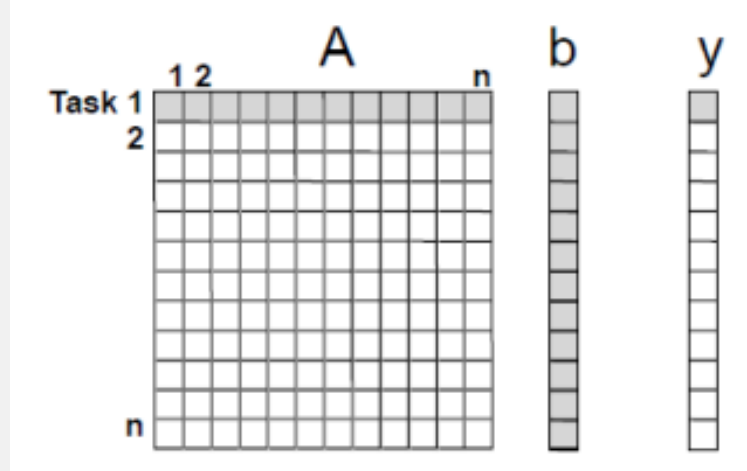
# Data Decomposition



- Identify the data on which computations are performed.
  - Partition data into sub-unit
  - Data can be input, output or intermediate for different computations
  - Data partition is used to induce a partitioning of the computations into tasks
- 

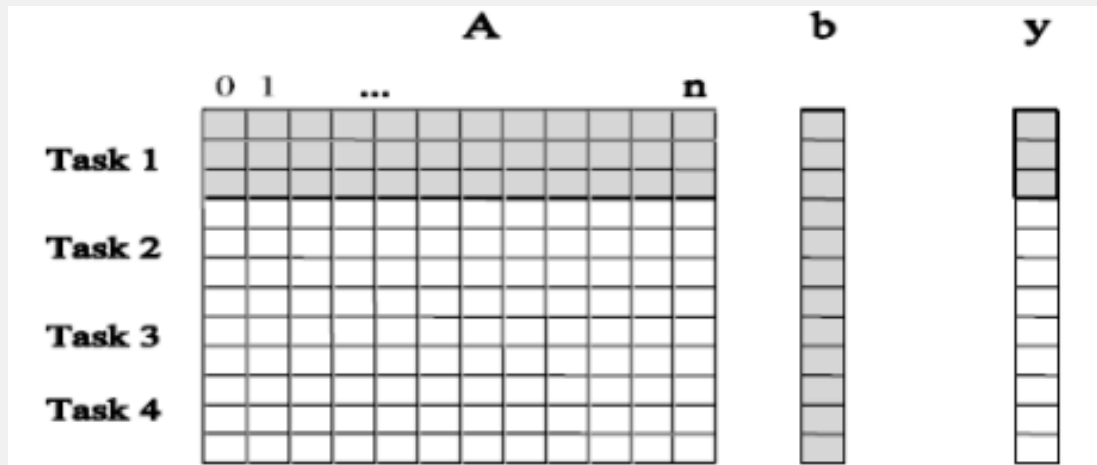
# Data Decomposition: Example

- If each element of the output can be computed independently of others as a function of the input.
- Example. matrix-vector multiplication.



# Granularity of Task Decomposition

- **Fine-grained decomposition:** large number of small tasks
- **Coarse-grained decomposition:** small number of large tasks
- In Matrix-vector multiplication example, a coarse-grain decomposition



# 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 as follows

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

(a)

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

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

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

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

(b)

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

Task 4:  $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$

Task 5:  $C_{2,1} = A_{2,1}B_{1,1}$

Task 6:  $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$

Task 7:  $C_{2,2} = A_{2,1}B_{1,2}$

Task 8:  $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$



# 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). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12
(a)			

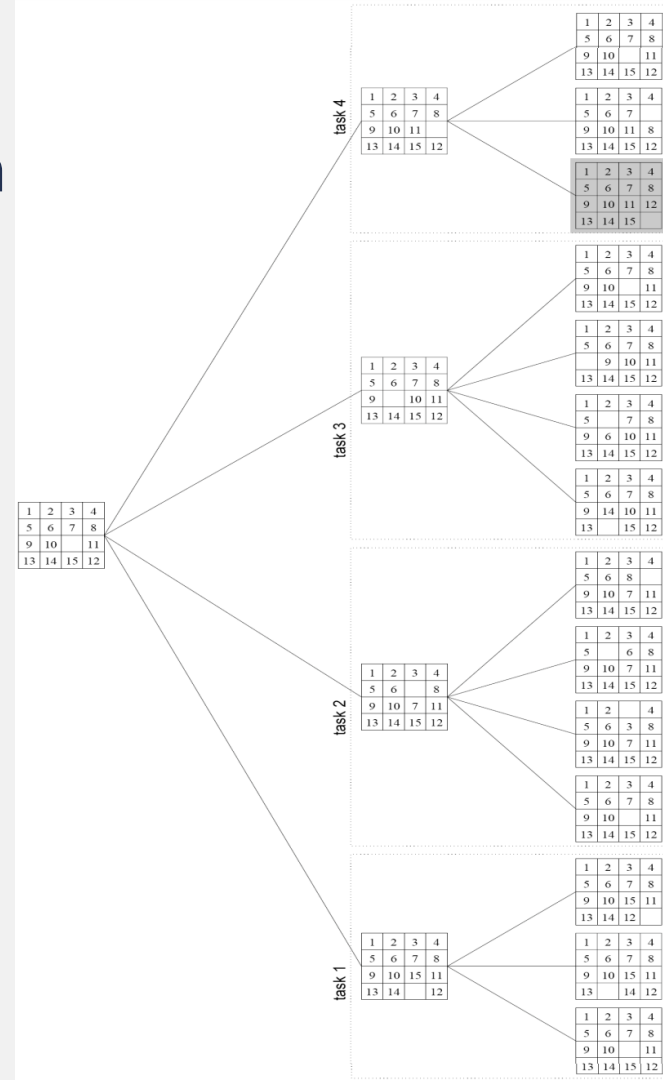
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12
(b)			

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12
(c)			

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	
(d)			

# Exploratory Decomposition


- The state space can be explored by generating various successor states of the current state and viewing them as independent tasks





# Speculative Decomposition



- In some cases, **dependencies between tasks are not known a-priori**.
  - There are 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



- A system to handle **real-time stock market trading decisions**. The system needs to analyze various market indicators (e.g., price trends, trading volumes) to make buy or sell decisions. However, some indicators are dependent on others, and the relationship between them may not be clear in advance.
  - **Conservative Approach**: You analyze the indicators sequentially, waiting until each is fully processed, and all dependencies are resolved. This ensures correctness but limits concurrency.
  - **Optimistic Approach**: You schedule the analysis of market sentiment and price trends simultaneously, assuming they are independent. However, if the system later finds out that the decision based on market sentiment should have waited for the updated price trend, it would need to roll back
- 