

Excellent  — here's a **clean, summarized, and well-structured version** of your topic “**3.2 Decomposition Techniques**” (including **Recursive Decomposition** and the **Quicksort example**) that you can use directly in your notes or slides:

---

## 3.2 Decomposition Techniques

### Overview

To design a parallel algorithm, one of the fundamental steps is to **split computations into tasks** that can be executed **concurrently**.

These tasks and their relationships are represented using a **task-dependency graph**.

There are several standard **decomposition techniques** used to achieve concurrency.

---

### Types of Decomposition Techniques

1. **Recursive Decomposition**
  2. **Data Decomposition**
  3. **Exploratory Decomposition**
  4. **Speculative Decomposition**
- 

### General Notes

- These techniques help identify **parallel portions** of a computation.
  - A single problem may use **one or a combination** of these techniques.
  - The chosen decomposition method **may not always guarantee** the best performance, but it provides a **good starting point** for designing parallel algorithms.
- 

### 3.2.1 Recursive Decomposition

#### Definition:

**Recursive Decomposition** is used for problems that follow the **divide-and-conquer** strategy.

In this approach:

1. The problem is **divided** into a set of **independent subproblems**.

- 
2. Each subproblem is **solved recursively**, often by applying the same division process.
  3. The **results are combined** to produce the final output.
- 

## Characteristics:

- Naturally exposes **concurrency**, as independent subproblems can be solved **in parallel**.
  - Recursion continues until the problem is small enough to be solved directly (base case).
  - Commonly used in algorithms like **Merge Sort**, **Quick Sort**, **Matrix Multiplication**, and **FFT**.
- 

## Example — Quicksort

### Algorithm Overview:

- Quicksort works by selecting a **pivot element**  $x$ .
- The array  $A$  of size  $n$  is **partitioned** into two subsequences:
  - $A_0$ : Elements **less than** pivot  $x$
  - $A_1$ : Elements **greater than or equal to** pivot  $x$

### Steps:

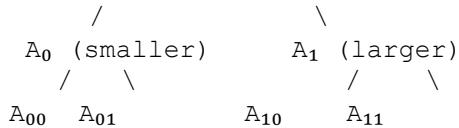
1. **Divide Step:**  
Partition the array around the pivot.
  2. **Conquer Step:**  
Recursively sort each subsequence ( $A_0$  and  $A_1$ ).
  3. **Combine Step:**  
Concatenate the sorted subsequences to form the final sorted list.
- 

## Parallelism in Quicksort:

- After partitioning, the two recursive calls on  $A_0$  and  $A_1$  are **independent** and can be executed **concurrently**.
  - The recursion continues until each subsequence has only **one element**, at which point sorting is complete.
- 

## Visualization (Conceptually):

$A$  (12 elements)



Each level of recursion can execute in **parallel**, providing **natural concurrency**.

---

Would you like me to continue with **3.2.2 Data Decomposition** next (the next major technique after Recursive Decomposition)?

Here's a **clear and summarized explanation** of the section you shared 

---

## 3.2 Decomposition Techniques

To achieve **parallelism**, a problem must be broken down into smaller **tasks** that can run concurrently.

This process is called **decomposition**.

There are **four main techniques** used to decompose problems into parallel tasks:

1. **Recursive Decomposition**
2. **Data Decomposition**
3. **Exploratory Decomposition**
4. **Speculative Decomposition**

*(The last two are special-purpose techniques, while the first two are general-purpose and widely used.)*

---

### 3.2.1 Recursive Decomposition

#### Concept:

Recursive decomposition is based on the **divide-and-conquer** approach.

The idea is to break a problem into smaller subproblems that can be solved **independently and in parallel**.

Each subproblem is again divided into smaller subproblems until it becomes simple enough to solve directly.

Once solved, the partial results are combined to form the final answer.

---

## Example: Quicksort

### Step 1 – Divide:

- Select a **pivot** element  $x$ .
- Split array  $A$  into:
  - $A_0$ : elements smaller than  $x$
  - $A_1$ : elements greater than or equal to  $x$

### Step 2 – Conquer:

- Sort  $A_0$  and  $A_1$  **recursively and in parallel**.

### Step 3 – Combine:

- Merge the sorted subsequences.

 The **task-dependency graph** for quicksort looks like a **binary tree**, where:

- The **root** is the initial partitioning task.
  - Each level represents tasks that can be executed **concurrently**.
  - As you go deeper, **concurrency increases**.
- 

## Restructuring for Recursion

Some problems are not naturally divide-and-conquer, but can still be restructured for recursive decomposition.

### Example: Finding Minimum Element

#### Serial version:

Scans all elements one by one (no concurrency).

```
for i = 1 to n-1:  
    if A[i] < min:  
        min = A[i]
```

#### Parallel version (recursive):

1. Split the array into two halves.
2. Find the minimum in each half **recursively and concurrently**.
3. Compare the two results to get the overall minimum.

This forms a **task-dependency tree** similar to a tournament — where each node compares two numbers.

---

## Advantages of Recursive Decomposition

- Naturally introduces **parallelism**.
- Works well with problems expressible as **divide-and-conquer**.
- Easy to implement and reason about.

## Limitations

- Not suitable for problems where subproblems are **highly dependent**.
  - May cause **overhead** if the problem is divided too finely.
- 

### Key takeaway:

Recursive decomposition is the foundation of many parallel algorithms. By breaking a large task into smaller independent tasks (like in quicksort or finding the minimum), you expose opportunities for **concurrent execution** across processors.

---

Would you like me to explain **Data Decomposition** (the next type) in the same clear and short style next?

That's an excellent and detailed summary — you've covered **Section 3.2 “Decomposition Techniques”** very well from the book *Introduction to Parallel Computing* by Grama et al. 

Here's a **simplified conceptual summary + key points** to help you **remember or explain it easily** (e.g., for exams, viva, or notes):

---

## ◆ 3.2 Decomposition Techniques — Summary

### Goal:

To split a problem into smaller *concurrent tasks* that can run in parallel (defined by a **task-dependency graph**).

There's no single “best” way — but the main techniques are:

- 
1. **Recursive Decomposition**
  2. **Data Decomposition**
  3. **Exploratory Decomposition**
  4. **Speculative Decomposition**
  5. **Hybrid Decomposition** (combination of the above)
- 

## 1. Recursive Decomposition

**Idea:** Use the **divide-and-conquer** strategy.

- Split the main problem into **independent subproblems**.
- Solve subproblems *recursively* and then combine results.
- Each subproblem can run in **parallel**.

**Example:**

- **Quicksort** — after choosing a pivot, two recursive calls (for  $A_0$  and  $A_1$ ) can run concurrently.
- **Finding Minimum** — split the array in half, find min recursively, then take min of two results.

**Key Point:**

Natural concurrency increases as recursion goes deeper.

---

## 2. Data Decomposition

**Idea:** Partition the **data** and assign computations to each partition.

**Types:**

1. **Partitioning Output Data:**  
Each output element is computed independently.  
→ Example: **Matrix multiplication** (each submatrix  $C_{ij}$  computed separately).
2. **Partitioning Input Data:**  
Divide input data; each task processes its part and later combines results.  
→ Example: **Sum of numbers**, or **finding frequency of itemsets** in a database.
3. **Partitioning Both Input and Output:**  
Split both sides to increase concurrency further.
4. **Partitioning Intermediate Data:**  
Decompose multi-stage algorithms at an intermediate stage for more parallelism.  
→ Example: In **matrix multiplication**, introducing an intermediate matrix D for subresults increases concurrency but uses more memory.

**Rule:**

 **Owner-Computes Rule** — each task computes results for the data it owns.

---

## 3. Exploratory Decomposition

**Idea:** Used in **search-based** or **exploration** problems.

- The search space is split into multiple parts.
- Each part is searched **concurrently**.
- Stop when one task finds the solution.

**Example:**

- **15-puzzle problem:** Each task explores a subtree of possible moves until one finds the correct sequence.

**Important:**

- Parallel version may do **less or more** total work than the serial one.  
(If the solution is found early — huge speedup; if late — wasted work.)
- 

## 4. Speculative Decomposition

**Idea:** Used when the program may take **different branches** depending on conditions not yet known.

- Execute multiple possible branches **in parallel** before the decision is known.
- When the condition resolves, keep the correct result, discard others.

**Example:**

- **Parallel discrete event simulation:** Tasks simulate different possible inputs before knowing the real one.

**Note:**

Performs **extra (wasted)** computation but reduces total runtime.

**Difference from Exploratory:**

Feature	Exploratory	Speculative
Unknown Part Output (solution)		Input (condition)

Feature	Exploratory	Speculative
Serial Work	May explore multiple paths	Explores only one path
Parallel Work	May be less or more	Always equal or more

---

## 5. Hybrid Decomposition

**Idea:** Combine multiple techniques for better efficiency.

**Examples:**

- **Finding Minimum:**  
Use **data decomposition** (split array into p parts) + **recursive decomposition** (find min among p partial results).
  - **Parallel Quicksort:**  
Combine **recursive** + **input decomposition** to balance task sizes and improve concurrency.
- 

## In Short

Technique	Based On	Example	Key Feature
Recursive	Divide & Conquer	Quicksort, Min Finding	Natural concurrency via recursion
Data	Partition Data	Matrix Multiplication	Most general technique
Exploratory	Search Space	15-Puzzle	Stop when solution found
Speculative	Future Branching	Event Simulation	Parallelize uncertain future paths
Hybrid	Combination	Parallel Quicksort, Min Finding	Best of multiple worlds

---

Would you like me to make a **diagram summary (flowchart + examples)** of these decomposition techniques for your notes or presentation? It'll make it very easy to visualize and remember.

That's an excellent and complete section from your textbook — it explains **all the core decomposition techniques** used in parallel algorithm design.

Here's a **simplified summary** of the key points so you can remember them easily for exams or project explanations 

---

## 3.2 Decomposition Techniques

Decomposition = breaking a problem into **smaller tasks** that can run **concurrently**.

There are **4 main types** (plus hybrid):

---

### **1 Recursive Decomposition**

**Used in:** divide-and-conquer problems

**Idea:** break a problem into smaller subproblems recursively.

**Example:**

- **Quicksort** → divide array into two parts based on a pivot, sort both parts concurrently.
- **Finding Minimum** → divide array into halves, find min of each half recursively, then compare.

 **Key point:** subproblems are independent → can run in parallel.

---

### **2 Data Decomposition**

**Used in:** problems operating on large data structures (arrays, matrices).

**Idea:** split data (input, output, or intermediate) and assign each portion to a processor.

#### Ways to partition data:

- **Output data partitioning:** each task computes part of the output (e.g., one block of a matrix product).
- **Input data partitioning:** each task processes part of the input (e.g., partial sum).
- **Intermediate data partitioning:** divide temporary data between stages (e.g., matrix multiplication storing partial products).

 **Rule:** “**Owner-computes rule**” — each processor computes the data it owns.

#### Examples:

- Matrix multiplication
  - Computing itemset frequencies in a transaction database
-

## **3** Exploratory Decomposition

**Used in:** search or exploration problems.

**Idea:** divide the **search space** into parts and explore all concurrently until a solution is found.

### Example:

- **15-Puzzle** — each processor explores a different branch of the possible moves.

### Special feature:

- Some tasks might stop early when another finds the answer.
  - Total work may be **less or more** than serial version (depends where the solution lies).
- 

## **4** Speculative Decomposition

**Used in:** problems with conditional branches or unknown future inputs.

**Idea:** start executing **multiple possible future computations** in parallel **before** knowing which is correct.

### Example:

- **Parallel discrete event simulation** — simulate multiple possible future events/speculative paths.

### Note:

- Some computations may be **wasted** (if the speculation was wrong).
  - Useful when the cost of waiting is high.
- 

## **5** Hybrid Decomposition

**Used in:** complex problems combining multiple strategies.

**Idea:** combine two or more decomposition techniques for efficiency.

### Examples:

- **Finding minimum:** first use data decomposition, then recursive decomposition to combine results.
- **Parallel Quicksort:** combine recursive and input-data decomposition.

---

## ▣ Summary Table

Type	Basic Idea	Example	Key Feature
Recursive	Divide problem recursively	Quicksort, Min finding	Natural concurrency
Data	Split data (input/output/intermediate)	Matrix multiplication	Owner-computes rule
Exploratory	Split search space	15-puzzle	Variable total work
Speculative	Parallelize future possible branches	Event simulation	May waste computation
Hybrid	Combine two or more methods	Quicksort + Input partition	Maximized concurrency

---

Would you like me to make **diagrams** for each decomposition type (like simplified versions of Figures 3.8–3.21) so you can use them in your notes or presentation slides?