**1. Introduction to Parallel Algorithm Design**

**Core Concepts**

* **Parallel Algorithm**: A computational procedure to solve a problem using multiple processors, requiring explicit specification of:
  1. **Concurrency**: Identifying steps that can execute simultaneously.
  2. **Mapping**: Assigning work to processes.
  3. **Data Distribution**: Managing input/output/intermediate data.
  4. **Synchronization**: Coordinating process execution.

**Key Challenges**

* **Performance Trade-offs**: Choices in decomposition/mapping affect efficiency.
* **Architecture Dependence**: Optimal designs vary across platforms (e.g., shared vs. distributed memory).

**2. Preliminaries**

**Decomposition & Tasks**

* **Decomposition**: Dividing computation into smaller **tasks** (units of work).
  + Tasks may vary in size.
  + Example: Matrix-vector multiplication decomposes into:
    - **Fine-grained**: Each task computes one dot product (row × vector).
    - **Coarse-grained**: Each task computes a block of rows (Figure 3.1).

**Task Dependency Graphs**

* **Definition**: Directed acyclic graph (DAG) where:
  + **Nodes** = Tasks.
  + **Edges** = Dependencies (task A must finish before task B starts).
* **Examples**:
  1. **Matrix-Vector Multiply**: No dependencies (disconnected graph).
  2. **Database Query**: Tasks depend on intermediate table results (Figures 3.2–3.3).

**3. Granularity & Concurrency**

**Granularity**

* **Fine-grained**: Many small tasks (e.g., one row per task).
* **Coarse-grained**: Fewer large tasks (e.g., row blocks per task).
* **Impact**:
  + Fine-grained → Higher concurrency but more communication overhead.
  + Coarse-grained → Lower overhead but may underutilize resources.

**Degree of Concurrency**

* **Maximum Concurrency**: Peak simultaneous tasks (e.g., 4 in database query).
* **Average Concurrency**: Total work / Critical path length.
  + **Critical Path**: Longest dependency chain (e.g., Figure 3.4).
    - Determines minimum execution time.
    - Shorter critical path → Higher concurrency.

**4. Examples**

**Example 1: Matrix-Vector Multiplication**

* **Decomposition**:
  + **Task**: Compute one element of output vector y[i]=A[i,:]⋅b*y*[*i*]=*A*[*i*,:]⋅*b*.
  + **Dependency Graph**: Disconnected (all tasks independent).
* **Granularity**:
  + Fine-grained: n*n* tasks for n×n*n*×*n* matrix.
  + Coarse-grained: p*p* tasks (p≪n*p*≪*n*), each computing n/p*n*/*p* elements.

**Example 2: Database Query Processing**

* **Query**: MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR "White").
* **Tasks**:
  1. Filter tables by MODEL, YEAR, COLOR.
  2. Compute intersections/unions (e.g., Civic ∩ 2001).
* **Dependency Graphs**:
  1. **Version 1**: Intersect Civic and 2001 first (Figure 3.2).
  2. **Version 2**: Union Green and White first (Figure 3.3).
* **Critical Path**: Varies with decomposition (e.g., 3 steps in Version 1 vs. 2 in Version 2).

**5. Key Metrics & Analysis**

**Critical Path Analysis**

* **Definition**: Sum of weights (task sizes) along the longest path in the DAG.
* **Example**:
  + Database query (Figure 3.2): Critical path = 3 tasks.
  + Matrix-vector multiply: Critical path = 1 task (no dependencies).
* **Implications**:
  + Limits speedup (Amdahl’s Law).
  + Guides optimization (e.g., load balancing).

**Concurrency Metrics**

| **Metric** | **Formula** | **Example (Database Query)** |
| --- | --- | --- |
| Maximum Concurrency | Max simultaneous tasks | 4 (leaf tasks in DAG) |
| Average Concurrency | Total work / Critical path length | ~Total tasks/3 |

**6. Practical Considerations**

**Trade-offs in Design**

1. **Granularity**:
   * **Fine-grained**: Better load balance but high communication.
   * **Coarse-grained**: Lower overhead but risk of idle processors.
2. **Dependencies**:
   * Independent tasks (e.g., matrix-vector) → Easy parallelism.
   * High dependencies (e.g., PDE solvers) → Complex synchronization.
3. **Architecture Fit**:
   * Shared memory: Fine-grained tasks.
   * Distributed memory: Coarse-grained to minimize communication.

**Optimization Strategies**

* **Task Fusion**: Combine small tasks to reduce overhead.
* **Load Balancing**: Distribute work evenly (e.g., cyclic vs. block decomposition).
* **Critical Path Reduction**: Parallelize bottleneck tasks.

**Summary of Key Points**

1. **Decomposition**: Break problem into tasks (fine/coarse-grained).
2. **Dependencies**: Model via task graphs; critical path determines speedup.
3. **Concurrency**: Maximize parallelizable work; balance granularity.
4. **Examples**: Matrix ops (independent tasks) vs. queries (dependent tasks).
5. **Metrics**: Critical path length, max/average concurrency.

**Example Calculations**

1. **Critical Path**:
   * Database query (Figure 3.2): 3 steps (Civic ∩ 2001 ∩ (Green ∪ White)).
   * Speedup limited to ≤p≤*p* (processors).
2. **Average Concurrency**:
   * If total work = 7 tasks and critical path = 3:  
     Average concurrency=7/3≈2.33Average concurrency=7/3≈2.33.

This structured breakdown ensures **all slide content is covered**, with **clear definitions**, **examples**, and **visualizable concepts**. Let me know if you need further elaboration!

**1. Granularity, Concurrency, and Task Interaction**

**Key Concepts**

* **Granularity Limits**:
  + There's an inherent bound on how fine-grained a decomposition can be.
  + Example: Matrix-vector multiplication has O(n2)*O*(*n*2) operations → Cannot be decomposed into more than O(n2)*O*(*n*2) tasks.
* **Task Interaction**:
  + Tasks often share data (input/output/intermediate), causing dependencies.
  + Example: In database queries, intermediate tables are shared between tasks.

**Task-Interaction Graphs**

* **Definition**:
  + **Nodes**: Tasks.
  + **Edges**: Interactions between tasks (data sharing).
  + Weights: Can represent computation load (nodes) or communication cost (edges).
* **Properties**:
  + Undirected edges (bidirectional data flow) unless specified.
  + Superset of task-dependency graphs (includes all interactions, not just dependencies).
* **Example**:
  + Sparse matrix-vector multiplication: Tasks interact based on non-zero elements (Figure 3.5).

**2. Processes and Mapping**

* **Challenge**:
  + Typically, #tasks >> #processors → Need to map multiple tasks to each processor.
* **Goals**:
  + Balance workload.
  + Minimize communication (place interacting tasks on the same processor if possible).

**3. Decomposition Techniques**

**A. Recursive Decomposition**

* **Approach**: Divide-and-conquer.
  + Split problem into independent subproblems → Solve recursively → Combine results.
* **Example**: Quicksort (Figure 3.6):
  + **Divide**: Partition array around pivot.
  + **Conquer**: Recursively sort subarrays.
  + **Task Graph**: Binary tree where each node is a partitioning task.
  + **Concurrency**: Increases as we move down the tree (more subarrays to sort in parallel).

**B. Data Decomposition**

**i. Partitioning Output Data**

* **When**: Each output element can be computed independently.
* **Example**: Matrix-vector multiplication (Figure 3.7):
  + Output vector partitioned → Each task computes one segment.

**ii. Partitioning Input Data**

* **When**: Output cannot be easily partitioned (e.g., sum/min/max).
* **Example**: Sum of N*N* numbers:
  + Partition input into p*p* subsets → Each task computes partial sum → Final reduction.

**iii. Partitioning Both Input & Output**

* **Example**: Itemset frequency counting (Figure 3.8):
  + Partition transactions (input) and frequencies (output) → 4 tasks compute partial frequencies → Combine results.

**iv. Partitioning Intermediate Data**

* **Use Case**: Multi-stage algorithms (e.g., MapReduce).
  + Example: Distributed sorting (partition intermediate sorted segments).

**Owner-Computes Rule**

* **Principle**: Each task performs all computations involving its owned data partition.
* **Applies to**: Input/output partitioning.

**C. Exploratory Decomposition**

* **Use Case**: Search problems.
* **Approach**: Partition search space → Explore concurrently.
* **Example**: 15-puzzle (Figure 3.9):
  + Each task explores a different move sequence.
* **Anomalies**:
  + **Superlinear speedup**: Early termination in one branch.
  + **Sublinear speedup**: Redundant work across branches.

**D. Speculative Decomposition**

* **Use Case**: Branching computations (e.g., switch statements).
* **Approach**:
  + Execute likely branches in parallel before condition is resolved.
  + Discard incorrect paths.
* **Example**: Network simulation (Figure 3.10):
  + Simulate multiple delay parameters concurrently → Keep correct result.
* **Optimization**: Prioritize most likely branch to minimize wasted work.

**E. Hybrid Decomposition**

* **Combination**: Use multiple techniques in different stages.
* **Examples**:
  1. **Finding Minimum**:
     + Data decomposition: Partition input → Compute local minima.
     + Recursive decomposition: Combine results hierarchically.
  2. **Quicksort**:
     + Data decomposition: Partition array into chunks.
     + Recursive decomposition: Sort subarrays in parallel.
  3. **Discrete Event Simulation**:
     + Data decomposition: Partition events.
     + Speculative decomposition: Simulate likely next events.

**4. Practical Considerations**

**Trade-offs**

| **Technique** | **Best For** | **Challenges** |
| --- | --- | --- |
| Recursive | Divide-and-conquer algorithms | Limited concurrency in early stages |
| Data (Output) | Embarrassingly parallel problems | Requires independent outputs |
| Data (Input) | Reductions (sum, min, etc.) | Needs follow-up combination step |
| Exploratory | Search/optimization | Risk of redundant work |
| Speculative | Branch-heavy code | Wasted computation if wrong branch |

**Optimization Strategies**

1. **Load Balancing**:
   * Evenly distribute tasks (e.g., cyclic partitioning for irregular workloads).
2. **Communication Minimization**:
   * Cluster interacting tasks (e.g., block partitioning for matrix operations).
3. **Granularity Adjustment**:
   * Coarse-grained for high communication, fine-grained for high concurrency.

**5. Examples in Detail**

**A. Quicksort (Recursive + Data Hybrid)**

1. **Initial Data Decomposition**:
   * Partition array into p*p* chunks → Each processor sorts its chunk.
2. **Recursive Decomposition**:
   * Merge sorted chunks in parallel (tree reduction).

* **Why Hybrid?**:
  + Pure recursion limits concurrency early (only one partition initially).
  + Data decomposition ensures all processors are utilized from the start.

**B. Sparse Matrix-Vector Multiply**

* **Challenge**: Non-zero elements are irregularly distributed.
* **Decomposition**:
  1. Partition output vector (owner-computes rule).
  2. Tasks interact based on non-zero entries (task-interaction graph = dependency graph).
* **Communication**: Tasks need remote vector elements → Message passing.

**6. Key Metrics**

* **Critical Path**: Longest dependency chain (limits speedup).
* **Concurrency**:
  + **Maximum**: Peak parallel tasks (e.g., leaves in quicksort tree).
  + **Average**: Total work / Critical path length.
* **Efficiency**:

Efficiency=Speedupp=Tserialp⋅TparallelEfficiency=*p*Speedup​=*p*⋅*T*parallel​*T*serial​​

**Summary**

1. **Decomposition**:
   * Choose based on problem structure (data, recursion, search, etc.).
   * Balance granularity vs. overhead.
2. **Mapping**:
   * Assign tasks to processors to minimize idle time/communication.
3. **Interaction**:
   * Model via task-interaction graphs to optimize data locality.
4. **Hybrid Approaches**:
   * Combine techniques (e.g., data + recursion) for better scalability.

This structured breakdown ensures **all slide content is covered**, with **clear definitions**, **examples**, and **visualizable concepts**. Let me know if you'd like to dive deeper into any section!

**1. Characteristics of Tasks and Interactions**

**A. Task Characteristics**

**1. Task Generation**

* **Static Generation**:
  + All tasks are known before execution.
  + Examples:
    - *Data decomposition* (matrix multiplication).
    - *Static recursive decomposition* (finding minimum of a list).
  + **Advantage**: Easier to map and schedule.
* **Dynamic Generation**:
  + Tasks created during execution.
  + Examples:
    - *Quicksort* (task tree shape depends on pivot choices).
    - \*15-puzzle\* (solution path length unknown).
  + **Challenge**: Requires adaptive mapping strategies.

**2. Task Size**

* **Uniform Tasks**: Similar execution times (e.g., matrix multiplication).
* **Non-Uniform Tasks**: Variable execution times (e.g., quicksort partitions).

**3. Knowledge of Task Size**

* **Known Sizes**: Facilitates static load balancing (e.g., matrix ops).
* **Unknown Sizes**: Requires dynamic load balancing (e.g., search problems).

**4. Size of Task Data**

* **Data Locality**: Tasks with large data should minimize movement.
  + Example: Sparse matrix-vector multiply (non-zero elements dictate data needs).

**B. Inter-Task Interactions**

**1. Static vs. Dynamic Interactions**

* **Static Interactions**:
  + Predetermined timing and partners.
  + Example: Image dithering (fixed neighbor communication).
  + **Ease of Implementation**: Straightforward in message-passing.
* **Dynamic Interactions**:
  + Unpredictable timing/partners.
  + Example: Sparse matrix multiplication (irregular non-zero pattern).
  + **Challenges**:
    - Harder in message-passing (requires polling/synchronization).
    - Easier in shared-memory (direct memory access).

**2. Regular vs. Irregular Interactions**

* **Regular**: Structured patterns (e.g., grid-based computations).
  + **Optimization**: Exploit spatial locality (e.g., block partitioning).
* **Irregular**: Unstructured (e.g., graph algorithms).
  + **Challenge**: Requires flexible communication (e.g., graph partitioning).

**3. Read-Only vs. Read-Write Interactions**

* **Read-Only**: Tasks share data without modification.
  + Example: Broadcast operations.
  + **Advantage**: No synchronization needed.
* **Read-Write**: Tasks modify shared data.
  + Example: Parallel reduction.
  + **Challenge**: Requires locks/atomic operations.

**4. One-Way vs. Two-Way Interactions**

* **One-Way**: Single task initiates (e.g., producer-consumer).
  + **Shared-Memory**: Natural fit (direct reads/writes).
  + **Message-Passing**: Requires conversion to two-way (e.g., send/receive pairs).
* **Two-Way**: Both tasks participate (e.g., MPI point-to-point).
  + **Example**: Pipeline stages exchanging data.

**2. Mapping Techniques for Load Balancing**

**A. Objectives**

1. **Minimize Interaction Overhead**:
   * Cluster interacting tasks on the same process.
2. **Minimize Idling**:
   * Evenly distribute work to avoid idle processes.

**B. Static Mapping**

* **When to Use**:
  + Tasks are static and uniform.
  + Data sizes and interactions are predictable.
* **Techniques**:
  + **Block Distribution**: Assign contiguous chunks (e.g., matrix rows).
  + **Cyclic Distribution**: Assign tasks in round-robin (improves load balance for uneven workloads).
  + **Graph Partitioning**: Minimize edge cuts (for irregular tasks).
* **Example**: Matrix multiplication with block-distributed output.

**C. Dynamic Mapping**

* **When to Use**:
  + Tasks are dynamic or non-uniform.
  + Interaction patterns are unpredictable.
* **Techniques**:
  + **Task Queues**: Centralized or distributed work pools.
  + **Work Stealing**: Idle processes steal tasks from busy ones.
* **Example**: Quicksort with dynamically generated partitions.

**D. Trade-offs**

| **Strategy** | **Pros** | **Cons** |
| --- | --- | --- |
| **Static Mapping** | Low runtime overhead | Poor load balance if tasks vary |
| **Dynamic Mapping** | Adapts to workload variations | Higher scheduling overhead |

**3. Practical Examples**

**A. Sparse Matrix-Vector Multiplication**

* **Task Decomposition**: Partition output vector (owner-computes rule).
* **Interactions**:
  + *Irregular* (depends on non-zero pattern).
  + *Dynamic* (if matrix structure changes).
* **Mapping**:
  + Use graph partitioning to minimize communication.

**B. Quicksort**

* **Task Generation**: Dynamic (depends on pivot choices).
* **Interactions**:
  + *Static* in recursive merging (if known).
  + *Dynamic* in partitioning.
* **Mapping**:
  + Hybrid: Initial data decomposition + dynamic work stealing.

**C. 15-Puzzle (Exploratory Decomposition)**

* **Task Generation**: Dynamic (solution path unknown).
* **Interactions**:
  + *Read-only* (shared board state).
  + *One-way* (tasks report solutions independently).
* **Mapping**:
  + Dynamic task queues with termination detection.

**4. Key Takeaways**

1. **Task Characteristics**:
   * Generation (static/dynamic), size (uniform/non-uniform), data locality.
2. **Interaction Types**:
   * Static/dynamic, regular/irregular, read/write, one-way/two-way.
3. **Mapping Strategies**:
   * Static for predictable workloads; dynamic for adaptability.
4. **Load Balancing**:
   * Goal: Minimize idling + communication overhead.

**Summary Table: Interaction Types**

| **Characteristic** | **Static** | **Dynamic** | **Read-Only** | **Read-Write** |
| --- | --- | --- | --- | --- |
| **Definition** | Predetermined partners | Unpredictable partners | No data modification | Data modification |
| **Example** | Image dithering | Sparse matrix multiply | Broadcast | Reduction |
| **Programming Ease** | Easy (MPI) | Hard (requires polling) | No sync needed | Needs locks/atomics |

This structured breakdown ensures **all slide content is covered**, with **clear definitions**, **examples**, and **practical insights**. Let me know if you'd like to explore any section further!

Block Distribution Schemes for Data Mapping

**Definition**:  
Block distribution divides data into contiguous blocks assigned to different processors. For an array of size N and P processors, each processor gets N/P contiguous elements.

**Key Characteristics**:

* Simple to implement
* Preserves spatial locality
* Minimizes communication for nearest-neighbor interactions

**Example**:  
For N=8 and P=2:  
Processor 0: elements 0-3  
Processor 1: elements 4-7

**Advantages**:

* Low communication overhead for local operations
* Good cache utilization

**Disadvantages**:

* Can lead to load imbalance if computation varies across data sections
* Not optimal for all access patterns

Cyclic and Block Cyclic Distributions

**Cyclic Distribution**:

* Elements are assigned to processors in round-robin fashion
* For N=8 and P=2:  
  Processor 0: elements 0,2,4,6  
  Processor 1: elements 1,3,5,7

**Block Cyclic Distribution**:

* Combines block and cyclic approaches
* Data divided into blocks of size B, then blocks assigned cyclically
* For N=8, P=2, B=2:  
  Processor 0: blocks 0-1, 4-5  
  Processor 1: blocks 2-3, 6-7

**When to Use**:

* Cyclic: When load is uniformly distributed but fine-grained
* Block Cyclic: Balances benefits of both approaches
* Particularly useful for linear algebra operations

Randomized Block Distribution

**Definition**:  
Data blocks are assigned to processors using a random or pseudorandom function.

**Characteristics**:

* Provides probabilistic load balancing
* Disrupts any inherent patterns in data/computation
* Requires more communication than structured distributions

**Use Cases**:

* When computation load is unpredictable
* For problems with irregular data access patterns
* When simple static distributions fail to balance load

**Advantages**:

* Simple to implement
* Good average-case performance

**Disadvantages**:

* May require more communication
* Less predictable performance

Graph Partitioning

**Purpose**:  
Partition computational graphs to:

1. Balance workload
2. Minimize communication between partitions

**Key Metrics**:

* Edge cut: Number of edges between partitions
* Vertex cut: Number of vertices shared between partitions

**Common Algorithms**:

1. Spectral Partitioning:
   * Uses eigenvectors of graph Laplacian
   * Provides theoretical guarantees but computationally expensive
2. Geometric Partitioning:
   * Uses spatial coordinates of vertices
   * Fast but requires geometric information
3. Multilevel Methods:
   * Coarsen graph, partition coarse graph, then refine
   * Good balance between quality and speed

**Applications**:

* Sparse matrix operations
* Finite element methods
* Network analysis

Mapping Based on Task Partitioning

**Approach**:

1. Decompose computation into tasks
2. Analyze task dependency graph
3. Assign tasks to processors considering:
   * Computational load
   * Communication requirements
   * Data dependencies

**Techniques**:

1. List Scheduling:
   * Prioritize tasks based on dependencies
   * Assign to first available processor
2. Clustering:
   * Group tightly-coupled tasks
   * Assign clusters to processors
3. Task Graph Scheduling:
   * Model as graph optimization problem
   * Minimize makespan considering communication

**Challenges**:

* NP-hard problem in general
* Requires good estimates of task durations
* Dynamic task generation complicates scheduling

Hierarchical Mapping

**Concept**:  
Apply different mapping strategies at different levels of hardware hierarchy.

**Typical Hierarchy Levels**:

1. Nodes (shared memory within node)
2. Sockets/processors
3. Cores
4. Vector units

**Strategy Example**:

1. First level: Distribute across nodes using graph partitioning
2. Second level: Within node, use shared memory task queue
3. Third level: SIMD vectorization within cores

**Advantages**:

* Matches algorithm to hardware capabilities
* Can optimize for different constraints at each level
* Combines benefits of different approaches

**Implementation Considerations**:

* Requires understanding of hardware architecture
* May need hybrid programming models (e.g., MPI+OpenMP)
* Communication costs vary between levels

Comparison of Mapping Techniques

| **Technique** | **Best For** | **Load Balance** | **Communication** | **Implementation** |
| --- | --- | --- | --- | --- |
| Block | Regular data, local operations | Moderate | Low | Easy |
| Cyclic | Fine-grained uniform load | Good | Medium | Easy |
| Block Cyclic | Semi-regular operations | Good | Medium | Moderate |
| Randomized | Irregular, unknown patterns | Good | High | Easy |
| Graph Partitioning | Sparse, irregular computations | Very Good | Minimal | Hard |
| Task Partitioning | Task-parallel applications | Good | Varies | Hard |
| Hierarchical | Complex modern architectures | Very Good | Optimized | Very Hard |

Practical Considerations

1. **Data vs. Task Parallelism**:
   * Data parallelism often uses block/cyclic distributions
   * Task parallelism often uses graph/task partitioning
2. **Hybrid Approaches**:
   * Common to combine techniques (e.g., graph partitioning at coarse level with block cyclic at fine level)
3. **Dynamic Adaptation**:
   * Some systems use runtime measurements to adjust mappings
4. **Tool Support**:
   * METIS/ParMETIS for graph partitioning
   * HPX, Charm++ for task-based approaches
   * MPI for data distribution

This comprehensive explanation covers all requested topics while maintaining technical accuracy and practical relevance. Each section provides sufficient detail to understand the concepts while showing how they relate to each other in parallel algorithm design.

Methods for Containing Interaction Overheads (Section 3.5)

1. Data Locality

**Definition**: Data locality refers to optimizing data access patterns to minimize communication between processing elements.

**Types**:

1. **Spatial Locality**:
   * Accesses data elements close to each other in memory
   * Example: Processing contiguous array elements on the same processor
   * Benefit: Reduces communication distance in distributed systems
2. **Temporal Locality**:
   * Reuses recently accessed data
   * Example: Caching frequently used variables
   * Benefit: Minimizes repeated data transfers

**Implementation**:

* Block data distribution keeps related data together
* Loop tiling improves cache reuse
* Data alignment matches memory access patterns

2. Maximizing Data Locality

**Strategies**:

1. **Minimize Volume of Data Exchange**:
   * Store intermediate results locally
   * Example: Partial dot products computed locally before final summation
   * Reduces shared memory accesses from n to p (for p processors)
2. **Minimize Frequency of Interactions**:
   * Access data in large chunks
   * Amortizes communication startup costs
   * Example: Fetching entire cache lines in shared memory
3. **Application to Sparse Matrices**:
   * Gather required vector elements first
   * Then perform local computations
   * Reduces multiple small messages to fewer large ones

3. Minimizing Contention and Hot Spots

**Causes**:

* Multiple tasks accessing same resources:
  + Network links
  + Memory blocks
  + Centralized processes

**Solutions**:

* Randomized work distribution
* Data replication
* Decentralized task queues
* Example: Hash-based data partitioning

4. Overlapping Computations with Interactions

**Techniques**:

1. **Early Initiation**:
   * Start communication before data is needed
   * Requires predictable access patterns
2. **Non-blocking Operations**:
   * MPI\_Isend/Irecv in message passing
   * Hardware prefetching in shared memory
3. **Work Request Anticipation**:
   * Request more work before current tasks complete
   * Requires workload estimation

**Hardware Support**:

* Prefetch instructions
* DMA engines
* Multithreaded cores

5. Replicating Data or Computations

**When to Use**:

* Frequent read-only access to shared data
* Irregular access patterns
* Small data size relative to computation

**Trade-offs**:

* Pros: Eliminates subsequent communication
* Cons: Increases memory usage (linearly with processors)

**Examples**:

* Replicating lookup tables
* Caching frequently accessed data
* Broadcast of small configuration data

6. Overlapping Interactions

**Example**: One-to-all broadcast

* Step 1: P0 → P2
* Step 2: Concurrently:
  + P0 → P1
  + P2 → P3
* Completes in log(p) steps

**Benefits**:

* Utilizes full network bandwidth
* Parallel communication paths

Parallel Algorithm Models (Section 3.6)

1. Data-Parallel Model

**Characteristics**:

* Same operation on different data
* Static/semi-static task mapping
* Regular communication patterns

**Examples**:

* Matrix operations
* Image processing
* SIMD architectures

**Advantages**:

* Simple programming model
* Good load balance for regular problems

2. Task Graph Model

**Characteristics**:

* Explicit task dependencies
* Large computation to data ratio
* Static mapping common

**Applications**:

* Sparse matrix computations
* Divide-and-conquer algorithms
* Dependency-heavy workflows

**Optimizations**:

* Critical path analysis
* Task clustering
* Locality-aware scheduling

3. Work Pool Model

**Features**:

* Dynamic task assignment
* Centralized or distributed queues
* Requires termination detection

**Variants**:

* Shared task queue
* Work stealing
* Task stealing

**Use Cases**:

* Irregular problems
* Unpredictable task sizes
* Branch-and-bound algorithms

4. Master-Slave Model

**Components**:

1. Master:
   * Generates work
   * Maintains task queue
   * Handles load balancing
2. Workers:
   * Request tasks
   * Process data
   * Return results

**Design Considerations**:

* Task granularity
* Asynchronous communication
* Fault tolerance

5. Pipeline Model

**Characteristics**:

* Stream processing
* Producer-consumer pattern
* Balanced stage workloads

**Topologies**:

* Linear pipelines
* Tree structures
* General graphs

**Examples**:

* Video processing
* Network packet processing
* Compiler phases

6. Hybrid Models

**Combination Approaches**:

1. **Hierarchical**:
   * Different models at different scales
   * Example: Data-parallel within nodes, task-parallel across nodes
2. **Phased**:
   * Different models for algorithm phases
   * Example: Task graph for setup, data-parallel for computation

**Case Study - Parallel Quicksort**:

1. Initial data-parallel partitioning
2. Recursive task-parallel sorting
3. Pipeline for merging results

Key Concepts Summary

**Interaction Reduction Techniques**:

| **Technique** | **Benefit** | **When to Use** |
| --- | --- | --- |
| Data Locality | Reduces communication volume | Regular data access patterns |
| Computation Overlap | Hides latency | Predictable communication |
| Data Replication | Eliminates repeated transfers | Small, read-only data |
| Overlapped Communication | Utilizes full bandwidth | Independent communication paths |

**Algorithm Models Comparison**:

| **Model** | **Strengths** | **Weaknesses** |
| --- | --- | --- |
| Data-Parallel | Simple, scalable | Limited to regular problems |
| Task Graph | Handles dependencies | Scheduling complexity |
| Work Pool | Load balancing | Overhead for dynamic scheduling |
| Master-Slave | Flexible control | Master bottleneck risk |
| Pipeline | Stream processing | Balancing challenges |

This comprehensive explanation covers all concepts from the slides while maintaining technical accuracy and practical relevance. Each section provides detailed explanations with examples and clear organization for study purposes.