

Part V

Parallel DBS

# Overview

- Foundations
- Parallel Query Processing

- Parallel Database Systems (PDBS):
  - ▶ DB-Processing on Parallel Hardware Architectures
  - ▶ Goal: performance improvement by using parallel processors
- General overview of approaches:
  - ▶ Inter-TXN parallelization : simultaneous execution of independent parallel txns  $\rightsquigarrow$  improved throughput
  - ▶ Intra-TXN parallelization : simultaneous execution processing of operations within one txn  $\rightsquigarrow$  improved response time

# Speedup

- **Speedup:** Measure for performance improvement of a IT system through optimization
- For parallelization: Response Time (RT) speedup by using  $n$  processors

$$\text{RT Speedup}(n) = \frac{\text{RT for sequential processing with 1 processor}}{\text{RT for parallel processing on } n \text{ Processors}}$$

- According to Amdahl's Law optimization improvement limited by fraction  $0 \leq F_{\text{opt}} \leq 1$  of operations which can be optimized by parallelized execution

$$\text{RT Speedup} = \frac{1}{(1 - F_{\text{opt}}) + \frac{F_{\text{opt}}}{\text{Speedup}_{\text{opt}}}}$$

# Speedup

- Speedup furthermore limited by
  - ▶ Overhead for coordination and communication for parallel execution
  - ▶ Interferences between parallel executions through shared resource and locks
  - ▶ response time depending on slowest execution thread (non-equal distribution is called Skew: Processing Skew and Data Skew)
- Speedup limitation: there is a limit number  $n_{max}$  of processors from where on more processors do not improve performance or performance even declines

# Scaleup

- Speedup: increase processor number to improve response time for same problem size
- **Scaleup**: linear growth of number of processors with growing problem size (e.g. more users, more data, etc.)
- Response Time Scaleup:
  - ▶ Ratio of the Reponse Time for  $n$  processors and  $n$  times DB size compared to original problem with 1 processor
  - ▶ Goal: Response Time Scaleup = 1
- Throughput Scaleup:
  - ▶ Ratio of TXN using  $n$  Prozessoren compared to solution with 1 processor
  - ▶ Goal: Throughput Scaleup =  $n$

# Architectures of PDBMS

- Shared Everything →
  - ▶ Typical architecture: DBMS support for multi-processor computers
  - ▶ Ideal for Response Time Speedup
- Shared Disk →
  - ▶ DBMS support for tightly connected (e.g. fast network) nodes with shared disk access
  - ▶ Good for Response Time and Scalability
  - ▶ Requires: synchronization of disk accesses
- Shared Nothing →
  - ▶ Connected nodes with their own disks
  - ▶ Ideal for Scalability
  - ▶ Requires thoughtful data fragmentation

# Fragmentation in PDBMS

- Goal: use of horizontal fragmentation to avoid data skew
  - ▶ Perform part of operation on equal size fragments
  - ▶ Less data skew
    - less processing skew
    - optimal parallel execution
    - optimal speedup
- Fixed or dynamic assignment of processors to fragments possible

# Fragmentation in PDBMS /2

- Approaches:

- ▶ **Range-based Fragmentation:**

- ★ Assignment of tuples to disk based on pre-defined or dynamic range specifications for relation attributes
- ★ Complex task to define ranges that minimize data and processing skew

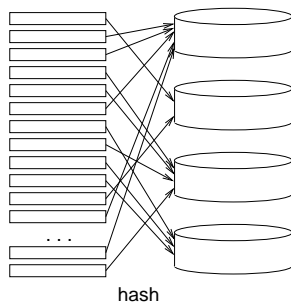
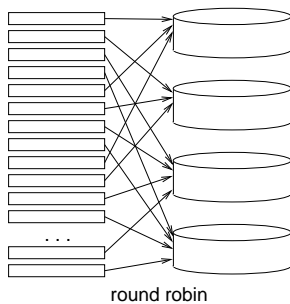
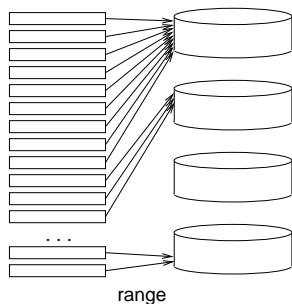
- ▶ **Hash-based Fragmentation:**

- ★ Assignment of tuples to disk based on hash function on relation attributes
- ★ More overhead for range queries

- ▶ **Round Robin Fragmentation:**

- ★ Assignment of tuples to disk upon creation: record  $i$  is assigned to disk  $(i \bmod M) + 1$  for  $M$  disks
- ★ Avoids skew, but no support for exact match or range queries

# Fragmentation in PDBMS /2



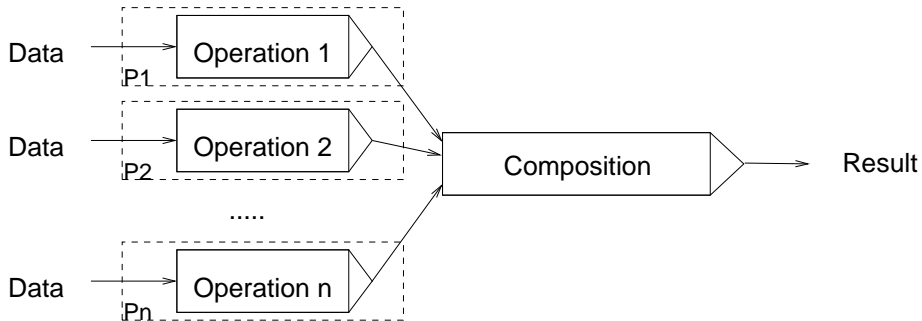
# Parallel Query Processing

- Intra-Query Parallelization
- Intra-Operation Parallelization
  - ▶ Unary operations (Selection, Projection, Aggregation)
  - ▶ Sorting
  - ▶ Joins
- Processor Allocation

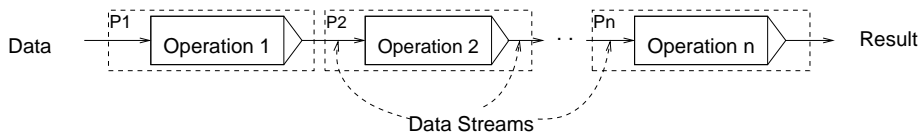
# Intra-Query Parallelization

- Independent Parallelization
  - ▶ Parallel execution of independent parts of a query, e.g. multi-way joins, separate execution threads below a union, etc.
- Pipelining Parallelization
  - ▶ Pipelining: processed data is considered as a data stream through sequence of operations (path from base relation to top of query plan tree)
  - ▶ Operations are executed by different processors with incoming data from processor handling lower operations in the query plan
- Intra-Operator Parallelization
  - ▶ Parallel processing of parts of one operation, e.g. selections from fragments, hash-based problem decomposition for join operations, etc.

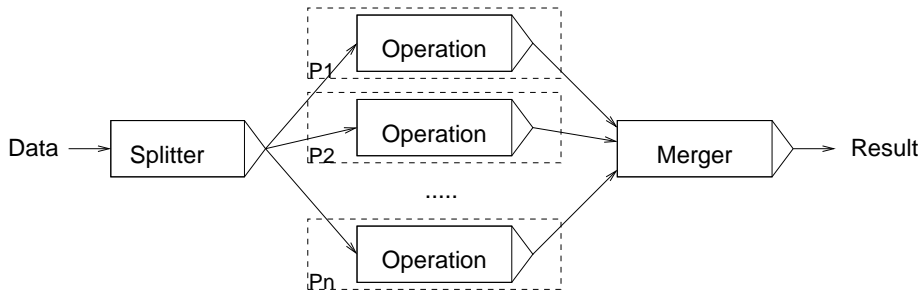
# Independent Parallelization



# Pipelining Parallelization



# Intra-Operator Parallelization



# Parallelization of unary Operations

- Selection:

$$r = \bigcup_i r_i \Rightarrow \sigma_P(r) = \bigcup_i \sigma_P(r_i)$$

- Projection without duplicate elimination: dito
- Duplicate elimination: using sorting ( $\rightarrow$ )
- Aggregate functions:
  - ▶  $\mathbf{min}(r.\text{Attr}) = \mathbf{min}(\mathbf{min}(r_1.\text{Attr}), \dots, \mathbf{min}(r_n.\text{Attr}))$
  - ▶  $\mathbf{max}(r.\text{Attr}) = \mathbf{max}(\mathbf{max}(r_1.\text{Attr}), \dots, \mathbf{max}(r_n.\text{Attr}))$

## Parallelization of Unary Operations /2

- Aggregate functions (if no duplicate elimination necessary):

- ▶  $\mathbf{count}(r.\mathbf{Attr}) = \sum_i \mathbf{count}(r_i.\mathbf{Attr})$

- ▶  $\mathbf{sum}(r.\mathbf{Attr}) = \sum_i \mathbf{sum}(r_i.\mathbf{Attr})$

- ▶  $\mathbf{avg}(r.\mathbf{Attr}) = \mathbf{sum}(r.\mathbf{Attr}) / \mathbf{count}(r.\mathbf{Attr})$

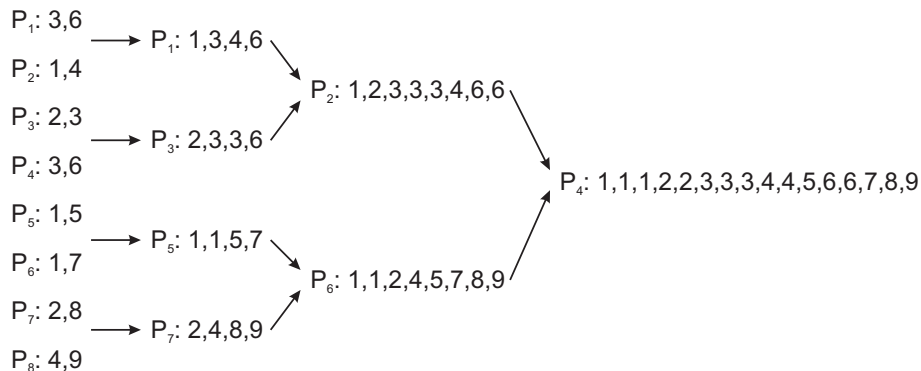
# Parallel Sorting

- Classification regarding number of in- and output streams:
  - ▶ 1:1, 1:many, many:1, many:many
- Requirements for (?:many):
  - ▶ Partial result on each node is sorted
  - ▶ Complete final result can be achieved by simple concatenation of partial results (no further merging necessary)

# Parallel Binary Merge Sort

- Many:1 approach
- Processing:
  - Phase 1: fragments are sorted locally on each node (Quicksort, External Merge Sort)
  - Phase 2: merging two partial results on one node at a time until all intermediate results are merged in one final result
- Merging can also be performed in parallel and even be pipelined

# Parallel Binary Merge Sort /2



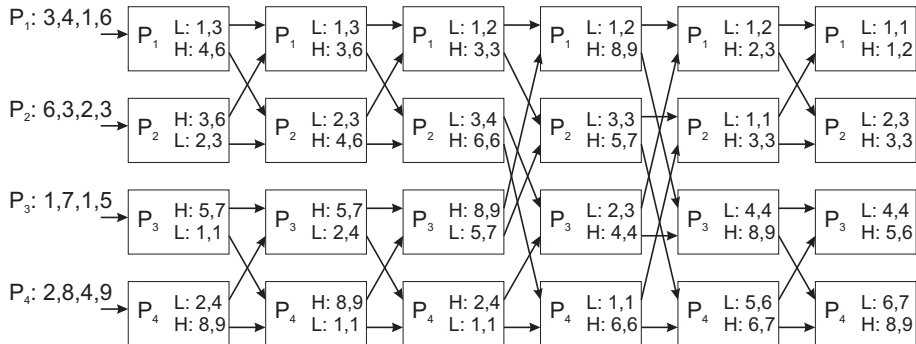
# Block Bitonic Sort

- Many:many-approach applying fixed number of processors (e.g. those managing a fragmented relation)
- 2 Phases
- ① Sort-Split-Ship
  - (a) Sort fragments on each node locally
  - (b) Split sorted fragments in two parts of equal size
    - ★ every value in one (lower) sub-fragment  $\leq$  every value in the other (higher) sub-fragment
  - (c) Ship sub-fragments to other nodes according to predefined scheme

## 2 2-Way-Merge Split

- (a) On arrival of two fragments: 2-Way-Merge to get one sorted fragment
- (b) Split and ship fragments according to 1(b) until
  - (1) every node  $P_i$  has a sorted fragment and
  - (2) every value in fragment at  $P_i \leq$  every value in fragment at  $P_j$  for  $i \leq j$
- Key point is shipping scheme, which is fixed for a certain number of nodes  $n$  (see following example)
- Number of necessary steps:  $\frac{1}{2} \log 2n(\log 2n + 1)$  for  $n$  nodes

# Block Bitonic Sort /3



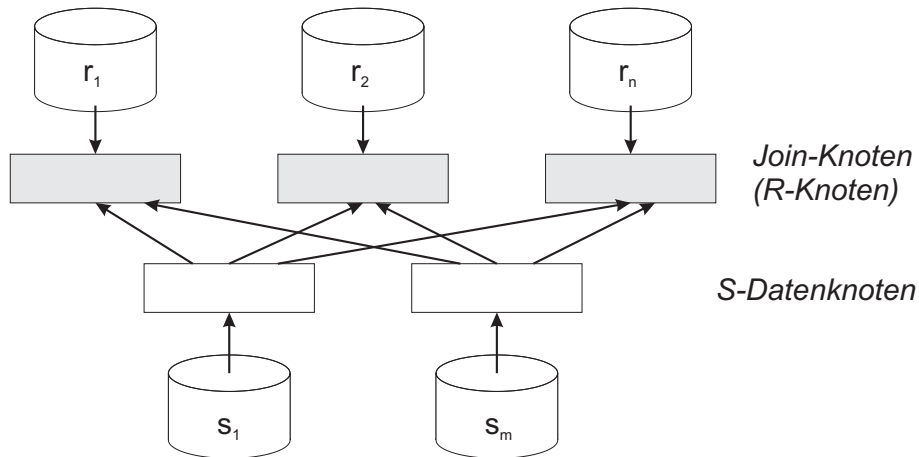
# Parallel Join-Processing

- Join-Processing on Multiple Processors supported by 2 main approaches
- Dynamic Replication
  - ▶ Replication (transfer) of the smaller relation  $r$  to each join node
  - ▶ Local execution of partial joins on each processor
  - ▶ Full result by union of partial results
- Dynamic Partitioning
  - ▶ Partition tuples to (ideal) same size partitions on each node
  - ▶ Distribution: hash function  $h$  or range partitioning

# Dynamic Replication

- 1 Assumption: relations  $S$  and  $R$  are fragmented and stored across several nodes
- 2 Coordinator: initiate join on all nodes  $R_i$  (join nodes) and  $S_j$  (data nodes) ( $i = 1 \dots n, j = 1 \dots m$ )
- 3 Scan Phase: parallel on each  $S$ -node:  
read and transfer  $s_j$  to each  $R_i$
- 4 Join-Phase: parallel on each  $R$ -node with partition  $r_i$ :
  - ▶  $s := \bigcup s_j$
  - ▶ Compute  $t_i := r_i \bowtie s$
  - ▶ send  $t_i$  to coordinator
- 5 Coordinator: receive and merge all  $t_i$  (union)

# Dynamic Replication /2



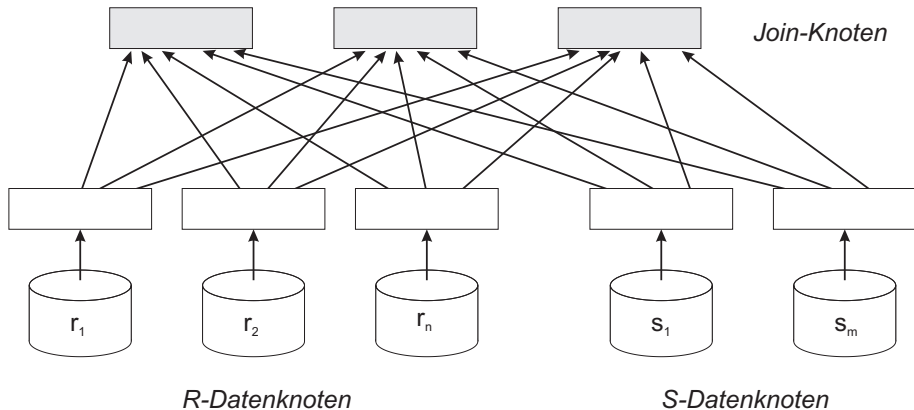
# Dynamic Partitioning

- 1 Coordinator: initiate joins on all  $R$ ,  $S$  and (possibly separate) join nodes
- 2 Scan Phase
  - ▶ parallel on each  $R$ -node:  
read and transfer each tuple of  $r_i$  to **responsible** join node
  - ▶ parallel on each  $S$ -node:  
read and transfer each tuple of  $s_j$  to **responsible** join node
- 3 Responsible node is computed by hash or range function → avoid or handle skew

# Dynamic Partitioning /2

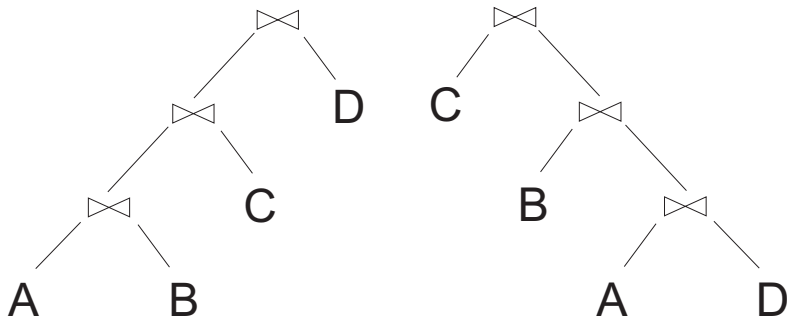
- ③ Join Phase: parallel on each join node  $k(k = 1 \dots p)$ 
  - ▶  $r'_k := \bigcup r_{ik}$  (set of  $r$ -tuples received at node  $k$ )
  - ▶  $s'_k := \bigcup s_{ik}$  (set of  $r$ -tuples received at node  $k$ )
  - ▶ compute  $t_k := r'_k \bowtie s'_k$
  - ▶ transfer  $t_k$  to coordinator
- ④ Coordinator: receive and merge  $t_k$

# Dynamische Partitionierung /3



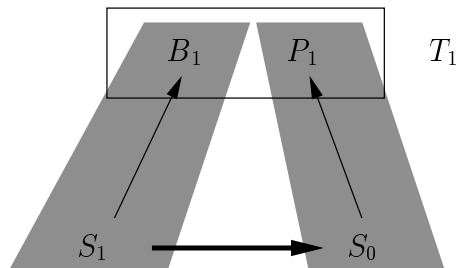
# Multi-Way-Joins

- Variants considered: left and right-deep trees



# Multi-Way-Joins /2

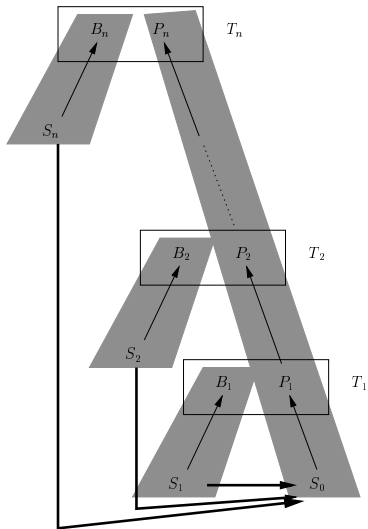
- Assumption: single join is executed as a Hash Join consisting of 2 phases
  - (1) **B**uild Phase: build hash table for first join relation
  - (2) **P**robe-Phase: check whether there are join partners from second relation
- Dependency: probe phase can only start after build phase has finished



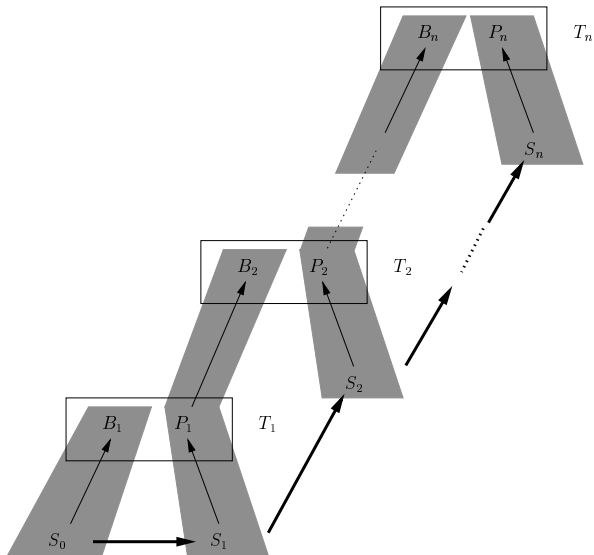
# Multi-Way-Joins /3

- Right Deep Trees
  - ▶ Perform build phases in parallel for all joins by independent execution
  - ▶ Perform probe phases in parallel by pipelined execution
- Left Deep Trees
  - ▶ Perform build phase of each join in parallel with probe phase of previous join by pipelined execution

# Join: Right Deep Tree



# Join: Left Deep Tree



# Multi-Way-Joins /4

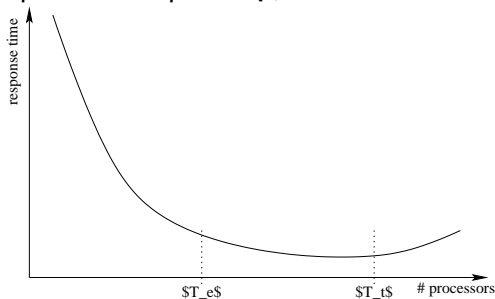
- Problem: multi-way-join processing with high requirements regarding memory, e.g. for hash tables
- Solution:
  - ▶ Scheduling of joins, e.g. breaking up deep trees to process only part of the join that fits in memory
  - ▶ Static or dynamic decision about segmentation

# Processor Allocation

- Assignment of operations (for now, join operations) to processors
- *2-Phase Optimization*
  - 1 Find a optimal plan based on costs
  - 2 Assign execution order and processor allocation
- *1-Phase Optimization*
  - ▶ perform both tasks as one step (part of query optimization)
- Relevant aspects
  - ▶ Dependencies between operations: e.g. consider join order to avoid waiting
  - ▶ Number of processors per join: increasing number does not yield linear improvement of processing time because of initialisation and merging, which can not be parallelized

# Processor Allocation /2

- Thresholds for number of processors: minimal response time point  $T_t$ , Execution efficiency point  $T_e$



- Execution efficiency for  $n$  processors

$$\text{Execution efficiency} = \frac{\text{Response time for 1 processor}}{n \cdot \text{Response time for } n \text{ processors}}$$

# Processor Allocation: Strategies

- Strategy 1: sequential execution of the joins with  $T_t$  processors
- Strategy 2: Time Efficiency Point
  - ▶ For each join, which can be started, allocate  $T$  processors

$$T = c \cdot T_e + (1 - c) \cdot T_t \quad 0 \leq c \leq 1$$

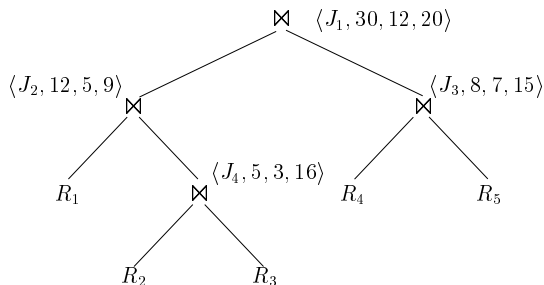
- ▶ Execution for  $N$  available processors and each join  $J$ 
  - 1  $T(J) = c \cdot T_e(J) + (1 - c) \cdot T_t(J)$
  - 2 **if**  $T(J) \leq N$  **then**
    - { allocate  $T(J)$  processors;  $N := N - T(J)$  }

# Processor Allocation: Strategies /2

- Strategy 3: Synchronous Top Down Allocation
  - ▶ Presumption: costs for each join are known (costs for execution of query plan subtree)
  - ▶ Processing: Top-down traversal of the join tree and allocation of processors
    - 1 Allocate  $\min\{N, T_t(\text{root})\}$  processors for the join tree root: minimal response time for last join;
    - 2 For join  $J$  is  $N_J$  the number of available processors for that join
      - if  $J$  has only one child  $J_1$ , allocate  $\min\{N_J, T_t(J_1)\}$  processors for  $J_1$
      - ff  $J$  has to childrene  $J_1$  and  $J_2$ , split  $N_J$  into two processor sets for  $J_1$  and  $J_2$  with size  $\sim$  costs
    - 3 Replace  $J$  by  $J_1$  and  $J_2$  and continue allocation recursively

# Processor-Allokation: Beispiel

- 20 processors,  $\langle J_i, \text{costs}, T_e, T_t \rangle$



- Sequential Execution:  $J_4: 16 \rightarrow J_3: 15 \rightarrow J_2: 9 \rightarrow J_1: 20$
- Time-Efficiency Point with  $c = 0.5$ :  $(J_4: 9, J_3: 11) \rightarrow J_2: 7 \rightarrow J_1: 16$
- Synchronous Top Down Allocation:  $J_1: 20, J_2$ -Subtree: 12 ( $J_2: 9$ ),  $J_3: 8, J_4: 12$