PARALLEL DATABASE SYSTEMS

- uniprocessor technology has reached its limit
  - difficult to build machines powerful enough to meet the CPU and I/O demand of DBMS serving large number of users
- a DBMS implemented on a multiprocessor
- attempts to achieve high performance through parallelism

PDBS vs DDBS

- DDBS share data stored in computers that are geographically distributed
- processors of a PDBS are often tightly coupled using a fast interconnection network or bus
- number of sites in DDBS is small compared to the number of processors in PBDS, ie., higher degree of parallelism in PDBS
- sites/nodes of a DDBS are autonomous computers
  - do not share memory, disks
  - run under independent OS and DBMS
- processors of a PDBS are not autonomous computers
  - share memory, disk
  - controlled by single OS and DBMS

HISTORY OF PDBS

- special purpose database machines (late 1970s, early 1980s)

- offload DBMS functions onto special processors

  - intelligent storage devices or database filters
  - distribution of functions

- poor cost-performance ratio
  - high costs of customized hardware
  - limited extensibility

WHAT'S DIFFERENT IN LATE 80s

- technology advancement
  - employs cheap commodity disks, processors and memories
  - software-oriented solutions
- large scale multiprocessors (over 1000 nodes) have been shipped!
- provide more total power at a lower cost
- modular architecture allows incremental growth
- widespread adoption of relational model
- relational queries are suited to parallel execution
  - consist of uniform operations applied to uniform streams of data
  - each operator produces a new relation, so operators can be composed into highly parallel dataflow graphs

TYPES OF PARALLELISM

- partitioned parallelism
  - input data is partitioned among multiple processors and memories
  - operator can be split into many independent operators each working on a part of the data

- pipelined parallelism
  - output of one operator is consumed as input of another operator

GOALS & METRICS

- speedup
  - elapsed time on a single processor/elapsed time on N processors
  - problem size is constant, and grows the system
  - *linear* speedup
    - twice as much hardware can perform the task in half the elapsed time
- scaleup
  - elapsed time on small problem/elapsed time on large problem
  - measures the ability of the system to grow both the system and the problem
  - *linear* scaleup
    - twice as much hardware can perform twice as large a task in the same elapsed time

BARRIERS

- startup
  - time needed to start a parallel operation
  - can dominate actual execution time with 1000s of processors

- interference
  - the slowdown each new process imposes on others when accessing shared resources

- skew
  - service time of a job is the service time of the slowest step of the job

ARCHITECTURE

- ideal machine: infinitely fast CPU with an infinite memory with infinite bandwidth

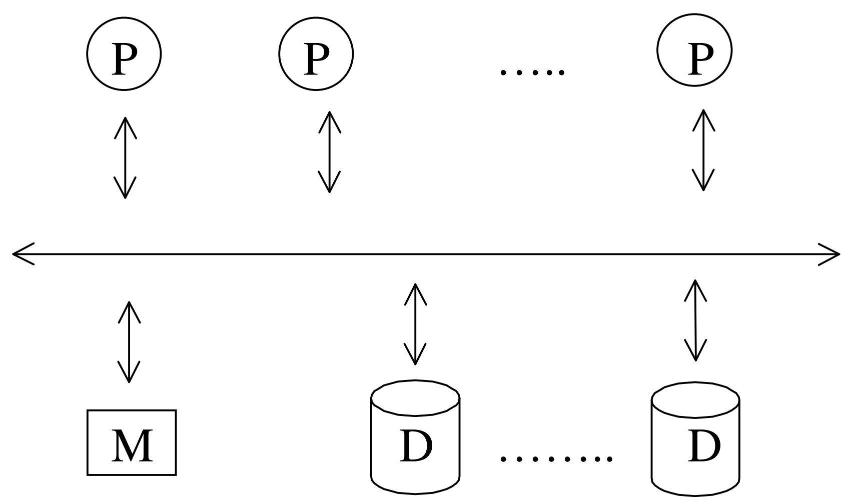
- Challenge – build such a machine out of large number of processors of finite speed

- Simple taxonomy for the spectrum of designs:
  - shared-memory (shared-everything)
  - shared-nothing
  - shared-disk

SHARED-MEMORY

- self-balancing
- limitations to scaling
  - bandwidth of interconnection network must equal the sum of the processors and disks
  - interference on shared resources (eg. lock tables, buffer manager)
  - design predicated on high cache hit rates
- affinity scheduling is a form of data partitioning

Shared-memory Architecture

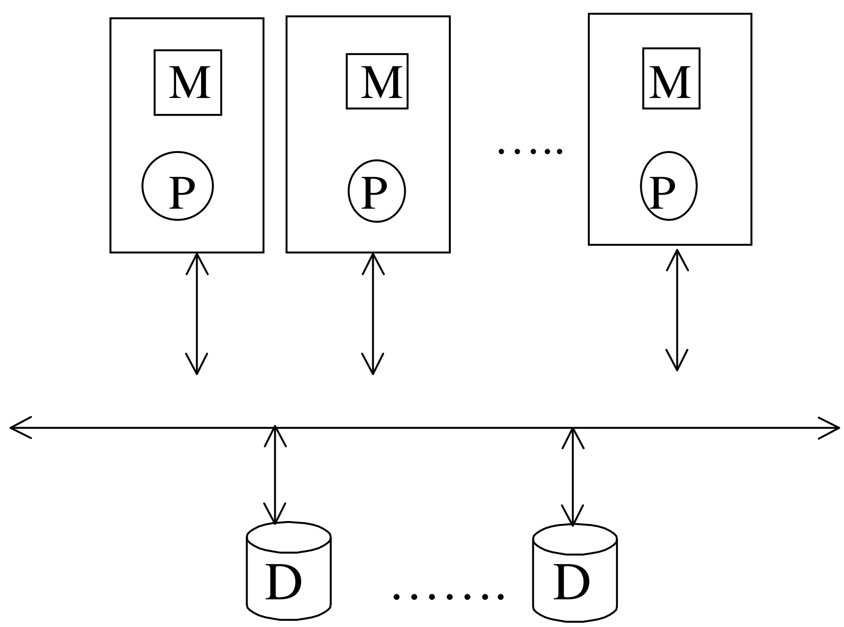


Examples: symmetric multiprocessors (Sequent, Encore) and some mainframes (IBM 3090, Bull's DPS8)

SHARED-DISK

- limitations to scaling
  - complex software mechanisms required to coordinate access to shared data
  - solution is to give data processor affinity; processors access remove data via messages

Shared-disk Architecture

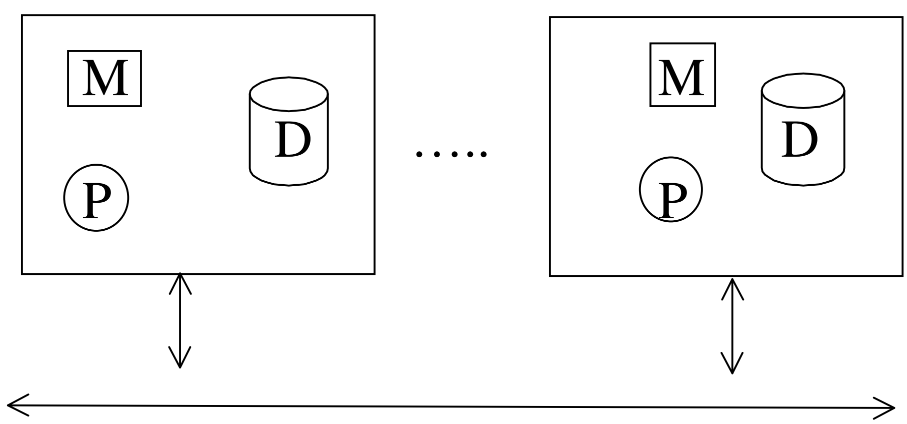


Examples: DEC's VAXcluster, IBM's IMS/VS Data Sharing

SHARED-NOTHING

- minimzes data moved through the network – only questions and answers
- actual interconnection network varies
  - trees, hyercubes, rings, switches
- potential to scale to thousands of nodes

Shared-Nothing Architecture



Examples: Teradate's DBC, Tandem, Intel's Paragon, NCR's 3600 and 3700, SP2, AP3000.

KEY TECHNIQUES TO PARALLELISM

- parallelism is an unanticipated benefit of the relational model

- relational query
  - relational operators applied to data

- 3 key techniques
  - data partitioning of relations across multiple disks
  - pipelining of tuples between operators
  - partitioned execution of operators across nodes

DATA PARTITIONING

- partitioning a relation involves distributing its tuples across several disks
- similar to fragmentation in DDBS
- provides high I/O bandwidth without any specialized hardware
- three basic partitioning strategies:
  - round-robin, hash and range

ROUND-ROBIN PARTITIONING

- excellent for sequential scans of full relations

- poor for associative access
  - find student with matrix no: ABC

HASH PARTITIONING

- excellent for exact match selection operations

- each tuple mapped to a disk by hashing

RANGE PARTITIONING

- contiguous attribute ranges mapped to same disk

- excellent for selections involving range predicates

PARALLELIZING RELATIONAL OPERATORS

- Objective
  - Use existing operator implementations without modification

- Only 3 mechanisms are needed
  - Operator replication, merge operator, split operator

- Result is a parallel DBMS capable of providing linear speedup and scaleup!

OPERATOR REPLICATION

- Assume relation ACCOUNTS is partitioned as follows:
  - p1 stores tuples whose acc# < 1000
  - p2 stores tuples whose acc# is in the range 1000 to 8000
  - p3 stores tuples whose acc# > 8000

- **Query Q1**: select name, balance from ACCOUNTS where balance > 10K
- Q1 will be executed in parallel on p1, p2, and p3

- **Query Q2**: select name, balance from ACCOUNTS where ACC# = 339
- Q2 will only be sent to p1 for execution
- p2 and p3 are free to execute other queries
- operator replication mechanism scales arbitrarily subject to the cost of starting an operator on each participating processor

MERGE OPERATOR

- merges output streams from two or more producers

- insert into TCUST
  select name, balance
  from ACCOUNTS
  where balance > 10K

SPLIT OPERATOR

- splits a tuple stream into two or more streams by applying a function to each tuple in the input stream

- split function: ACC# mod 4

USE OF SPLIT OPERATOR

- redistribution of result relations

- partitioned execution of complex operations, eg. join

  select *
  from A, B
  where A.x = B.y

BASIC PARALLEL JOIN ALGO

- comprises three phases

  - decompose the joins into *n* tasks
    - tasks themselves are joins of subrelations of the two base relations
  - assign those tasks to *p* processors
  - assemble the results as the join result

NESTED-LOOPS JOIN

- consider the join between R and S
- suppose R and S are partitioned across all the nodes
- nested loops join works as follows:
  - each node broadcast its fragment of S to all processors
  - each node joins its local R portion with the entire S relation

SORT-MERGE JOIN

- sort in parallel
  - redistribute records of R to the *p* nodes
    - find p-1 values from R. Denote them as r1, r2, r3, …, r(p-1)
    - records < r1 are allocated to p1, records between r1 and r2 are allocated to p2, and so on
    - each node sorts its local portion
  - redistribute S in the same manner, that is using the same range as R
- merge-join in parallel
  - each node merge-join its portion of R and S

HASH-BASED JOIN

- most widely studied
- many different variations
- make use of two splitting vectors (hash ftn's)
- during partitioning
  - each node splits its portion into n buckets according to the partitioning split vector (hash ftn 1)
  - the buckets are stored back to the disk at the node
- during the joining phase
  - each node reads bucket j of R and distributes records to the respective nodes using the join split vector (hash ftn 2)
  - upon receiving the records, the node build the hash table for the received data
  - redistribute bucket j of S
  - upon receiving records of S, use them to probe for matches in the hash table

DATA SKEW

- 4 types
  - tuple placement skew
    - initial distribution of record varies between partitions
  - selectivity skew
    - selectivity of selection predicates varies between nodes
  - redistribution skew
    - mismatch between distribution of join key values in a relation and the distribution expected by split vector
  - join product skew
    - join selectivity at each node differs

EFFECTIVENESS OF PARALLEL ALGO

- load imbalance
- performance determined by node that completes last
- need some form of load-balancing mechanism

PARALLEL JOIN ALGORITHMS: ISSUES & SOLUTIONS

- based on existing solutions
- three (logical) phases in parallel join algorithms
  - task generation
  - task allocation/load-balancing
  - task execution

TASK GENREATION

- a join is split into a set of tasks
  - each task is essentially a join of subrelations of the base relations
  - union of the results of all tasks should be the same as the original join

- four issues
  - decomposition of base relations
  - number of tasks to be generated
  - formation of tasks after decomposition
  - what statistics to maintain

DECOMPOSITION

- basic criterion
  - result of subtasks = result of original join
- 3 methods
  - most commonly used (no replication) but may not be effective for highly skewed data
  - each record participates in one task
  - tasks are generated by partitioning the relations into buckets
  - only corresponding buckets need to be joined
  - hash partitioning, range partitioning

DECOMPOSITON (Cont)

- Fragmentation and replication
  - larger relation is split into partitions
  - smaller relation is replicated
  - each task is the join of a bucket and the entirely of the smaller relation
  - e.g. nested-block join
- full replication
  - both relations are replicated
  - base relations are split into m and n buckets respectively
  - m * n tasks

NUMBER OF TASKS

- what is the optimal number of tasks?
- too many tasks may lead to poor performance since each task operates on only a few records
- too few tasks limit the flexibility of load balancing
- 4 methods (Given p processors)
  - generate p tasks --- one per node
  - generate more than p tasks
    - allocate to balance the load based on some criterion
    - split larger tasks to smaller ones, then allocate to balance some criterion
    - merge smaller ones, then allocate

TASK FORMATION

- whether the tasks are *physical* or *logical*

- physical tasks exists

- logical tasks yet to be formed
  - more useful for load balancing
  - lower cost (transmission and I/O)

STATISTICS MAINTAINED

- what statistics to maintain during task generation?
- used by task allocation/load-balancing phase to determine how tasks should be allocated
- perfect information
  - relation size, result size
- group records into equivalence classes
  - maintain number of distinct classes
  - for each class, the number of distinct attributes
  - for each class the number of records from the relations
  - allow estimation of result size and execution time

TASK ALLOCATION

- at the end of partitioning phase, we have a pool of tasks to be allocated to nodes for concurrent processing
- two strategies
  - allocate without load-balancing
    - when tasks are generated, they are sent to the appropriate nodes (preassigned)
    - bad for skew data

TASK ALLOCATION (cont)

- Load metrics
  - what criterion?
    - cardinality
      - ♦ balance the nodes based on the size of the tasks allocated
      - ♦ assume that the execution time is proportional to the size
    - estimated execution t time
      - ♦ time to process the tasks are estimated using cost models
      - ♦ balance completion time across all nodes
      - ♦ good if time can be accurately determined

TASK ALLOCATION (cont)

- with load balancing
  - tasks are allocated so that load across all nodes are approximately the same
    - tasks are sorted according to some criterion
    - allocated to nodes in descending order in a greedy manner (the next task to be allocated is sent to the node that is expected to finish first)

TASK ALLOCATION (cont)

- Allocation strategy
  - how each task is to be allocated
    - adaptively
      - ♦ demand-driven
      - ♦ each node assigned and process a task a time
      - ♦ acquire the next task only when the current task is completed
    - statically
      - ♦ allocated prior to the execution of the tasks
      - ♦ each node knows exactly how many and which tasks to process
      - ♦ most widely used

TASK EXECUTION

- each node independently perform the join of tasks allocated to it
- two concerns
  - workload
    - no redistribution once execution begins
      - ♦ if the system is unbalanced, too bad
    - redistribution if system is unbalanced
      - ♦ need a task redistribution phase to move tasks (subtasks) from overloaded node to underloaded nodes
      - ♦ must figure out who is the donor and who is idling
- local join methods – any uniprocessor join algorithms
  - nested loops
  - sort-merge
  - hash join

STATIC LOAD-BALANCED ALGO

- balance the load of nodes when the tasks are allocated

- once a task is initiated for execution, no migration of task (subtask) from one node to another

AN EXAMPLE

- consider join of R and S
- parallel system has p nodes
- assume that R and S are declustered across all nodes (using RR scheme)

EXAMPLE (cont)

- generate k tasks (k > p)
  - full fragmentation approach
  - each node is preassigned a set of tasks
  - each node reads its portion of R, hashes the records into k partitions and transmits the partitions to the respective nodes
  - each node receives its partitions from other nodes, and write them to its disk
  - repeat for S
- at the end, we have k physical tasks
- during partitioning, each node also collects statistics for the partitions assigned to it
  - sizes of the partitions
- a node designated as coordinator collects all the information
  - estimate the result size, and execution time
  - estimate the average completion time and inform all nodes about it

EXAMPLE (cont)

- based on the *average* completion time, each node finds the smallest number of tasks such that the completion time of these tasks will not exceed the average completion time
- a node is overloaded if it still has task remaining
- each node reports to coordinator the difference between the load and the average load, and the excess tasks
- coordinator reallocates the excess tasks to underloaded nodes
- after redistribution of workload, each processor independently processes its tasks

STATIC TECHNIQUE

- problems
  - extra communication and I/O costs
  - granularity of load-balancing is an entire tasks
  - bad for highly skewed data

- solutions
  - form logical tasks
  - split expensive tasks to smaller subtasks

DYNAMIC LOAD-BALANCING

- useful when estimation is wrong
- redistribute tasks which may be partially executed from one node to another
- task generation and allocation phases are the same as that for static load-balancing techniques
- task execution phase is different

DYNAMIC LOAD-BAL (cont)

- task execution phase
  - each node maintains additional information about the tasks that is being processed
    - size of data remaining
    - size of result generated so far
  - when a node finishes ALL the tasks allocated to it, it will *steal* from other nodes
    - the overloaded node and the amount of load to be transferred are then determined, and the transfer is realized by shipping data from donor to idle nodes

DYNAMIC LOAD-BAL (cont)

- process of transferring load
  - idle node sends a load-request message to coordinator to ask for more work
  - at coordinator node, requests are queued in a FCFS basis. Coordinator broadcasts a load-information message to all nodes
  - each node computes its current load and informs the coordinator
  - coordinator determines the donor and sends a transfer-load message to donor, after which it proceeds to serve the next request
  - donor determines the load to be transferred and sends the load to the idle node
  - once the load is determined, donor can proceed to compute its new load for another request

DYNAMIC LOAD-BAL (cont)

- process or transferring loads between a busy and an idle node is repeated until the minimum time has been achieved (or no more tasks to transmit)
- what information and to what accuracy should each processor maintain to facilitate load-balancing?
- which node (donor or idle) should inititate the load-balancing process?
- which nodes should be chosen as task donors?
- what is the amount of load to be transferred?
- at all nodes, number of pages processed and resultant pages generated
- initiated by idle node

DYNAMIC LOAD-BAL (cont)

- determination of donor and amount depends on join method
  - any busy node can be donor, but the one with the heaviest load preferred
  - use the estimated completion time of a node (ECT)
- ECT = estimated time of unprocessed task + estimated completion time of current task
- heuristics to determine amount
  - transfer unprocessed task first, if any
  - amount of load transferred should be large enough to provide a gain in the completion time of the join operation
  - completion time of donor should still be larger than that of idle node after the transfer