Advanced Databases: Parallel Databases A.Poulovassilis

1 Parallel Database Architectures

Parallel database systems use **parallel processing** techniques to achieve faster DBMS performance and handle larger volumes of data than is possible with single-processor systems.

There are three major architectures for parallel database systems: **shared-memory**, **shared-disk**, and **shared-nothing**. In all three architectures, all processors have their own cache, and cache accesses are much faster than accesses to main memory:[1]

- Shared-memory: multiple CPUs access a common region of main memory and a common set of disks.
- Shared-disk: each node has its own CPU(s) and its own main memory, but all nodes access a shared set of disks.
- Shared-nothing: each node has its own CPU(s), own main memory and own set of disks.

The two primary measures of DBMS performance are *throughput* — the number of tasks that can be performed within a given time, and *response time* — the time it takes to complete a single task.

Processing of a large number of small tasks can be speeded up by processing many tasks in parallel. Processing of individual large tasks can be speeded up by processing sub-tasks in parallel.

Speed-up refers to the performance of tasks faster due to more processors being added. Linear speed-up means that a system with n times more processors can perform the same task n times faster.

Scale-up refers to the performance of larger tasks due to more processors being added. Linear scale-up means that a system with n times more processors can perform a task that is n times larger in the same time.

[

Aside 1:

Adding more computing resources in a shared-memory or shared-disk architecture is called "scaling up" or "vertical scaling" (not to be confused with vertical data partitioning). Adding more

[1] Shared-memory and shared-disk architectures are found in SMP (Symmetric Multi-Processor) machines. In shared-memory systems, each processor maintains its own cache memory and the main memory is shared between all of them. Cache coherency needs to be maintained across all the processors in the face of updates in main memory.

Shared-disk architectures are found in NAS (network attached storage) architectures ("disk farms") and SANs (storage area networks). Cache consistency needs to be maintained as updates are being made to data pages by different processors.

Shared-nothing architectures are found in MPP (Massively Parallel Processing) machines that may comprise hundreds or thousands of processors.

computing resources in a shared-nothing architecture is called "scaling out" or "horizontal scaling" (again, not to be confused with horizontal data partitioning).

Horizontal scaling is the basis of *cloud computing*, in which computing resources are provisioned as necessary in order to meet increasing demand.

Aside 2: Parallel DBs vs DDBs

A key difference between parallel database systems and distributed database (DDB) systems is that the processors of a parallel database system are linked by high-speed interconnections within a single computer or at a single geographical site, and queries/transactions are submitted to a single database server.

The aim of a parallel database system is to utilise multiple processors in order to speed up query and transaction processing; whereas the aim of a DDB system is to manage multiple databases stored at different sites of a computer network.

In DDB systems, a significant cost is therefore the comunications cost incurred when transmitting data between the DDB servers over the network, whereas communications costs are much lower within a parallel database system.

If is possible of course for multiple parallel database instances to be connected into an overall distributed database system.

]

A number of factors can affect speed-up and scale-up in parallel database architectures:

- start-up costs for initiating multiple parallel processes;
- assembly costs associated with combining the results of parallel evaluation;
- interference between multiple processes for shared system resources;
- communication costs between processes.

The **shared-memory** architecture benefits from fast communication between processors through their shared memory and high-speed interconnection, and also ease of load-balancing. However, if there is a memory hardware fault all the processors will be affected. Also, it may suffer from the problem of interference: as more processors are added, existing processors are slowed down because of contention for memory accesses.

The **shared-disk** architecture does not suffer from this problem, and also memory faults are isolated from other processors. Load-balancing can again be easily achieved, through the shared access to the disks. However, this architecture may suffer from contention for disk accesses with large numbers of processors (e.g. beyond 100). Also, its communication costs are higher than the shared-memory architecture since the processors cannot communicate through a shared memory, but need to pass messages to each other along the interconnection network.

The **shared-nothing** architecture does not suffer from problems of contention for memory accesses or disk accesses, though it has higher communication costs than the other two architectures. Due to the lack of contention problems, it has the potential to achieve linear speed-up and scale-up. It also has higher availability as both memory and disk faults can be isolated from other processors.

However, load balancing may be hard to achieve as the data needs to be partitioned effectively between the disks accessed by the different processors.

There are also **hybrid architectures** that combine in different ways these three fundamental architectures, aiming to combine the simplicity and efficiency of shared-memory architectures with the higher extensibility and availability of shared-disk and shared-nothing architectures. For example:

- *clusters* of homogeneous off-the-shelf components can support a shared-memory architecture within each cluster; and the clusters can be interconnected through a high-speed LAN into a shared-disk or shared-nothing architecture. This combines the performance benefits of shared-memory within each cluster with the extensibility benefits of shared-disk or shared-nothing across clusters.

A common way of utilising the multiple processors and disks available in a shared-nothing architecture is by **horizontally partitioning** relations across the disks in the system in order to allow parallel access to, and processing of, the data. Partitioning can be applied to the intermediate relations being produced during the evaluation of a query, as well to as the original stored relations; the aim is to achieve even load-balancing between processors and linear scale-up/speed-up performance characteristics.

(Recall that horizontal partitioning means that the relation is split into groups of rows. Vertical partitioning means that the relation is split into groups of columns. Data partitioning is also known as "sharding".)

A query accessing a horizontally partitioned relation can be composed of multiple *subqueries*, one for each fragment of the relation. These subqueries can be processed in parallel in a shorter time than the original query.

This is known as **intra-query parallelism** i.e. parallelising the execution of one query. We will look more closely at this in these notes.

(**Inter-query parallelism** is also present in parallel databases i.e. executing several different queries concurrently.)

A key issue with intra-query parallelism in shared-nothing architectures is what is the best way to partition the tuples of each relation across the available disks. Three major approaches are used:

(i) round robin partitioning: tuples are placed on the disks in a circular fashion;

(ii) hash partitioning: tuples are placed on the disks according to some hash function applied to one or more of their attributes;

(iii) range partitioning: tuples are placed on the disks according to the sub-range in which the value of an attribute, or set of attributes, falls.

The advantage of (i) is an even distribution of data on each disk. This is good for load balancing a scan of the entire relation. However, it is not good for exact-match queries since all disks will need to be accessed whereas in fact only one a subset of them is likely to contain the relevant tuples.

The advantage of (ii) is that exact-match queries on the partitioning attribute(s) can be directed to the relevant disk. However, (ii) is not good for supporting range queries.

Approach (iii) is good for range queries on the partitioning attribute(s) because only the disks that are known to overlap with the required range of values need be accessed.

However, one potential problem with (iii) is that data may not be evenly allocated across the disks — this is known as **data skew**.

One solution to this problem is to maintain a **histogram** for the partioning attribute(s) i.e. divide the domain into a number of ranges and keep a count of the number of rows that fall within each range as the relation is updated. This histogram can be used to determine the subrange of values allocated to each disk (rather than have fixed ranges of equal length).

Shared-disk and shared-nothing architectures use similar protocols for currency control and atomic commitment of transactions as in distributed databases e.g. distributed 2PL, 2PC.

Replication of data across multiple processors and disks allows the system to carry on operating if a processor or disk fails.

We briefly discuss parallel query processing below.

2 Parallel implementation of the relational algebra

Note: The algorithms described below generalise to all three architectures. Just the mode of communication between the processors is different: via the interconnection network with shared-nothing — which is what is assumed in the below description; and via shared memory or shared disk with shared-memory/shared-disk.

The basic foundations of parallel query evaluation are the **splitting** and **merging** of parallel streams of data:

Data streams arising from different disks or processors provide the input for each operator in the query.

The results being produced as an operator is evaluated in parallel over a set of nodes need to be merged at one node in order to produce the overall result of the operator.

This result set may then be split again in order to parallelize subsequent processing in the query.

2.1 sort

Given the frequency with which sorting is required in query processing, parallel sorting algorithms have been much studied.

A commonly used parallel sorting algorithm is the **parallel merge sort**:

This first sorts each fragment of the relation individually on its local disk e.g. using the **external merge sort** algorithm we have already looked at.

Groups of fragments are then shipped to one node per group, which merges the group of fragments into a larger sorted fragment. This process repeats until a single sorted fragment is produced at one of the nodes.

2.2 projection and selection

Selection and projection operators on a relation are usually performed together in one scan. This can be considered as a **reduction** operator.

Reducing a fragment at a single node can be done as in a centralised DBMS i.e. by a sequential scan or by utilising appropriate indexes if available.

If duplicates are permitted in the overall result, or if they are not permitted but are known to be impossible (e.g. if the result contains all the key attributes of the relation), then the fragments can just be shipped to one node to be merged.

If duplicates may arise in the overall result and they need to be eliminated, then a variant of the parallel merge sort algorithm described in Section 2.1 can be used to sort the result while at the same time eliminating all duplicates encountered.

2.3 join

(i) $R \bowtie S$ using Parallel Nested Loops or Index Nested Loops Join

First the 'outer' relation has to be chosen. In particular, if S has an appropriate index on the join attribute(s) then R should be the outer relation.

All the fragments of R are then shipped to all nodes. So each node i now has a whole copy of R as well as its own fragment of S, S_i.

The local joins $R \bowtie S_i$ are performed in parallel on all the nodes, and the results are finally shipped to a chosen node for merging.

(ii) $R \bowtie S$ using Parallel Sort-Merge Join (for natural/equijoins only)

The first phase of this involves sorting R and S on the join attribute(s). These sorts can be performed using the parallel merge sort operation described in Section 2.1.

The sorted relations are then partitioned across the nodes using range partitioning with the same subranges on the join attribute(s) for both relations.

The local joins of each pair of sorted fragments $R_i \bowtie S_i$ are performed in parallel, and the results are finally shipped to a chosen node for merging.

(iii) $R \bowtie S$ using Parallel Hash Join (for natural/equijoins only)

Each bucket of R and S is logically assigned to one node.

The first hashing phase, using the first hash function h_1, is undertaken in parallel on the all nodes. Each tuple t from R or S is shipped to node i if the bucket assigned to it by h_1 is the i^{th} bucket.

The next phase is also undertaken in parallel on all nodes. On each node i, a hash table is created from the local fragment of R, R_i, using another hash function h_2. The local fragment of S, S_i, is then scanned and h_2 is used to probe the hash table for matching records of R_i for each record of S_i.

The results produced at each node are shipped to a chosen node for merging.

2.4 Parallel Query Optimisation

The above parallel versions of the relational algebra operators have different costs compared to their sequential counterparts, and it is these costs that need to be considered during the generation and selection of parallel query plans:

- The costs of partitioning and merging the data now need to be taken into account.
- If the data distribution is skewed, this will have an impact on the overall time taken to complete the evaluation of an operator, so that too needs to be taken into account.
- There is now the possibility of pipelining the results being produced by one operator into the evaluation of another operator that is executing at the same time on a different node.

  For example, consider this left-deep join tree where all the joins are nested loops joins:

  $$((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$$

  The tuples being produced by $R_1 \bowtie R_2$ can be used to 'probe' R_3 and the resulting tuples can be used to probe R_4, thus setting up a pipeline of three concurrently executing join operators on three different nodes.
- There is now the possibility of executing different operators of the query concurrently on different nodes.

  For example, with this bushy join tree:

  $$((R_1 \bowtie R_2) \bowtie (R3 \bowtie R_4))$$

  the join of R_1 with R_2 and the join of R_3 with R_4 can be executed concurrently. The partial results of these joins can also be pipelined into their parent join operator.

In uniprocessor systems only left-deep join orders are usually considered (and this still gives $n!$ possible join orders for a join of n relations).

In multi-processor systems, other join orders can result in more parallelisation e.g. bushy trees, as illustrated above. Even if such a plan is more costly in terms of the number of I/O operations performed, it may execute more quickly than a left-deep plan due to the increased parallelism it affords.

So, in general, the number of candidate query plans is much greater in parallel database systems and more heuristics need to be employed to limit the search space of query plans. For example, one possible approach is for the query optimiser to first find the best plan for sequential evaluation of the query; and then find the fastest parallel execution of that plan.

Reading

For interest: "Parallel database systems: the future of high performance database systems" (up to the top of page 94), David DeWitt and Jim Gray, Comm. ACM, 35(6), 1992, pp 85-98. At `http://dl.acm.org/citation.cfm?id=129894`