Overview of Parallel Architectures for Databases

B. BERGSTEN*, M. COUPRIE† AND P. VALDURIEZ‡

*Bull Systems & Products, Rue Jean Jaures, BP68, 78340 Les Clayes-sous-Bois, France
†ESIEE, 2, boulevard Blaise Pascal, BP99, 93162 Noisy-le-grand Cedex, France
‡INRIA & Bull, Domaine de Voluceau, BP105, 78153 Le Chesnay Cedex, France

We present and compare hardware and system architectures for databases that take advantage of parallelism. First, we state the problem and identify the main comparison criterions (price, performance, extensibility, data availability). Then, we review the major architecture classes (shared nothing, shared everything, shared disks, hybrid) and discuss their advantages and drawbacks for different kinds of workloads.

Received June 1993; revised September 1993

1. INTRODUCTION

In the information management system market, the demand for more processing power and storage capacity is still growing exponentially. Parallelism is a way to cope with this problem, but different approaches are possible and choices must be made. What is the best architecture? For what kind of applications? What are the advantages and drawbacks of each known solution?

Many papers have been devoted to this topic in recent years, ranging from quantitative, detailed, model-based studies, to more qualitative and synthetic approaches.

Most often, modelling is used to study particular modules in the system, such as the concurrency control and coherency control, which are critical for performance, under a wide range of workloads. An overview of modelling studies on these topics can be found in Rahm (1991).

Few studies, such as Bhide (1988), tried to use this approach to compare different architectures. Although these studies are useful, they consider only a small number of system configurations and workloads, and concentrate only on performance criterions (throughput, response time, etc.).

Thus, there is a need for a more synthetic, qualitative approach to architecture comparison. In this paper, we review the major architecture classes (shared nothing, shared everything, shared disks, hybrid) and discuss their advantages and drawbacks for different kinds of workloads.

Before we go on more deeply into detail, let us define the main criteria to compare the different architectures and let us show what benefits parallelism can offer with respect to these criteria. In fact, these criteria summarize the claims of any DataBase Management System (DBMS) customer: competitive *price*, high *performance*, *extensibility* and data *availability*.

- *Price.* An important advantage of parallel systems is to allow the use of inexpensive standard elements, collaborating together and offering a performance level as good as sophisticated, expensive systems (traditional mainframes).
- *Performance.* Two metrics are commonly used to measure database systems performance: throughput (in transactions per second, t.p.s.) and response time. Parallelism increases the global throughput when several queries are processed concurrently (*inter*-query parallelism) and decreases the response time if several processing elements cooperate to process the same query (*intra*-query parallelism). Nevertheless, intra-query parallelism implies some overhead (initialization, synchronization, communication) and if one tries to enhance response time using intra-query parallelism, this will probably result in degrading the global throughput. Thus, throughput and response time cannot be tuned independently.
- *Extensibility.* Two metrics are commonly used to measure parallel system extensibility: *speedup* and *scaleup*. Speedup is the performance gain obtained by multiplying processing elements. Scaleup measures the growth of the database size allowed by the addition of more processing elements, while maintaining a constant performance level. If multiplying the number of processing elements always multiplies the performance by the same factor, the speedup is said to be linear. Similarly, if multiplying the number of processing elements always multiplies the allowed database size (with equal performance) by the same factor, the scaleup is said to be linear.

  In a parallel system, it should be easy to cope with the growth of data size and/or processing power needs. Ideally, it should be sufficient to add new identical processing elements, in proportion with the new requirements. Actually, several factors play against speedup and scaleup linearity, mainly: parallelization overhead, communication and synchronization costs, bottleneck saturation, and poor load balancing.

- *Data availability.* Traditionally, data availability is limited by all kinds of failures (hardware or system) and by maintenance operations.

  Since a parallel system is composed of several identical elements, data replication among these elements may be exploited in order to increase data availability. For example, in case of a disk failure, another copy of the data should be available on the other disks. Of course, this is done at the expense of multiplying disks and a protocol must be used to ensure the coherence of multiple copies. This protocol consumes CPU and communication resources, and introduces some delays due to synchronization.

Now, let us consider the DBMS designer's point of view. What are the problems to solve in order to meet the customer's requirements?

- *Disk I/O throughput.* The main problem in traditional DBMS is the disk I/O bottleneck, due to the high disk access time, typically 10 000 times the main memory access time. The performance of many database applications is limited by the I/O access time. Such applications are called *I/O-bound.*

  Parallelism may help to increase the I/O throughput in several ways. The disks may be shared, physically or logically, by all processing elements (*shared disk*, *shared everything*) or distributed among the different processing elements (*shared nothing*). A different approach is to associate several disks with an 'intelligent' disk controller in charge of managing the data distribution and redundancy among the disks (*disk arrays*). Such a disk array can be seen from the outside as a very fast reliable disk.
- *CPU power.* In addition to simple and repetitive transactions that used to be the main workload of traditional DBMS, there is a growing proportion of more complex, longer, and sometimes *ad hoc* transactions. For example, *decision support* queries may need to sort and join very large relations. The use of *integrity constraints* also contributes to complexify some very basic operations, such as updates. Some of these complex queries may be more CPU-bound than I/O-bound. A parallel system offers a large spectrum of configurations, which allows to balance the number and the power of CPUs and disks in order to avoid resource contention.
- *Communication.* In order to cooperate, the processing elements must exchange various kinds of informations, such as: intermediate results, synchronization messages, concurrency control messages, etc. Such communication is a part of the parallelism overhead. Furthermore, the interconnection network throughput has to be sufficient to avoid a communication bottleneck. Caching techniques are generally used to lower the communication volume. Besides, the interconnection network often limits the scability of parallel systems.
- *Load balancing.* An homogeneous distribution of the load between the processing elements is a prime condition to achieve linear speedup and linear scaleup. Load balancing can be a more or less difficult problem, depending on the class of applications (debit/credit, decision support, etc.) and the architecture.

These are the four main points to consider when designing a parallel database system. The designer must also consider implementation issues such as design complexity, portability, which may depend on the chosen architecture.

In the next section, we examine the general framework of client/server architectures, which is commonly adopted in modern information systems. Section 3 analyses the different parallel database server architectures (*shared everything*, *shared nothing*, *shared disk*, *hybrid*), with their respective advantages and drawbacks. Section 4 concludes the overview of parallel architectures for databases.

2. CLIENT–SERVER ARCHITECTURES

Modern information systems are all based on a client–server organization. A client–server computational model implies a relationship between two processes in which one makes requests to the other. This organization allows a decomposition of functionality in complex systems, e.g. in database systems, where presentation services are usually supported by the client, and database accesses are managed by the server.

Typically, client processes are executed in workstations, while server processes are executed in powerful platforms. In many cases, the application is mostly written in a host language (C or COBOL) for the user interface part and for specific computations, and embeds database requests written in a database language (such as SQL or CODASYL).

The connection between the host language and the database language introduces a data type conversion overhead, that can be avoided by using a DataBase Programming Languages (DBPL). In this case, the application programmer uses a common language and data types for database access and programming which deals with both persistent and volatile data (Atkinson, 1987). In order to be efficient, the DBPL-based systems separate persistent and volatile data, either statically or dynamically. Thus, the 'programming' part is separated from the 'database access' part, even if this separation is invisible to the user.

An application may either trigger few complex database queries (*coarse grain* interaction—access to more 10 000 data items) or many simple database queries (*fine grain* interaction—access to less than 10 data items). Applications may be classified depending on their interaction grain:

- Decision support applications are coarse grained (a decision support application typically accesses a large

data volume in a non-predictable manner). The Wisconsin Benchmark (Gray, 1991) characterizes typically this class of applications.

- OLTP (On Line Transaction Processing) applications are medium grained. The TPC-A and TPC-B (Gray, 1991) characterize typically this class of applications.
- Deductive applications (deduction and integrity constraints) are medium grained (a deductive application dealing with very large data volumes may be considered as a decision support application). Today, no Benchmarks characterize this class correctly, mainly because this class is new and only partially supported by current Database Management Systems. However, TPC-C (TPC, 1992) is a step toward this direction, since key and referential integrity constraints have to be satisfied.
- The applications that 'navigate' in the database under the control of a program are fine grained (CAD, CAM, CASE-type applications). This class is typically characterized by the 001 (Gray, 1991) and 007 (Carey, 1993) Benchmarks.

Note that we just classified *applications*, not *systems*. In the following, we describe client–server architectures for fine grained applications and client–server architectures for coarse grained applications, and we discuss their suitability with the different application classes. We do not describe specifically architectures for medium grained applications since these applications are most of the time well supported by architectures designed for coarse grained applications.

2.1. Architectures for fine grained applications

This is the case where the application 'navigates' in the database. Each navigation step involves a short interaction with the DBMS. This approach is historically the oldest one (CODASYL systems), but has recently gained much popularity with object-oriented systems like Versant, ObjectStore, O_2. Typically, the application is written in a programming language (C/C++, Smalltalk, COBOL, Pascal, FORTRAN, etc.) and embeds the database access queries.

A fine grain application needs to be closely coupled with the DBMS in order to avoid high database access overhead (often compared with the impedance mismatch between electronic components). To achieve a reduction of the communication and data conversion costs, the DBMS must be split into a client DBMS, running with the application on the same workstation, and a server DBMS, running on the server machine (Figure 1). A data caching mechanism may thus be used to reduce client–server communication and allow a local navigation in the client machine. On a cache miss, the client sends a page or an object demand to the server. The server is thus reduced to a page (or object) server, consuming few CPU resources. Still, a high I/O throughput may be needed, depending on the number of connected clients. Disk I/O parallelism may be necessary in the server to achieve this throughput.

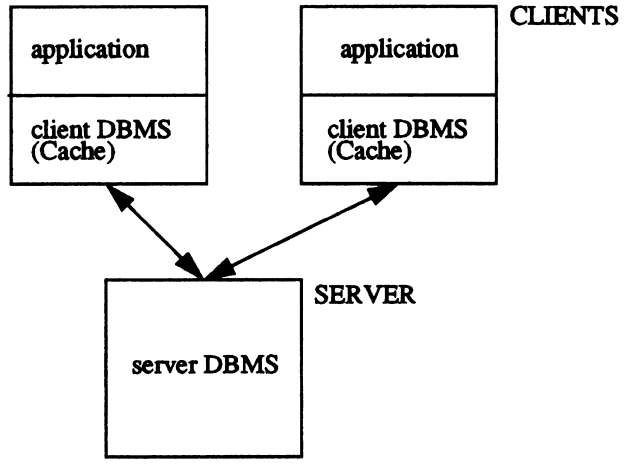

FIGURE 1. Client–server architecture for fine grained applications.

With this architecture, a multiple cache coherency protocol is needed, either to make sure that an update in a page is automatically propagated to all clients which cache it or to invalidate pages that are no longer up-to-date.

Note that, if we want to execute transactions in parallel (intra-transaction parallelism), we need a parallel client, not a parallel server. Besides, as clients usually run on different workstations, we have inter-transaction parallelism, without a parallel server. The client main memory should be large enough to hold the frequently accessed data (the so-called 'hot data'), in order to avoid swapping (from client's main memory to client's disk or to server).

A recent study (Franklin, 1992) investigates global memory management strategies for client–server architectures, that allow a better utilization of the clients' memories and reduce data replication in the system.

Advantages

- Reduced communication thanks to client cache.
- Use of the local client's processors (less costly than server's processors) to achieve inter-query parallelism.
- Offload the server of some DBMS work.
- Simple server.

Drawbacks

- Swapping if the client's memory is not large enough.
- Cost of the multiple copy coherency protocol.
- High communication volume if there is little data locality.
- Complex to optimize.

2.2. Architectures for coarse grained applications

The coarse grained applications (decision support applications) benefit from a parallel computation of the database queries, which reduces the response time. These

applications may be loosely coupled with the DBMS, since there are few interactions (Figure 2).

Some coarse grained queries cannot be expressed in some query languages. It is the case, for example, with SQL, which cannot express actions like conditionals, complex predicates or transitive closure. This lack in expressive power forces the application to gather *all* the potentially useful data, and then to do the computation locally (and sequentially).

This problem may be approached by three different and complementary ways:

- By *ADTs* (Abstract Data Types): some user-defined methods (programs), associated to data definition templates, may be loaded and executed (possibly in parallel) on the server.
- By *procedures* including several queries: the execution unit is no longer the query, but rather a set of queries. It is a traditional way to obtain good performance with debit/credit-type applications.
- By *supporting new operators* or adopting a more powerful query language, allowing recursion.

Advantages

- Little communication volume (only queries and final results) if most of the work can be done in the server.
- No need for a coherency control protocol between clients.

Drawbacks

- Client resources (CPU, disk) may be underloaded, while the server may be a bottleneck.

3. PARALLEL SERVER ARCHITECTURES

In this section, we describe and discuss the main classes of parallel server architectures: the systems sharing both disks and main memory (*shared everything*, *SE*), the systems with distributed memory but sharing, logically or physically, the I/O subsystem (*shared disk*, *SD*), the systems with distributed memory and disks (*shared nothing*, *SN*) and hybrid *SD–SE* systems. We will consider that an architecture with physically distributed disks can be classified as a shared disks architecture, if there is a system layer that hides the data distribution on disks to the higher level layers (*data sharing system*).

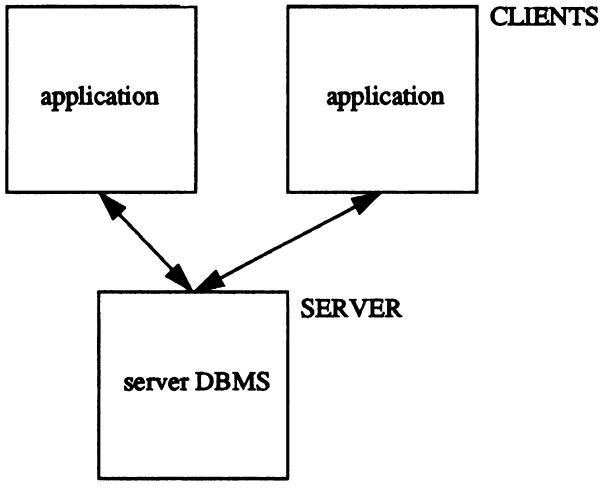

FIGURE 2. Client–server architecture for coarse grained applications.

3.1. SE architecture

With this architecture (Figure 3), all processors share in a symmetrical way all the main memory and all the disks. In general, the allocation of processors to processes is done automatically by the operating system. Thus, a parallel execution can be easily obtained by creating several processes (or threads).

Actually, each processor has a cache memory (typically 64 KB to 4 MB). The coherence between the different caches is often ensured by a mechanism based on some specific hardware. This mechanism is generally based on a shared bus that is continuously 'snooped' by all processor boards, to see if their cached data is required elsewhere.

This kind of architecture offers the best processor utilization because the load balancing is easily and efficiently handled by the run-time system, and because the cache memories provide fast access to the most frequently used data and code. Besides, the specific hardware that supports the cache coherency protocol is expensive, precisely because it is specific. It is likely that this cost will lower as multiprocessor UNIX servers become a market standard.

Inter-query parallelism is easily achieved by forking processes. Intra-query parallelism is still rather simple to obtain for read-only queries: one straightforward solution is to use classical parallel algorithms for costly operators in the query execution graph (like sort and join). Also, some administration procedures like index creation can be parallelized easily.

Advantages

- Simple for inter-query parallelism.
- Quite simple for intra-query parallelism.
- Good resource utilization: efficient automatic load-balancing.
- Inter-processor communication using the shared memory (very fast).

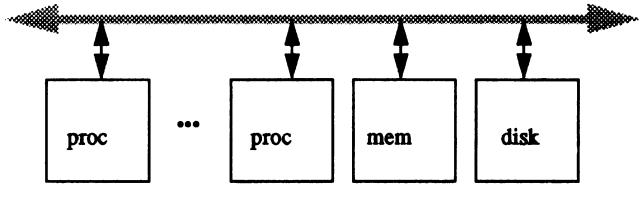

FIGURE 3. Shared everything.

Drawbacks

- Limited scalability (10–20 processors).
- Data availability cannot be easily enhanced.
- Quite expensive today, but will probably lower.

Systems

- Exploitation of inter-query parallelism: DB2, Oracle, etc., ported on classical multiprocessor mainframes (IBM 3090, Bull DPS8, etc.). Oracle, Ingres, Sybase, etc., on UNIX based multiprocessors (Sequent, Encore, etc.).
- Exploitation of intra-query parallelism: Research prototypes: XPRS (Berkeley) (Stonebraker, 1988), DBS3 (Bull-RAD/BDI) (Bergsten, 1991).

3.2. SD architecture

In a SD architecture (Figure 4), the main memory is distributed, but the disks may be accessed by any node (composed of one or several processors with some local memory) in a symmetrical manner. The distributed software system, or the application software, must ensure the coherency between the multiple copies of disk pages requested by several nodes. This problem is similar to the cache coherency control problem in SE systems. The major differences are that coherency control protocols for SD are implemented by software, using a communication link that is typically 10–100 times slower than in SE, and deals with objects (pages) having a coarser granularity.

Several studies have shown the benefit of tightly coupling the concurrency control and coherency control mechanisms, in order to reduce the overheads and the communication volume. See Rahm (1991) for a survey of such techniques.

SD architectures are interesting for database systems for two major reasons: (i) they allow better scalability and availability than SE and (ii) database management systems designed for centralized systems may evoluate to SD, while this is not true for SN. Oracle V4.2 has been ported on a parallel NCUBE with 64 nodes, sharing 128 disks, and is presently the fastest relational transactional system with more than 1000 TPC-B transactions per second.

The disks may be physically coupled to the nodes (like in SN), but they must be logically shared: every page on every disk must be accessible from any node with (roughly) the same access time. In general, accessing a page that is not stored on a local disk is not much longer than a local access, because a page transfer on the network is typically 10 times faster than a page I/O. It is thus possible to mix diskless nodes with disk-owning nodes, and new disks may be easily added to the system without reorganizing the database.

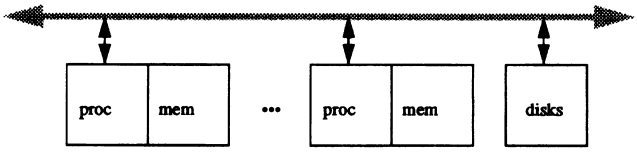

FIGURE 4. Shared disk.

With this architecture, the transaction code need not migrate where the data resides (like in SN). It is also possible to balance the load dynamically (according to the current node utilization), rather than statically (according to the data placement). Also, a transaction may be processed by a single node, as in traditional systems (porting a centralized DBMS on a SD system is feasible).

Unlike SN, data accessed in read mode may be replicated on several nodes by the cache mechanism, transparently to the user/programmer. This contributes to reducing 'hot spots' (frequently accessed data that become bottlenecks).

Surprisingly, little interest has been shown for intra-query parallelization in SD systems.

Advantages

- Good scalability (100 processors).
- Low cost because of the use of standard components.
- Possible migration of centralized systems to SD systems.
- Good load balancing (automatic replication of 'hot' data).
- Good availability and reliability.

Drawbacks

- Cost of the multiple copy coherency mechanism, especially if there are many updates.
- The interconnection network may be saturated by the page transfer traffic.

Systems

- Exploitation of inter-query parallelism: IBM: IMS/VS Data Sharing Product (Strickland, 1982), TPF (Scrutchin, 1987), Amoeba (Shoens, 1985); DEC: VAX DBMS, VAX Rdb/VMS (Kronenberg, 1986); NEC: DCS (Sekino, 1984); Oracle on NCUBE and on Vax Cluster.

3.3. SN architecture

This architecture has attracted a great deal of attention in the database research community, certainly because it is scalable to a large number of processors (some authors speak about 'massive parallelism') and, besides, it can be constructed from cheap components. The SN architecture appeared as a solution to the major problem of databases: the disk I/O parallelization. Each node of the SN system independently manages one or several disks (Figure 5). For example, to scan a relation (that is distributed on several nodes) in parallel, the query is simply divided into sub-queries that are executed on the different nodes holding parts of the relation. Techniques for parallelization of relational operations in SN are now well understood (Ozsu, 1990).

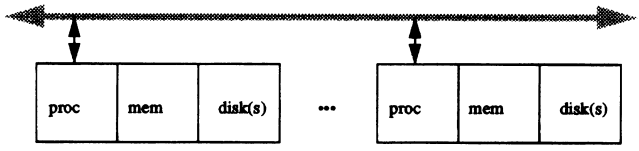

FIGURE 5. Shared nothing.

The major advantage of SN over SD is that the messages exchanged by the nodes contain only useful data. In SD, the pages (typically 1–8 KBytes) that are exchanged via the interconnection network may contain a proportion of data that will not be used. Thus, the network traffic is lower in SN and the scalability should be better.

On the other hand good load balancing is very difficult to obtain in a SN architecture. This is because load balancing is mainly obtained by the physical (and static) distribution of data among the nodes. In order to enhance load balancing, some researchers proposed to distribute data according to a measure of the 'relation heat' (based on the access frequency) and 'temperature' (based on size) (Copeland, 1988), but this raises the problem of applications with highly variable access pattern (some queries are triggered only in the evening, or at the end of the month, etc). In such cases, the relation heat is difficult to define, and it would be costly to reorganize the database according to heat variations. Note also that, if this architecture is physically very scalable, adding or suppressing nodes implies a reorganization of the database.

Unlike SD, if a data item is frequently accessed, the node where this data resides must sequentially manage these accesses. Thus, this node may become a bottleneck and degrade the system's performance. A solution to this problem should be to replicate this data on several nodes, but an explicit coherency protocol is needed (e.g. ROWA, Read One, Write All). The use of such techniques, together with the load balancing techniques, raise very difficult administration problems.

Last but not least, a major problem with SN is that all queries, even the simplest ones, must be parallelized. Let us consider a query that has to compare two data items I_1, I_2 that reside on two different nodes N_1, N_2. Then, the compiler must produce two code fragments. The first code fragment will be executed on N_1, and will send I_1 to the second node N_2. The second code fragment will be executed on N_2, to receive I_1 and compare it with I_2. Because of this necessary parallelization of all queries, centralized DBMSs cannot be ported on this kind of architecture.

Advantages

- Very good scalability (1000 processors).
- Cheap hardware.
- Good availability and reliability.

Drawbacks

- Hard to administrate.
- Parallelization overhead, even for simple queries.
- Load balancing is difficult.
- Centralized DBMSs cannot be ported.

Systems

- Exploitation of intra- and inter-query parallelism: TANDEM: NonStop SQL (Englert, 1989); Teradata: DBC/1012 (Teradata, 1983, 1985); MCC: Bubba (Boral, 1990); Univ. Wisconsin: Gamma (DeWitt, 1990); EDS ESPRIT project: EDBS (EDS, 1990).

3.4. Hybrid architecture (SD–SE)

We conclude from the previous sections that SE systems are best for load balancing, but their scalability is limited; SD systems offer better scalability and still good load balancing, but page transfers may saturate the network; and, finally, SN systems are best for scalability, but load balancing is very difficult. Thus, SD looks like a good compromise between scalability and load balancing. In order to reduce the number of nodes, and consequently, the page traffic, it is necessary to increase the power of each node. A possible solution is to have several SE systems as nodes of a SD system. The disks may be shared logically and distributed physically (Figure 6) or they may be physically shared (Figure 7).

As each node is a multiprocessor machine, it must be fed by a high speed disk system. A solution is to connect RAIDs (redundant array of inexpensive disks) to each node, or to each couple of nodes (Teradata uses this last option in order to increase fault tolerance). In a hybrid SD–SE system based on physically shared disks, one

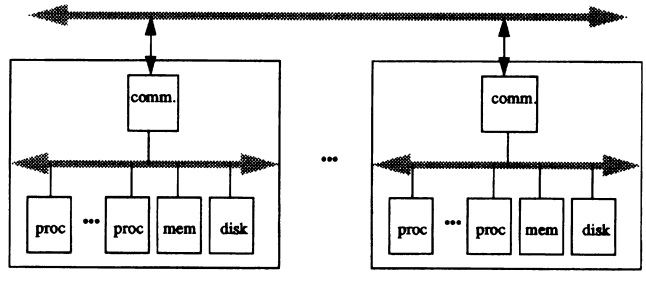

FIGURE 6. Hybrid architecture, with logical disk sharing.

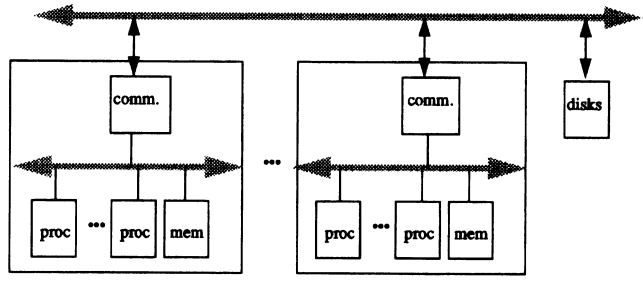

FIGURE 7. Hybrid architecture, with physical disk sharing.

may need an 'intelligent' controller to manage a set of RAIDs.

This hybrid architecture allows us to execute parallel queries in a single SE node. It is an advantage since it is easier to parallelize a query in SE than in a distributed system. Furthermore, the parallelism degree available on a single SE node may be sufficient for some applications.

For OLTP applications, logging may be done locally in each node, saving some communication.

The main disadvantages of SE (low scalability, data availability difficult to enhance) disappear with several nodes.

Such an architecture may be composed of standard parallel UNIX servers, interconnected by a standard network.

Advantages: same as SD, plus:

- Flexibility in the configuration (number of nodes, number of disks, CPUs per node) according to the application and database profile.
- For a fixed processing power, reduces the communication traffic on the network by reducing the number of nodes.
- The possible parallelization of a complex query on a single multiprocessor SE node (exploiting all the SE nodes to speed-up single queries is difficult).

Drawbacks:

- Cost of the multiple copy coherency mechanism.
- Price of the SE nodes.

4. CONCLUSIONS

As an alternative to conventional mainframe technology, standard and unexpensive components can be used to build high performance parallel systems. Furthermore, the parallel systems ease scalability and data availability.

Each multiprocessor parallel architecture (shared everything, shared disks, shared nothing, hybrid) has advantages and drawbacks, that we discussed in this paper.

A parallel DBMS is usually integrated in a more general client/server organization. A coarse grained interaction between the application and the DBMS allows a full exploitation of the parallel server power. A high level interface (through SQL, for example) favours such a coarse grained client/server interaction.

Two important factors let us conclude that DBMSs take most advantage of the parallel architecture technology:

- The existence of high level query languages (a single SQL query can be parallelized automatically, without modifying the application program).
- Parallelism may be obtained by data partitioning, which may be logical and/or physical. In a relational system, this partitioning is quite easy to achieve.

ACKNOWLEDGEMENTS

This work has been partially supported by the CEC under the Esprit project no. 7091, Pythagoras.

References

Atkinson, M. and Buneman, P. (1987) Types and persistence in database programming languages. *ACM Comp. Surv.*, **19**.

Bergsten, B., Couprie, M. and Valduriez, P. (1991) Prototyping DBS3, a shared-memory parallel database System. *Proc. Int. Conf. on Parallel and Distributed Information Systems.*

Bhide, A. (1988) An analysis of three transaction processing architectures. *Proc. Int. Conf. on Very Large DataBases (VLDB).*

Boral, H. *et al.* (1990) Prototyping Bubba, A highly parallel database system. *IEEE Knowledge and Data Engineering*, **2**.

Carey, M. J., DeWitt, D. J. and Naughton, J. F. (1993) The 007 Benchmark. *Proc. Int. Conf. on Very Large DataBases (VLDB'93).*

Copeland, G., *et al.* (1988) Data placement in Bubba. *Proc. ACM SIGMOD Int. Conf. on Management of Data.*

DeWitt, D., *et al.* (1990) The Gamma database machine project. *IEEE Knowledge and Data Engineering*, **2**.

EDS Database Group (1990) EDS—collaborating for a high-performance parallel relational database. *Proc. ESPRIT Conf.*

Englert, S., *et al.* (1989) *A Benchmark of NonStop SQL Release 2 Demonstrating Near-Linear Speedup and Scaleup on Large Databases.* Technical Report 89.4. Tandem Computers, Cupertino, CA.

Franklin, M. and Carey, M. (1992) Global memory management in client–server DBMS architectures. *Proc. Int. Conf. on Very Large DataBases (VLDB).*

Gray, J. (1991) *The Benchmark Handbook.* Morgan Kaufmann, San Mateo, CA.

Kronenberg, N., Levy, H. and Strecker, W. (1986) VAXclusters: a closely coupled distributed system. *ACM Trans. Comp. Syst.*, **4**.

Ozsu, T. and Valduriez, P. (1990) *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, NJ.

Rahm, E. (1991) *Concurrency and Coherency Control in Database Sharing Systems.* Technical Report, University of Kaiserslautern, Germany.

Scrutchin, T. (1987) TPF: performance, capacity, availability. *Proc. IEEE Compcon, Spring.*

Sekino, A., Moritani, K., Masai, T., Tasaki, T. and Goto, K. (1984) The DCS—a new approach to multisystem data sharing. *Proc. Natl. Computer Conf.*

Shoens, K., *et al.* (1985) Amoeba Project. *Proc. IEEE Compcon, Spring.*

Stonebraker, M., *et al.* (1988) The design of XPRS. *Proc. Int. Conf. on Very Large DataBases (VLDB).*

Strickland, J., Uhrowczik, P. and Watts, V. (1982) IMS/VS: an evoluting system. *IBM Syst. J.*, **21**.

Teradata Corp. (1983) *DBC/1012 Database Computer Concepts and Facilities.* Document No. C02-0001-00, Teradata Corp.

Teradata Corp. (1985) *DBC/1012 Database Computer System Manual Release 2.0.* Document No. C10-0001-00, Teradata Corp.

TPC (1992) *TPC Benchmark C—Draft 6.6 Proposed Standard.* Transaction Processing Performance Council, ITOM Int. Co., Los Altos, CA.