The Case for Shared Nothing

Michael Stonebraker
University of California
Berkeley, Ca.

ABSTRACT

There are three dominent themes in building high transaction rate multiprocessor systems, namely shared memory (e.g. Synapse, IBM/AP configurations), shared disk (e.g. VAX/cluster, any multi-ported disk system), and shared nothing (e.g. Tandem, Tolerant). This paper argues that shared nothing is the preferred approach.

1. INTRODUCTION

The three most commonly mentioned architectures for multiprocessor high transaction rate systems are:

shared memory (SM), i.e. multiple processors shared a common central memory

shared disk (SD), i.e. multiple processors each with private memory share a common collection of disks

shared nothing (SN), i.e. neither memory nor peripheral storage is shared among processors

There are several commerical examples of each architecture. In this paper we argue that SN is the most cost effective alternative. In Section 2 we present a "back of the envelope" comparison of the alternatives. Then in Sections 3 through 5 we discuss in more detail some of the points of comparison.

2. A SIMPLE ANALYSIS

In Table 1 we compare each of the architectures on a collection of 12 points. Each architecture is rated 1, 2 or 3 to indicate whether it is the best, 2nd best or 3rd best on each point. For certain points of comparison, there are apparent ties. In such situations we give each system the lower rating. Most of the ratings are self-evident, and we discuss only a few of our values.

The first two rows indicate the difficulty of transaction management in each environment. SM requires few modifications to current algorithms and is the easiest environment to support. Hence it receives a "1" for crash recovery. The "2" for concurrency control results from the necessity of dealing with the lock table as a hot spot. SN is more difficult because it requires a distributed deadlock detector and a multi-phase commit protocol. SD presents the most complex transaction management problems because of the necessity of coordinating multiple copies of the same lock table and synchronizing writes to a common log or logs.

The third and fourth points are closely related. Data base design is difficult in current one-machine environments, and becomes much harder in an SN system where the location of all objects must be specified. The other environments do not add extra complexity to the one-machine situation. Balancing the load

This research was sponsored by the U. .S. Air Force Office of Scientific Research Grant 83-0254 and the Naval Electronics Systems Command Contract N39-82-C-0235

System Feature	shared nothing	shared memory	shared disk
difficulty of concurrency control	2	2	3
difficulty of crash recovery	2	1	3
difficulty of data base design	3	2	2
difficulty of load balancing	3	1	2
difficulty of high availability	1	3	2
number of messages	3	1	2
bandwidth required	1	3	2
ability to scale to large number of machines	1	3	2
ability to have large distances between machines	1	3	2
susceptibility to critical sections	1	3	2
number of system images	3	1	3
susceptibility to hot spots	3	3	3

A Comparison of the Architectures

Table 1

of an SN system is complex, since processes and/or data must be physically moved. It is obviously much easier in the other environments. The next five points are fairly straightforward, and we skip forward to critical sections. They have been shown to be a thorny problem in one-machine systems [BLAS79], and an SN system does not make the problem any worse. On the other hand, an SM system will be considerably more susceptible to this problem, while an SD system will be in-between. SN and SD systems have one system image per CPU, and system administration will be more complex than an SM system which has only a single system image. Lastly, all architectures are susceptible to hot spots.

Several conclusions are evident from Table 1. First an SM system does not scale to a large number of processors. In my opinion this is a fundamental flaw that makes it less interesting than the other architectures. Second, an SD system excells at nothing, i.e. there are no "1"s in its column. Lastly, one should note the obvious marketplace interest in distributed data base systems. Under the assumption that every vendor will have to implement one, there is little or no extra code required for an SN system. In order to justify implementing something else (e.g. SD) and paying the extra software complexity, one should be sure that SN has some insurrountable flaws. In the next section we discuss the issues of data base design, load balancing and number of messages, which are points of comparison where SN was the poorest choice. In each case we argue that the problems are unlikely to be very significant. Then we discuss hot spots in Section 4, and argue that these are easier to get rid of than to support effectively. Lastly, we discuss concurrency control, and suggest that scaling to larger data bases is unlikely to change the ratings in Table 1. Hence, we will conclude that SN offers the most viable and cost effective architecture.

3. Problems with Shared Nothing

It appears that most data base users find data base design to require substantial wizardry. Moreover, tuning a data base is a subject that data base vendors have clearly demonstrated proficiency relative even to the wisest of their customers. To ordinary mortals tuning is a "black art".

Consequently, I expect many automatic tuning aids will be constructed for most data managers, if for no other reason than to lower the technical support burden. There is no evidence that I am aware of that such tuning aids will be unsuccessful. Similarly, there is no evidence that automatic data base design aids will fail in an SN environment where the data base is partitioned over a collection of systems. Furthermore, balancing the load of an SN data base by repartitioning is a natural extension of such a design aid. Moreover, applications which have a stable or slowly varying access pattern will respond successfully to such treatment and will be termed **tunable**. Only data bases with periodic or unpredictable access patterns will be untunable, and I expect such data bases to be relatively uncommon. Hence, load balancing and data base design should not be serious problems in typical environments.

Consider the number of messages which an SN system must incur in a typical high transaction processing environment. The example consists of a data base with N objects subject to a load consisting entirely of transactions containing exactly k commands, each affecting only one record. (For TP1 the value of k is 4). For any partitioning of the data base, these k commands remain single-site commands. Suppose that there exists a partitioning of the data base into non-overlapping collections of objects such that all transactions are locally sufficient [WONG83]. Such a data base problem will be termed **delightful**. Most data base applications are nearly delightful. For example, the TP1 in [ANON84] has 85% delightful transactions.

Assume further that the cost of processing a single record command is X and the cost of sending and receiving a round-trip message is Y. For convenience, measure both in host CPU instructions, and call T = X/Y the **technology ratio** of a given environment. Measured values of T for high speed networks and relational data bases have varied between 1 and 10 and reflect the relative efficiency of data base and networking software in the various situations. An environment where each is tuned well should result in a T of about 3. We expect the long term value of T to stay considerably greater than 1, because it appears much easier to offload network code than data base code.

As long as T >> 1, network costs will not be the dominent system cost in delightful data bases; rather it will be processing time on the local systems. Moreover, data bases that are nearly delightful will require a modest number of messages. (With a reasonable amount of optimization, it is conceivable to approach 2

messages per transaction for locally sufficient transactions.) Hence, the number of messages should not be a problem for the common case, that of nearly delightful data bases.

4. Hot Spots

Hot spots are a problem in all architectures, and there are at least three techniques to dealing with them.

1) get rid of them
2) divide a hot spot record into N subrecords [ANON84]
3) use some implementation of a reservation system [REUT81]

It has never been clear to me why the branch balance must be a stored field in TP1. In the absence of incredible retrieval volume to this item, it would be better to calculate it on demand. The best way to eliminate problems with hot spots is to eliminate hot spots.

Unfortunately, there are many hot spots which cannot be deleted in this fashion. These include critical section code in the buffer manager and in the lock manager, and "convoys" [BLAS79] results from serial hot spots in DBMS execution code. In addition, the head of the log and any audit trail kept by an application are guaranteed to be hot spots in the data base. In such cases the following tactic can usually be applied.

Decompose the object in question into N subobjects. For example, the log can be replicated N times, and each transaction can write to one of them. Similarly, the buffer pool and lock table can be decomposed into N subtables. Lastly, the branch balance in TP1 can be decomposed into N balances which sum to the correct total balance. In most cases, a transaction requires only one of the N subobjects, and the conflict rate on each subobject is reduced by a factor of N. Of course, the division of subobjects can be hidden from a data base user and applied automatically by the data base designer, whose existence we have speculated in Section 3.

Lastly, when updates are restricted to increment and decrement of a hot spot field, it is possible to use field calls (e.g. IMS Fast Path) or a reservation system [REUT82]. It is clear that this tactic can be applied equally well to any of the proposed architectures; however, it is not clear that it ever dominates the "divide into subrecords" tactic. Consequently, hot spots should be solvable using conventional techniques.

5. Will Concurrency Control Become a Bigger Problem?

Some researchers [REUT85] argue that larger transaction systems will generate a thorny concurrency control problem which may affect the choice of a transaction processing architecture. This section argues that such an event will probably be uncommon.

Consider the observation of [GRAY81] which asserts that deadlocks are rare in current systems and that the probability of a transaction waiting for a lock request is rare (e.g. .001 or .0001). The conclusion to be drawn from such studies is that concurrency control is not a serious issue in well designed systems today. Consider the effect of scaling such a data base application by a factor of 10. Hence, the CPU is replaced by one with 10 times the throughput. Similarly 10 times the number of drives are used to accelerate the I/O system a comparable amount. Suppose 10 times as many terminal operators submit 10 times as many transactions to a data base with 10 times the number of lockable objects. It is evident from queuing theory that the average response time would remain the same (although variance increases) and the probability of waiting will remain the same. The analyis in [GRAY81] can be rerun to produce the identical results. Hence, a factor of 10 scaling does not affect concurrency control issues, and today's solutions will continue to act as in current systems.

Only two considerations cloud this optimistic forcast. First, the conclusion is predicated on the assumption that the number of granules increases by a factor of 10. If the size of a granule remains a constant, then the size of the data base must be linear in transaction volume. We will term such a data base problem **scalable**. Consider the transactions against a credit card data base. The number of transactions per credit card per month is presumably varying slowly. Hence, only a dramatic increase in the number of cards outstanding (and hence data base size) could produce a large increase in transaction rates. This data

base problem appears to be scalable. In addition, suppose a fixed number of travel agent transactions are generated per seat sold on a given airline. Consequently, transaction volume is linear in seat volume (assuming that planes are a constant size) and another scalable data base results.

One has to think hard to discover nonscalable data bases. The one which comes to mind is TP1 in an environment where retail stores can debit one's bank account directly as a result of a purchase [ANON84]. Here, the number of transactions per account per month would be expected to rise dramatically resulting in a nonscalable data base. However, increasing the size of a TP1 problem will result in no conflicts for the account record (a client can only be initiating one retail transaction at a time) and no conflict for the teller record (a clerk can only process one customer at a time). Hence, the only situation with increased conflict would be on summary data (e.g. the branch balance). Concurrency control on such "hot spots" should be dealt with using the techniques of the previous section.

The following conclusions can be drawn. In scalable data bases (the normal case) concurrency control will remain a problem exactly as difficult as today. In nonscalable data bases it appears that hot spots are the main concurrency control obstacle to overcome. Hence, larger transaction systems in the best case present no additional difficulties and in the worst case aggravate the hot spot problem.

6. CONCLUSIONS

In scalable, tunable, nearly delightful data bases, SN systems will have no apparent disadvantages compared to the other alternatives. Hence the SN architecture adequately addresses the common case. Since SN is a nearly free side effect of a distributed data base system, it remains for the advocates of other architectures to demonstrate that there are enough non-tunable or non-scalable or non delightful problems to justify the extra implementation complexity of their solutions.

REFERENCES

[ANON84] Anon et. al., "A Measure of Transaction Processing Power", unpublished working paper.

[BLAS79] Blasgen, M. et. al., "The Convoy Phenomenon," Operating Systems Review, April 1979.

[GRAY81] Gray, J. et. al., "A Straw Man Analysis of Probability of Waiting and deadlock," IBM Research, RJ3066, San Jose, Ca., Feb 1981.

[REUT82] Reuter, A., "Concurrency on High-Traffic Data Elements," ACM-PODS, March 1982.

[REUT85] Reuter, A., (private communication).

[WONG83] Wong, E. and Katz, R., "Distributing a Data Base for Parallelism," ACM-SIGMOD, May 1983.