The Design of an ODMG Compatible Parallel Object Database Server (Invited Talk)

Paul Watson

Department of Computing Science, University of Newcastle-upon-Tyne, NE1 7RU, UK
Paul.Watson@newcastle.ac.uk

Abstract. The *Polar* project has the aim of designing a parallel, ODMG compatible object database server. This paper describes the server requirements and investigates issues in designing a system to achieve them. We believe that it is important to build on experience gained in the design and usage of parallel relational database systems over the last ten years, as much is also relevant to parallel object database systems. Therefore we present an overview of the design of parallel relational database servers and investigate how their design choices could be adopted for a parallel object database server. We conclude that while there are many similarities in the requirements and design options for these two types of parallel database servers, there are a number of significant differences, particularly in the areas of object access and method execution.

1 Introduction

The parallel database server has become the "killer app" of parallel computing. The commercial market for these systems is now significantly larger than that for parallel systems running numeric applications, making them mainstream IT system components offered by a number of major computer vendors. They can provide high performance, high availability, and high storage capacity, and it is this combination of attributes which has allowed them to meet the growing requirements of the increasing number of computer system users who need to store and access large amounts of information.

There are a number of reasons to explain the rapid rise of parallel database servers, including:

- they offer higher performance at a better cost-performance ratio than do the previous dominant systems in their market - mainframe computers.
- designers have been able to produce highly available systems by exploiting the natural redundancy of components in parallel systems. High availability is important because many of these systems are used for business-critical applications in which the financial performance of the business is compromised if the data becomes inaccessible.
- the 1990s have seen a process of re-centralisation of computer systems following the trend towards downsizing in the 1980s. The reasons for this include: cost

savings (particularly in software and system management), regaining central control over information, improving data integrity by reducing duplication, and increasing access to information. This process has created the demand for powerful information servers.

- there has been a realisation that many organisations can derive and infer valuable information from the data held in their databases. This has led to the use of techniques, such as data-mining, which can place additional load on the database server from which the base data on which they operate must be accessed.
- the growing use of the Internet and intranets as ways of making information available both outside and inside organisations has increased the need for systems which can make large quantities of data available to large numbers of simultaneous users.

The growing importance of parallel database servers is reflected in the design of commercial parallel platforms. Efficient support for parallel database servers is now a key design requirement for the majority of parallel systems.

To date, almost all parallel database servers have been designed to support relational database management systems (RDBMS) [1]. A major factor which has simplified, and so encouraged, the deployment of parallel RDBMS by organisations is their structure. Relational database systems have a client-server architecture in which client applications can only access the server through a single restricted and well defined query interface. To access data, clients must send an SQL (Structured Query Language) query to the server where it is compiled and executed. This architecture allows a serial server, which is not able to handle the workload generated by a set of clients, to be replaced by a parallel server with higher performance. The client applications are unchanged: they still send the same SQL to the parallel server as they did to the serial server because the exploitation of parallelism is completely internal to the server.

Existing parallel relational database servers exploit two major types of parallelism. Inter-query parallelism is concerned with the simultaneous execution of a set of queries. It is typically used for On-Line Transaction Processing (OLTP) workloads in which the server processes a continuous stream of small transactions generated by a set of clients. Intra-Query parallelism is concerned with exploiting parallelism within the execution of single queries so as to reduce their response time.

Despite the current market dominance of relational database servers, there is a growing belief that relational databases are not ideal for a number of types of applications, and in recent years there has been a growth of interest in object oriented databases which are able to overcome many of the problems inherent in relational systems [2]. In particular, the growth in the use of object oriented programming languages, such as C++ and Java, coupled with the increasing importance of object-based distributed systems has promoted the use of object database management systems (ODBMS) as key system components for object storage and access. A consequence of the interest in object databases has been an attempt to define a standard specification for all the key ODBMS interfaces - the Object Database Management Group ODMG 2.0 standard [3].

The *Polar* project has the aim of designing and implementing a prototype ODMG compatible parallel object database server. Restricting the scope of the project to this standard allows us to ignore many issues, such as query language design and programming language interfaces, and instead focus directly on methods for exploiting parallelism within the framework imposed by the standard.

In this paper we describe the requirements that the *Polar* server must meet and investigate issues in designing a system to achieve them. Rather than design the system in isolation, we believe that it is important to build, where possible, on the extensive experience gained over the last ten years in the design and usage of parallel relational database systems. However, as we describe in this paper, differences between the object and relational database paradigms result in significant differences in some areas of the design of parallel servers to support them. Those differences in the paradigms, which have most impact on the design, are:

- objects in an object database can be referenced by a unique identifier, or (indirectly) as members of a collection. In contrast, tables (collections of rows) are the only entities which can be referenced by a client of a relational database (i.e. individual table rows cannot be directly referenced).
- there are two ways to access data held in an object database: through a query language (OQL), and by directly mapping database objects into client application program objects. In a relational database, the query language (SQL) is the only way to access data.
- objects in an object database can have associated user-defined methods which may be called within queries, and by client applications which have mapped database objects into program objects. In a relational database, there is no equivalent of user-defined methods: only a fixed set of operations is provided by SQL.

The structure of the rest of this paper is as follows. In Section 2 we give an overview of the design of parallel relational database servers, based on our experiences in two previous parallel database server projects: EDS [4], and Goldrush [1]. Next, in Section 3 we define our requirements for the *Polar* parallel ODBMS. Some of these are identical to those of parallel RDBMS; others have emerged from experience of the limitations of existing parallel servers; while others are derived from our view of the potential use of parallel ODBMS as components in distributed systems. Based on these requirements, in Section 4, we present an overview of issues in the design of a parallel ODBMS. This allows us to highlight those areas in the design of a parallel object database server where it is possible to adopt solutions based on parallel RDBMS or serial ODBMS, and, in contrast, those areas where new solutions are required. Finally, in Section 5 we draw conclusions from our investigations, and point to further work.

1.1 Related Work

There have been a number of parallel relational database systems described in the literature. Those which have most influenced this paper are the two on which we previously worked: EDS and Goldrush.

The EDS project [4] designed and implemented a complete parallel database server, including hardware, operating system and database. The database itself was basically relational though there were some extensions to provide support for objects.

The Goldrush project within ICL High Performance Systems [1, 5, 6] designed a parallel relational database server product running Parallel Oracle.

There is extensive coverage in the literature of research into the design of serial ODBMS. One of the most complete is the description of the O2 system [7].

Recently, there has been some research into the design of parallel ODBMS. For example, Goblin [8] is a parallel ODBMS. However, unlike the system described in this paper, it is limited to a main-memory database.

Work on object servers such as Shore [9] and Thor [10] is also of relevance as this is a key component of any parallel ODBMS. We will refer to this work at appropriate points in the body of the paper.

2 The Design of Parallel Relational Database Servers

Parallel relational database servers have been designed, implemented and utilised for over ten years, and it is important that this experience is used to inform the design of parallel object database servers. Therefore, in this section we give an overview of the design of parallel relational database servers. This will then allow us to highlight commonalties and differences in the requirements (Section 3) and design options (Section 4) between the two types of parallel database servers. The rest of this section is structured as follows. We begin by describing the architectures of parallel platforms (hardware and operating system) designed to support parallel database servers. Next, we describe methods for exploiting parallelism found in relational database workloads. Throughout this section we will draw on examples from our experience in the design and use of the ICL Goldrush MegaServer [1].

2.1 Parallel Platforms

In this section, we describe the design of parallel platforms (which we define as comprising the hardware and operating system) to meet the requirements of database servers.

We are interested in systems utilising the highly scaleable distributed memory parallel hardware architecture [5] in which a set of computing nodes are connected by a high performance network (Fig. 1). Each node has the architecture of a uniprocessor or shared store multiprocessor - one or more CPUs share the local main memory over a bus. Typically, each node also has a set of locally connected disks for the persistent storage of data. Connecting disks to each node allows the system to provide both high IO performance, by supporting parallel disk access, and high storage capacity. In most current systems the disks are arranged in a share nothing configuration - each disk is physically connected to only one node. As will be seen, this has major implications for the design of the parallel database server software. It is likely that in future the availability of high bandwidth peripheral interconnects will lead to the design of platforms in which each disk is physically connected to more than one node [11], however this `shared disk' configuration is not discussed further in this paper as we focus on currently prevalent hardware platform technology. Database servers require large main memory caches for efficient performance (so as to reduce the number of disk accesses) and so the main memories tend to be large (currently 0.25-4GB is typical). Some of the nodes in a parallel platform will have external network connections to which clients are connected. External database clients send database queries to the server through these connections and later receive the results via them. The number of these nodes (which we term Communications Nodes) varies depending on the required performance and availability. In terms of performance, it is important that there are enough Communications Nodes to perform client communication without it becoming a bottleneck, while for availability it is important to have more than one route from a client to a parallel server so that if one fails, another is available.

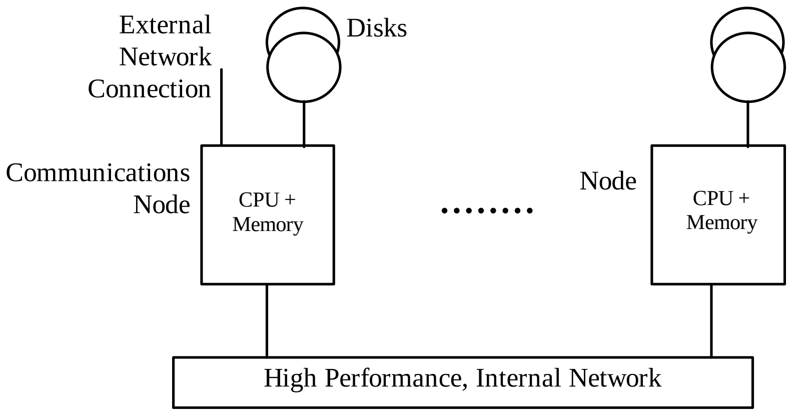
External Network Connection
Disks
Communications Node
CPU + Memory
Node
CPU + Memory
High Performance, Internal Network

Fig. 1. A Distributed Memory Parallel Architecture

The distributed memory parallel hardware architecture is highly scaleable as adding a node to a system increases all the key performance parameters including: processing power, disk throughput and capacity, and main memory bandwidth and capacity. For a database server, this translates into increased query processing power, greater database capacity, higher disk throughput and a larger database cache.

The design of the hardware must also contribute to the creation of a high availability system. Methods of achieving this include providing component redundancy and the ability to replace failed components through hot-pull & push techniques without having to take the system down [5].

The requirement to support a database server also influences the design of the operating system in a number of ways. Firstly, commercial database server software depends on the availability of a relatively complete set of standard operating system facilities, including support for: file accessing, processes (with the associated inter-process communications) and external communications to clients through standard protocols. Secondly, it is important that the cost of inter-node communications is minimised as this directly affects the performance of a set of key functions including remote data access and query execution. Finally, the operating system must be designed to contribute to the construction of a high availability system. This includes ensuring that the failure of one node does not cause other nodes to fail, or unduly affect their performance (for example through having key services such as distributed file systems delayed for a significant time waiting, until a long time-out occurs, for a response from the failed node).

Experience in a number of projects suggests that micro-kernel based operating systems provide better support than do monolithic kernels for adding and modifying the functionality of the kernel to meet the requirements of parallel database servers [1]. This is for two reasons: they provide a structured way in which new services can be added, and their modularity simplifies the task of modifying existing services, for example to support high-availability [4].

2.2 Exploiting Parallelism

There are two main schemes utilised by parallel relational database servers for query processing on distributed memory parallel platforms. These are usually termed Task Shipping and Data Shipping. They are described and compared in this subsection so that in Section 4 we can discuss their appropriateness to the design of parallel object database servers.

In both Task and Data shipping schemes, the database tables are horizontally partitioned across a set of disks (i.e. each disk stores a set of rows). The set of disks is selected to be distributed over a set of nodes (e.g. it may consist of one disk per node of the server). The effect is that the accesses to a table are spread over a set of disks and nodes, so giving greater aggregate throughput than if the table was stored on a single disk. This also ensures that the table size is not limited to the capacity of a single disk.

In both schemes, each node runs a database server which can compile and execute a query or, as will be described, execute part of a parallelised query. When a client sends an SQL query to one of the Communications Nodes of the parallel server it is directed to a node where it is compiled and executed either serially, on a single node, or in parallel, across a set of nodes. The difference between the two schemes is in the method of accessing database tables as is now described.

2.2.1 Data Shipping

In the data shipping scheme the parallel server runs a distributed filesystem which allows each node to access data from any disk in the system. Therefore, a query running on a node can access any row of any table, irrespective of the physical location of the disk on which the row is stored.

When inter-query parallelism is exploited, a set of external clients generate queries which are sent to the parallel server for execution. On arrival, the Communications Node forwards them to another node for compilation and execution. Therefore the Communication Nodes need to use a load balancing algorithm to select a node for the compilation and execution of each query. Usually, queries in OLTP workloads are too small to justify parallelisation and so each query is executed only on one node. When a query is executed, the database server accesses the required data in the form of table rows. Each node holds a database cache in main memory and so if a row is already in the local cache it can be accessed immediately. However, if it is not in the cache then the distributed filesystem is used to fetch it from disk. The unit of transfer between the disk and cache is a page of rows. When query execution is complete the result is returned to the client.

When a single complex query is to be executed in parallel (intra-query parallelism) then parallelism is exploited within the standard operators used to execute queries: scan, join and sort [12]. For example, if a table has to be scanned to select rows which meet a particular criteria then this can be done in parallel by having each of a set of nodes scan a part of the table. The results from each node are appended to produce the final result. Another type of parallelism, pipeline parallelism, can be exploited in queries which require multiple levels of operators by streaming the results from one operator to the inputs of others.

Because, in the Data Shipping model, any node can access any row of any table the compiler is free to decide for each query both how much parallelism it is sensible to exploit, and the nodes on which it should be executed. Criteria for making these decisions include the granularity of parallelism and the current loading of the server, but it is important to note that these decisions are not constrained by the location of the data on which the query operates. For example, even if a table is only partitioned over the disks of three nodes, a compiler can still choose to execute a scan of that table over eight nodes.

The fact that any node can access data from disks located anywhere in the parallel system has a number of major implications for the design of the parallel database server:

- the distributed filesystem must meet a number of requirements which are not found in typical, conventional distributed filesystems such as NFS. These include high performance access to both local and remote disks, continuous access to data even in the presence of disk and node failures, and the ability to perform synchronous writes to remote disks. Consequently, considerable effort has been expended in this area by developers of parallel database platforms.
- a global lock manager is required as a resource shared by the entire parallel system. A table row can be accessed by a query running on any node of the system. Therefore the lock which protects that row must be managed by a single component in the system so that queries running on different nodes of the system see the same lock state. Consequently, any node which wishes to take a lock must communicate with the single component that manages that lock. If all locks in a system were to be managed by a single component - a centralised lock manager - then the node on which that component ran would become a bottleneck, reducing the system's scalability. Therefore, parallel database systems usually implement the lock manager in a distributed manner. Each node runs an instance of the lock manager which manages a subset of the locks [1]. When a query needs to acquire or drop a lock it sends a message to the node whose lock manager instance manages that

lock. One method of determining the node which manages a lock is to use a hash function to map a lock identifier to a node. All the lock manager instances need to co-operate and communicate to determine deadlocks, as circles of dependencies can contain a set of locks managed by more than one lock manager instance [13].

- distributed cache management is required. A row can be accessed by more than one node and this raises the issue that a row could be cached in more than one node at the same time. It is therefore necessary to have a scheme for maintaining cache coherency. One solution is the use of cache locks managed by the lock manager.

The efficiency of database query execution in a Data Shipping system is highly dependent on the cache hit rate. A cache miss requires a disk access and the execution of filesystem code on two nodes (assuming the disk is remote), which increases response time and reduces throughput. Two techniques can be used to reduce the resulting performance degradation. The first is to, where possible, direct queries which access the same data to the same node. For example all banking queries from one branch could be directed to the same node. This increases cache hit rates and reduces pinging: the process by which cached pages have to be moved around the caches of different nodes of the system because the data contained in them is updated by queries running on more than one node. The second technique extends the first technique so that not only are queries operating on the same data sent to the same node, but that node is selected as the one which holds the data on its local disks. The result is that if there is a cache miss then only a local disk access is required. This not only reduces latency, but also increases throughput as less code is executed in the filesystem for a local access than a remote access. This second technique is closely related to Task Shipping, which is now described.

2.2.2 Task Shipping

The key difference between the two Shipping schemes is that in Task Shipping only those parts of tables stored on local disks can be accessed during query evaluation. Consequently, each query must be decomposed into sub-queries, each of which only requires access to the table rows stored on a single node. These sub-queries are then shipped to the appropriate nodes for execution. For a parallel scan operation, this is straightforward to organise, though it does mean that, unlike in the case of Data Shipping, the parallel speed-up is limited by the number of nodes across which the table is partitioned, unless the data is temporarily re-partitioned by the query. When a join is performed, if the two tables to be joined have been partitioned across the nodes such that rows with matching values of the join attribute are stored on disks of the same node, then the join can straightforwardly be carried out as a parallel set of local joins, one on each node. However, if this is not the case, then at least one of the tables will have to be re-partitioned before the join can be carried out. This is achieved by sending each row of the table(s) to the node in which it will participate in a local join. Parallel sorting algorithms also require this type of redistribution of rows, and so it may be said that the term Task Shipping is misleading because in common situations it is necessary to ship data around the parallel system. However because data is only ever accessed from disk on one node - the node connected to the disk on which it is stored, then some aspects of the system design are simplified when compared to the Data Shipping scheme, viz.:

- there is no requirement for a distributed filesystem because data is always accessed from local disk (however, in order that the system be resilient to node failure, it will still be necessary to duplicate data on a remote node or nodes - see Section 2.3).
- there is no requirement for global cache coherency because data is only cached on one node - that connected to the disk on which it is stored.
- lock management is localised. All the accesses to a particular piece of data are made only from one node - the node connected to the disk on which it is stored. This can therefore run a local lock manager responsible only for the local table rows. However, a 2-phase commitment protocol is still required across the server to allow transactions which have been fragmented across a set of nodes to commit in a co-ordinated manner. Further, global deadlock detection is still required.

An implication of the Task Shipping scheme is that even small queries which contain little or no parallelism still have to be divided into sub-queries if they access a set of table rows which are not all stored on one node. This will be inefficient if the sub-queries have low granularity.

2.3 Availability

If a system is to be highly available then its components must be designed to contribute to this goal. In this section we discuss how the database server software can be designed to continue to operate when key components of a parallel platform fail.

A large parallel database server is likely to have many disks, and so disk failure will be a relatively common occurrence. In order to avoid a break of service for recovery when a disk fails it is necessary to duplicate data on more than one disk. Plexing and RAID schemes can be used for this. In Goldrush, data is partitioned over a set of disk volumes, each of which is duplexed. The two plexes are always chosen to be remote from each other, i.e. held on disks not connected to the same node, so that even if a node fails then at least one plex is still accessible.

If a node fails then the database server running on that node will be lost, but this need not prevent other nodes from continuing to provide a service, all be it with reduced aggregate performance. Any lock management data held in memory on the failed node becomes inaccessible and so, as this may be required by transactions running on other nodes, the lock manager design must ensure that the information is still available from another node. This can be achieved by plexing this information across the memories of two nodes.

Failure of an external network link may cause the parallel server to loose contact with an external client. In the Goldrush system, the parallel server holds information on alternative routes to clients and monitors the external network connections so that if there is a failure then an alternative route can be automatically chosen.

3 Parallel ODBMS Requirements

In this section we describe and justify the requirements which we believe a parallel ODBMS server should meet. A major source of requirements is derived from our experience with the design and usage of the ICL Goldrush MegaServer [1, 5, 6] which embodies a number of the functions and attributes that will also be required in a parallel ODBMS. However, it is also the case that there are interesting differences between the requirements for a parallel RDBMS, such as Goldrush, and a parallel ODBMS. In particular, we envisage a parallel ODBMS as a key component for building high performance distributed applications as it has attributes not found in alternative options including: performance, availability, rich interfaces for accessing information (including querying), and transactional capability to preserve database integrity. However, if a parallel ODBMS is to fulfill its potential in this area then we believe that there are a set of requirements which it must meet, and these are discussed in this section.

The rest of this section is structured as follows. We first describe the overall system requirements, before examining the non-functional requirements of performance and availability.

3.1 Systems Architecture

Our main requirement is for a server which provides high performance to ODMG compatible database clients by exploiting both inter-query and intra-query parallelism. This requires a system architecture similar to that described for RDBMS in Section 2. Client applications may generate OQL queries which are sent for execution to the parallel server (via the Communications Nodes). To support intra-query parallelism the server must be capable of executing these queries in parallel across a set of nodes. The server must also support inter-query parallelism by simultaneously executing a set of OQL queries sent by a set of clients. Clients may also map database objects into program objects in order to (locally) access their properties and call methods on them. Therefore, the server must also satisfy requests from clients for copies of objects in the database. The roles of the parallel system, serving clients, are summarised by Figure 2.

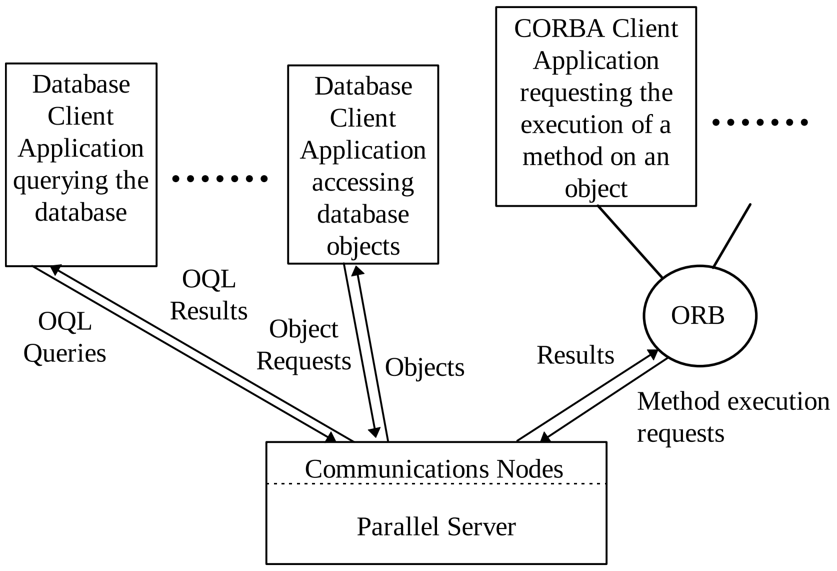
Database Client Application querying the database
•••••••
Database Client Application accessing database objects
CORBA Client Application requesting the execution of a method on an object
•••••••
OQL Queries
OQL Results
Object Requests
Objects
Results
ORB
Method execution requests
Communications Nodes
Parallel Server

Fig. 2. Clients of the Parallel ODBMS Server

The architecture does not preclude certain clients from choosing to execute OQL queries locally, rather than on the parallel server. This would require the client to have local OQL execution capability, but the client would still need to access the objects required during query execution from the server. Reasons for doing this might be to exploit a particularly high performance client, or to remove load from the server so that it can spend more of its computational resources on other, higher priority queries.

Where the ODBMS is to act as a component in a distributed system, we require that it can be accessed through standard CORBA interfaces [14]. The ODMG standard proposes that a special component - an Object Database Adapter (ODA) - is used to connect the ODBMS to a CORBA Object Request Broker (ORB) [3]. This allows the ODBMS to register a set of objects as being available for external access via the ORB. CORBA clients can then generate requests for the execution of methods on these objects. The parallel ODBMS services these requests, returning the results to the clients. In order to support high throughput, the design of the parallel ODBMS must ensure that the stream of requests through the ODA is executed in parallel.

3.2 Performance

Whilst high absolute performance is a key requirement, experience with the commercial usage of parallel relational database servers has shown that scalability is a key attribute. It must be possible to add nodes to increase the server's performance for both inter-query and intra-query parallelism. In Section 2, we described how the distributed memory parallel architecture allows for scalability at the hardware platform level, but this is only one aspect of achieving database server scalability. Adding a node to the system provides more processing power which may be used to speed-up the evaluation of queries. However for those workloads whose performance is largely dependent on the performance of object access, then adding a node will have little or no immediate effect. In the parallel ODBMS, sets of objects will be partitioned across a set of disks and nodes (c.f. the design of a parallel RDBMS as described in Section 2), and so increasing the aggregate object access throughput requires re-partitioning the sets of objects across a larger number of disks and nodes.

Experience with serial ODBMS has shown that the ability to cluster on disk objects likely to be accessed together is important for reducing the number of disk accesses and so increasing performance [15]. Indeed, the ODMG language bindings provide versions of object constructors which allow the programmer to specify that a new object should be clustered with an existing object [3]. Over time, the pattern of access to objects may change, or be better understood (due to information provided by performance monitoring), and so support for re-clustering is also required to allow performance tuning.

As will be seen in Section 4, the requirement to support the re-partitioning and re-clustering of objects has major implications for the design of a parallel ODBMS.

3.3 Performance Management

This section describes requirements which relate to the use of a database as a resource shared among a set of services with different performance requirements.

The computational resources (CPU, disk IOs, etc.) made available by high performance systems are expensive - customers pay more per unit resource on a high performance system than they do on a commodity computer system. In many cases, resources will not be completely exhausted by a single service but if the resources are to be shared then mechanisms are needed to control this sharing.

An example of the need for controlled resource sharing is where a database runs a business-critical OLTP workload which does not utilise all the server’s resources. It may be desirable to run other services against the same database, for example in order to perform data mining, but it is vital that the performance of the business-critical OLTP service is not affected by the loss of system resources utilised by other services. In a distributed memory parallel server, the two services may be kept apart on non-intersecting sets of nodes in order to avoid conflicts in the sharing of CPU resources. However, they will still have to share the disks on which the database is held. A solution might be to analyse the disk access requirements of the two services and try to ensure that the data is partitioned over a sufficiently large set of nodes and disks so as to be able to meet the combined performance requirements of both services. However this is only possible where the performance requirements of the two services are relatively static and therefore predictable. This may be true of the OLTP service, but the performance of a complex query or data mining service may not be predictable. A better solution would be to ensure that the OLTP service was allocated the share of the resources that it required to meet its performance requirements, and only allow the other service the remainder.

A further example of the importance of controlled resource sharing is where parallel ODBMS act as repositories for persistent information on the Internet and intranets. If information is made available in this way then it is important that the limited resources of the ODBMS are shared in a controlled way among clients. For example there is a danger of denial of service attacks in which all the resources of system are used by a malicious client. Similarly, it would be easy for a non-malicious user to generate a query which required large computational resources, and so reduced the performance available to other clients below an acceptable level. Finally, it may be desirable for the database provider to offer different levels of performance to different classes of clients depending on their importance or payment.

The need for controlled sharing of resources has long been recognised in the mainstream computing world. For example, mainframe computer systems have for several decades provided mechanisms to control the sharing of their computational resources among a set of users or services. Some systems allow CPU time and disk IOs to be divided among a set of services in any desired ratio. Priority mechanisms for allocating resources to users and services may also be offered. Such mechanisms are now also provided for some shared memory parallel systems.

Unfortunately, the finest granularity at which these systems control the sharing of resources tends to be at an Operating System process level. This is too crude for database servers, which usually have at their heart a single, multi-threaded process which executes the queries generated by a set of clients (so reducing switching costs when compared to an implementation in which each client has its own process running on the server). Therefore, in conclusion, we believe that the database server must provide its own mechanisms to share resources among a set of services in a controlled manner if expensive parallel system resources are to be fully utilised, and if object database servers are to fulfil their potential as key components in distributed systems.

3.4 Availability

The requirements here are identical to those of a parallel RDBMS. The parallel ODBMS must be able to be continue to operate in the presence of node, disk and network failures. It is also important that availability is considered when designing database management operations which may effect the availability of the database service. These include: archiving, upgrading the server software and hardware, and re-partitioning and re-clustering data to increase performance. Ideally these should all be achievable on-line, without the need to shut-down the database service, but if this is not possible then the time for which the database service is down should be minimised.

A common problem causing loss of availability during management operations is human error. Manually managing a system with up to tens of nodes, and hundreds of disks is very error prone. Therefore tools to automate management tasks are important for reducing errors and increasing availability.

3.5 Hardware Platform

The *Polar* project has access to a parallel database platform, an ICL Goldrush MegaServer [1], however we are also investigating alternatives. Current commercial parallel database platforms use custom designed components (including internal networks, processor-network interfaces and cabinetry), which leads to high design, development and manufacturing costs, which are then passed on to users. Such systems generally have a cost-performance ratio that is significantly higher than that of commodity, uniprocessor systems. We are investigating a solution to this problem and have built a parallel machine, the Affordable Parallel Platform (APP), entirely from standard, low-cost, commodity components - high-performance PCs inter-connected by a high throughput, scaleable ATM network. Our current system consists of 13 nodes interconnected by 155Mbps ATM. The network is scaleable as each PC has a full 155Mpbs connection to the other PCs. This architecture has the interesting property that high-performance clients can be connected directly to the ATM switch giving them a very high bandwidth connection to the parallel platform. This may, for example, be advantageous if large multimedia objects must be shipped to clients.

Our overall aim in this area is to determine whether commodity systems such as the APP can compete with custom parallel systems in terms of database performance. Therefore we require that the database server design and implementation is portable across, and tuneable for, both types of parallel platform.

4 Parallel ODBMS Design

In this section we consider issues in the design of a parallel ODBMS to meet the requirements discussed in the last section. We cover object access in the most detail as it is a key area in which existing parallel RDBMS and serial ODBMS designs do not offer a solution. Cache coherency, performance management and query processing are also discussed.

4.1 Object Access

Objects in an object database can be individually accessed through their unique identifier. This is a property not found in relational databases and so we first discuss the issues in achieving this on the parallel server (we leave discussion of accessing collections of objects until Section 4.3).

As described earlier, the sets of objects stored in the parallel server are partitioned across a set of disks and nodes in order to provide a high aggregate access throughput to the set of objects. When an object is created, a decision must be made as to the node on which it will be stored. When a client needs to access a persistent object then it must send the request to the node holding the object.

In the rest of this section we discuss in more detail the various issues in the storage and accessing of objects in a parallel ODBMS as these differ significantly from the design of a serial ODBMS or a parallel RDBMS. Firstly we discuss how the Task Shipping and Data Shipping mechanisms described in Section 2 apply to a parallel ODBMS. We then discuss possible options for locating and accessing persistent objects.

4.1.1 Task Shipping vs. Data Shipping

In a serial ODBMS, the database server stores the objects and provides a run-time system which allows applications running on external clients to transparently access objects. Usually a page server interface is offered to clients, i.e. a client requests an object from the server which responds with a page of objects [7]. The page is cached in the client because it is likely that the requested object and others in the same page will be accessed in the near future. This is a Data Shipping architecture, and has the implication that the same object may be cached in more than one client, so necessitating a cache coherency mechanism. If Data Shipping was also adopted within the parallel server then nodes executing OQL would also access and cache objects from remote nodes.

We now consider if the alternative used in some parallel RDBMS - a Task Shipping architecture - is a viable option for a parallel ODBMS. There are two computations which can be carried out on objects: executing a method on an object and accessing a property of an object. These computations can occur either in an application running on an external client, or during OQL execution on a node of the parallel server.

In order to implement task shipping, the external clients, and nodes executing OQL would not cache remote objects and process them locally. Instead, they would have to send a message to the node which stored the object requesting that it perform the computation (method execution or property access) locally and return the result. In this way, the client does not cache objects, and so cache coherency is not an issue. However Task Shipping in object database servers does have two major drawbacks. Firstly, the load on the server is increased when compared to the data shipping scheme. This means that a server is likely to be able to support fewer clients. Secondly, the latency of task shipping - the time between sending the task to the server and receiving the response - is likely to result in performance degradation at the client when compared to the data shipping scheme in which the client can operate directly on the object without any additional latency once it has been cached. Many database clients are likely to be single application systems, and so there will not be other processes to run while a computation on an object is being carried out on the parallel server - consequently the CPU will be idle during this time. Further, the user of an interactive client application will be affected by the increase in response time caused by the latency.

The situation is slightly different for the case of OQL query execution on the nodes of the server under a Task Shipping scheme. When query execution is suspended while a computation on an object is executed remotely, it is very likely that there will be other work to do on the node (for example, servicing requests for operations on objects and executing other queries) and so the CPU will not be idle. However, it is likely that the granularity of parallel processing (the ratio of computation to communication) will be low due to the frequent need to generate requests for computations on remote objects. Therefore, although the node may not be idle, it will be executing frequent task switches and sending/receiving requests for computations on objects. The effect of this is likely to be a reduction in throughput when compared to a data shipping scheme with high cache hit rates which will allow a greater granularity of parallelism.

For these reasons, we do not further consider a task shipping scheme, and so must address the cache coherence issue (Section 4.2). Figure 3 shows the resulting parallel database server architecture. The objects in the database are partitioned across disks connected to the nodes of the system, taking into account clustering. Requests to access objects will come from both: applications running on external database clients; and the nodes of the parallel server which are executing queries or processing requests for method execution from CORBA clients. The database client applications are supported by a run-time system (RTS) which maintains an object cache. If the application accesses an object which is not cached then an object request is generated and sent to a Communication Node (CN) of the parallel server. The CN uses a Global Object Access module to determine the node which has the object stored on one of its local disks. The Remote Object Access component forwards the request to this node where it is serviced by the Local Object Access component which returns the page containing the object back to the client application RTS via the CN.

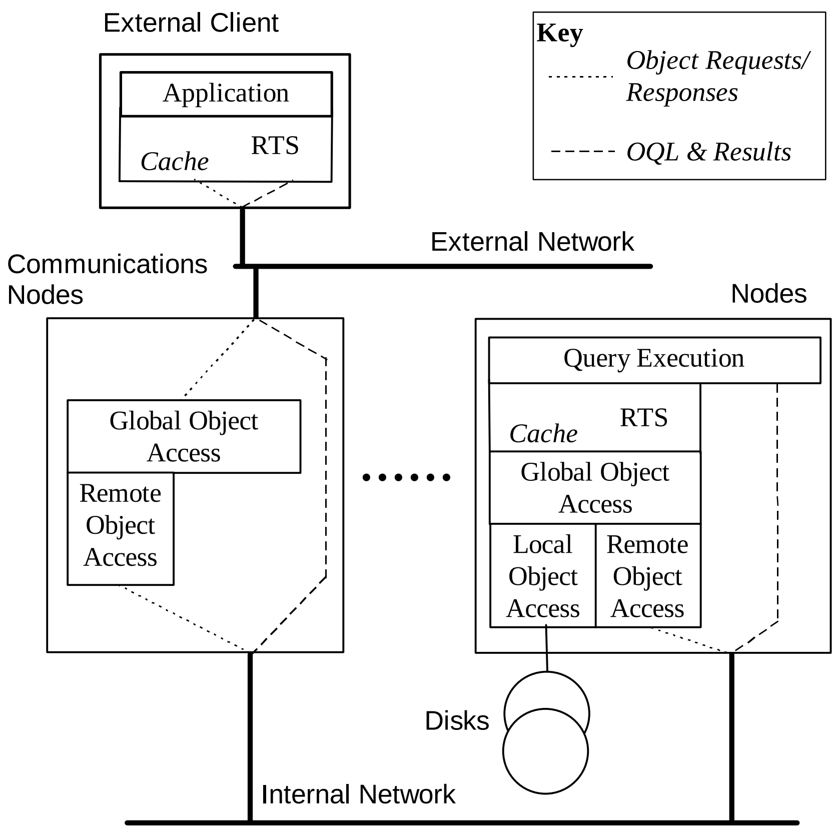

Fig. 3. Database Server System Architecture

If the client application issues an OQL query then the client's database RTS sends it to a CN of the parallel server. From there it is forwarded to a node for compilation. This generates an execution plan which is sent to the Query Execution components on one or more nodes for execution. The Query Execution components on each node are supported by a local run time system, identical to that found on the external database clients, which accesses and caches objects as they are required, making use of the Global Object Access component to locate the objects. The question of how objects are located is key, and is discussed in the next section.

If there are CORBA based clients generating requests for the execution of object methods then the requests will be directed (by the ORB) to a CN. In order to maximise the proportion of local object accesses (which are more efficient than remote object accesses) then the CN will route the request to the node which stores the object for execution. However, there will be circumstances in which this will lead to an imbalance in the loading on the nodes, for example if one object frequently occurs in the requests. In these circumstances, the CN may choose to direct the request to another, more lightly loaded, node.

4.1.2 Persistent Object Access

In this section we describe the major issues in locating objects. These include the structure of object identifiers (OIDs) and the mechanism by which an OID is mapped to a persistent object identifier (PID). A PID is the physical address of the object, i.e. it includes information on the node, disk volume, page and offset at which the object can be accessed.

Based on the requirements given in Section 3, the criteria which an efficient object access scheme for a parallel server must meet are:

- a low OID to PID mapping cost. This could be a significant contribution to the total cost of an object access and is particularly important as the Communications Nodes (which are only a subset of the nodes in the system) must perform this mapping for all the object requests from external ODBMS and CORBA clients. We want to minimise the number of CNs as they will to be more expensive than other nodes, and because the maximum number of CNs in a parallel platform may be limited for physical design reasons. The OID to PID mapping will also have to be performed on the other nodes when the Query Execution components access objects.
- minimising the number of nodes involved in an object access as each inter-node message contributes to the cost of the access.
- ability to re-cluster objects if access patterns change.
- ability to re-partition sets of objects over a different number of nodes and disks to meet changing performance requirements.
- a low object allocation cost.

We now consider a set of possible options for object access and consider how they match up against these criteria.

4.1.2.1 Physical OID schemes

In this option, the PID is directly encoded in the OID. For example, the OID could have the structure:

Node	Disk Volume	Page	Offset

The major advantage of this scheme is that the cost of mapping the OID to the PID is zero. The Communications Nodes (CNs) only have to examine the OIDs within incoming object requests from external clients and forward the request to the node contained in the OID. This is just a low-level routing function and could probably be most efficiently performed at a low level in the communications stack. Therefore, the load on a CN incurred by each external object access will be low. Similarly, object accesses generated on a node by the Query Execution component can be directed straight to the node which owns the object. Therefore this scheme does minimise the number of nodes involved in an access. However, because the OID contains the exact physical location of the object, it is not possible to move an object to a different page, disk volume or node. This rules out re-clustering or re-partitioning.

When an object is to be created then the first step is to select a node. This could be done by the CN using a round-robin selection scheme to evenly spread out the set of objects. The object is then created on the chosen node. Therefore object creation is a low cost operation.

Therefore, this scheme will perform well in a stable system but as it does not allow re-partitioning or re-clustering, it is likely that over time the system will become de-tuned as the server's workload changes.

We now consider three possible variations of the scheme which attempt to overcome this problem:

Indirections. The OID scheme used by O2 [7] includes a physical volume number in the OID, and so runs into similar problems if an Object is to be moved to a different volume. The O2 solution is to place an indirection at the original location of the object pointing to the new location of the object. Therefore, potentially two disk accesses might be incurred and (in a parallel server) two nodes might be involved in an object access. Also, this is not a viable solution for re-partitioning a set of objects across a larger number of disks and nodes in order to increase the aggregate throughput to those objects. The initial accesses to all the objects will still have to go to the indirections on the original disks/nodes and so they will still be a bottleneck. For this reason we disregard this variant as a solution.

Only include the Physical Node in the OID. In this variant, the OID structure would contain the physical node address, and then a logical address within the node, i.e. :

Node	Logical address within node

The logical address would be mapped to the physical address at the node; methods for this include hash tables and B-trees (the Direct Mapping scheme[16] is however not an option as it is not scaleable: once allocated, the handles which act as indirections to the objects cannot be moved). This would allow re-clustering within a node by updating the mapping function, and re-clustering across nodes using indirections, but it would not support re-partitioning across more nodes as the Node address is still fixed by the OID. Therefore we disregard this variant as a solution.

Update OIDs when an Object is Moved. If this could be implemented efficiently then it would have a number of advantages. Objects could be freely moved within and between nodes for re-clustering and re-partitioning, but because their OID would be updated to contain their new physical address then the mapping function would continue to have zero cost, and no indirections would be required. However, such a scheme would place restrictions on the external use of OIDs. Imagine a scenario in which a client application acquires an OID and stores it. The parallel ODBMS then undergoes a re-organisation which changes the location of some objects, including the one whose OID was stored by the client application. That OID is now incorrect and points to the wrong physical object location. A solution to this problem is to limit the temporal scope of OIDs to a client's database session. The ODMG standard allows objects to be named, and provides a mapping function to derive the object's OID from its name. It could be made mandatory for external database clients to only refer to objects by name outside of a database session. Within the session the client would use the mapping function to acquire the OID of an object. During the session, this OID could be used, but when the session ended then the OID would cease to have meaning and at the beginning of subsequent sessions OIDs would have to re-acquired using the name mapping function. This protocol leaves the database system free to change OIDs while there are no live database sessions running. When an object is moved then the name mapping tables would be updated. Any references to the object from other objects in the database would also have to be updated. If each reference in a database is bi-directional then this process is simplified, as all references to an object could be directly accessed and updated. The ODMG standard does permit unidirectional references but these can still be implemented bi-directionally. Another option is to scan the whole database to locate any references to the object to be moved. In a large database then this could be prohibitively time-consuming. An alternative is the Thor indirection based scheme [10], in which, over a period of time, references are updated to remove indirections.

A name based external access scheme could be used for CORBA client access to objects. Objects could be registered with an ORB by name. Therefore, incoming requests from CORBA based clients would require the object name to be mapped to an OID at a CN. A CORBA client could also navigate around a database so long as it started from a named object, and provided that the OIDs of any other objects that it reached were registered with the ORB. However, if the client wished to store a reference to an object for later access (outside the current session) then it would have to ask the ODBMS to create a name for the object. The name would be stored in the mapping tables on the CNs, ready for subsequent accesses. This procedure can, for example, be used to create interlinked, distributed databases in which references in one database can refer to objects in another.

4.1.2.2 Logical OID schemes

At the other extreme from the last scheme is one in which the OID is completely logical and so contains no information about the physical location of the object on disk. Consequently, each object access requires a full logical to physical mapping to determine the PID of the object. Methods of performing this type of mapping have been extensively studied with respect to a serial ODBMS [16], however their appropriateness for a parallel system requires further investigation.

As stated above, the logical to physical mapping itself can be achieved by a number of methods including hash tables and B-trees. Whatever method is used for the mapping, the Communication Nodes will have to perform the mapping for requests received by external clients. This may require disk accesses as the mapping table for a large database will not fit into main memory. Therefore there will be a significant load on the CNs, and it will be necessary to ensure that there are enough CNs configured in a system so that they are not bottlenecks. Similarly, the nodes executing OQL will also have to perform the expensive OID to PID mapping. However, once the mapping has been carried out then the request can be routed directly to the node on which the object is located.

As all nodes will need to perform the OID to PID mapping, then each will need a complete copy of the mapping information. We rule out the alternative solution of having only a subset of the nodes able to perform this mapping due to the extra message passing latency that this would add to each object access. This has two implications: each node needs to allocate storage space to hold the mapping; and whenever a new object is created then the mapping information in all nodes needs to be updated, which results in a greater response time, and worse throughput for object creation than was the case for the Physical OID scheme described in the last section.

With this scheme, unlike the Physical OID scheme, names are not mandatory for storing external references to objects. As OIDs are unchanging, they can be used both internally as well as externally. This removes the need for CNs to map names to OIDs for incoming requests from CORBA clients.

4.1.2.3 Virtual Address Schemes

There have been proposals to exploit the large virtual address spaces available in some modern processors, in object database servers, both parallel [17] and serial [18]. Persistent objects are allocated into the virtual address space and their virtual address is used as their OID. This has the benefit of simplifying the task of locating a cached object in main memory. However an OID to PID mapping mechanism is still required when a page fault occurs and the page containing an object must be retrieved from disk. The structure of a typical virtual address is:

Segment	Page	Offset

In a parallel ODBMS, when a page fault occurs then the segment and page part of the address are mapped to a physical page address. The page is fetched, installed in real memory and the CPU's memory management unit tables are configured so that any accesses to OIDs in that page are directed to the correct real memory address.

The key characteristic of this scheme is that the OID to PID mapping is done at the page level, i.e. in the [Segment, Page, Offset] structure shown above only the Segment and Page fields are used in determining the physical location of the object. Therefore it is possible to re-partition sets of objects by changing the mapping, but it is not possible to re-cluster objects within pages unless changing OIDs, or indirections are supported (as described in Section 4.1.2.1).

The OID to PID mapping can be done using the logical or physical schemes described earlier, with all their associated advantages and disadvantages. However, it is worth noting that if a logical mapping is used then the amount of information stored will be smaller than that required for the logical OID scheme of Section 4.1.2.2 as only pages, rather than individual objects must be mapped.

4.1.2.4 Logical Volume Scheme

None of the above schemes is ideal, and so we have also investigated the design of an alternative scheme which allows the re-partitioning of data without the need to update addresses nor incur the cost of an expensive logical to physical mapping in order to determine the node on which an object is located. In describing the scheme we will use the term *volume set* to denote the set of disk volumes across which the objects in a class are partitioned. OIDs have the following structure:

Class	Logical Volume	Body

The *Class* field uniquely identifies the class of the object. As will be described in detail later, the *Logical Volume* field is used to ensure that objects which must be clustered are always located in the same disk volume, even after re-partitioning. The size of this field is chosen so that its maximum value is much greater than the maximum number of physical volumes that might exist in a parallel system.

Each node of the server has access to a definition of the Volume Set for each class. This allows each node, when presented with an OID, to use the *Class* field of the OID to determine the number and location of the physical volumes in the Volume Set of the class to which the object belongs.

If we denote the cardinality of Volume Set V as |V|, then the physical volume on which the object is stored is calculated cheaply as:

$$\text{Physical Volume} = V[\ \text{LogicalVolume modulus } |V|\ ]$$

where the notation V[i] denotes the i'th Volume in the Volume Set V. A node wishing to access the object uses this information to forward the request to the node on which the object is held. On that node, the unique serial number for the object within the physical volume can be calculated as:

$$\text{Serial} = (\text{Body div } |V|) + |\text{Body}| * (\text{LogicalVolume div } |V|)$$

where: a div b is the whole number part of the division of a by b, and |Body| is the number of unique values that the body field can take.

The Serial number can then be mapped to a PID using a local logical to physical mapping scheme.

If it is necessary to re-partition a class then this can be done by creating a new volume set with a different cardinality and copying each object from the old volume set into the correct volume of the new volume set using the above formula. Therefore a class of objects can be re-partitioned over an arbitrary set of volumes without having to change the OIDs of those objects.

The main benefit of the scheme is that objects in the same class with the same *Logical Volume* field will always be mapped to the same physical volume. This is still true even after re-partitioning, and so ensures that objects clustered before partitioning can still be clustered afterwards. Therefore, when objects which must be clustered together are created they should be given OIDs with the same value in the *Logical Volume* field. To ensure that the OID is unique they must have different values for their *Body* fields.

Where it is desirable to cluster objects of different classes then this can be achieved by ensuring that both classes share the same volume set, and that objects which are to be clustered share the same *Logical Volume* value. This ensures that they are always stored on the same physical volume.

When an external client application needs to create a new object then it sends the request, including the class of the object, to a Communication Node. These hold information on all the volume sets in the server, and so can select one physical volume from that set (using a round-robin or random algorithm). The CN then forwards the `create object' request to the node that contains the chosen physical volume, and the OID is then generated on that node as follows. Each node contains the definition of the volume set for each class. The required values of the Class and Logical Volume fields of the object will have been sent to the node by the CN. The node also keeps information on the free *Body* values for each of the logical volumes which map to a physical volume connected to that node. The node can therefore choose a free *Body* value for the Logical Volume chosen by the CN. This completes the OID. The object is then allocated to a physical address and the local OID mapping information updated to reflect this new OID to PID mapping.

The creation algorithm is different in cases where it is desirable to create a new object which is to be clustered with an existing object. The new object must be created with the same *Logical Volume* field setting as the existing object so that it will always by stored on the same Physical Volume as the existing object, even after a re-partition. Therefore it is this Logical Volume value which is used to determine the node on which the object is created. On that node the Body field of the new object will be selected, and the object will be allocated into physical disk storage clustered with the existing object (assuming that space available).

This scheme therefore allows us to define a scaleable, object manager in which it is possible to re-partition a set of objects in order to meet increased performance requirements. Existing clustering can be preserved over re-partitioning of the data, however if re-clustering is required then it would have to be done by either using indirections or by updating OIDs. Determining the node on which an object resides is cheap and this should reduce the time required on the CNs to process incoming object access requests. The vast bulk of the information required to map an OID to a PID is local to the node on which the object is stored, so reducing the total amount of storage in the system required for OID mapping, and reducing the response time and execution cost of object creation.

4.1.2.5 Summary

The above schemes all have advantages and disadvantages:

- The Physical OID scheme has a low run time cost but our requirements for re-clustering and re-partitioning can only be met if OIDs can be changed. However, the use of names as permanent identifiers does permit this.
- The Logical OID system supports both re-partitioning and re-clustering. However it is expensive both for object creation and in terms of the load it places on the CNs.
- The Virtual Address scheme supports re-partitioning, but re-clustering is only possible through indirections or by changing OIDs. Creating an object is costly when a new page has to be allocated, as it requires an updating of the OID to PID mapping on all nodes. Also, the cost of mapping OIDs to PIDs for accesses from external clients may place a significant load on the CNs.
- The Logical Volume scheme may be a reasonable compromise between completely logical and completely physical schemes. However indirections or changeable OIDs are required for re-clustering.

We are therefore investigating these schemes further in order to perform a quantitative comparison between them.

4.2 Concurrency Control and Cache Management

Concurrency control is important in any system in which multiple clients can be accessing an object simultaneously. In our system, each node runs an Object Manager which is responsible for the set of local objects including: concurrency control when those objects are accessed; and the logging mechanisms required to ensure that it is possible to recover of the state of those objects after a node or system failure.

In the system we have described there are three types of clients of the Object Managers: database applications running on external clients; Query Execution components on the parallel server nodes; and, also on the nodes, object method calls instigated by CORBA based clients. Therefore there are caches both on the nodes of the server, and in the external database clients. More than one client may require access to a particular object and so concurrency control and cache coherency mechanisms must manage the fact that an object may be held in more than one cache. This situation is not unique to parallel systems, and occurs on any client-server object database system which supports multiple clients. A number of concurrency control/cache coherency mechanisms have been proposed for this type of system [19], and there appears to be nothing special about parallel systems which prevents one of these from being adopted.

4.3 Query Execution

In a parallel system we wish to be able to improve the response time of individual OQL queries by exploiting intra-query parallelism. In this section we discuss some of the issues in achieving this.

An OQL query generated by a client will be received first by a Communications Node which will use a load balancing scheme to choose a node on which to compile the query. Compilation generates an execution plan: a graph whose nodes are operations on objects, and whose arcs represent the flow of objects from the output of one operation to the input of another. As described earlier, the system we propose supports data shipping but for the reasons discussed in Section 2.2.1, it will give performance benefits if operations are carried out on local rather than remote objects where possible. To achieve this it is necessary to structure collections which are used to access objects (e.g. extents) so as to allow computations to be parallelised on the basis of location. One option [17] is to represent collections of objects through two-level structures. At the top level is a collection of sets of objects with one set per node of the parallel system. Each set contains only the objects held on one node. This allows computations on collections of objects to be parallelised in a straightforward manner: each node will run an operation which processes those objects stored locally. However, in our data shipping system if the objects in a collection are not evenly partitioned over the nodes then it is still possible to partition the work of processing the objects among the nodes in a way that is not based on the locations of the objects. One advantage of the Logical Volume OID mapping scheme described in Section 4.1.2.4 is that it removes the need to explicitly maintain these structures for class extents as the existing OID to PID mapping information makes it possible to locate all the objects in a given class stored on the local node.

A major difference between object and relational databases is that object database queries can include method calls. This has two major implications.

Firstly, as the methods must be executed on the nodes of the parallel server a mechanism is required to allow methods to be compiled into executable code that runs on the server. This is complicated by the fact that the ODMG standard allows methods to be written in a number of languages, and so either the server will have to provide environments to compile and execute each of these languages, or alternatively client applications using the system will have to be forced to specify methods only in those languages which the server can support. Further, the architecture that we have proposed in this paper has client applications executing method calls in navigational code on the client. Therefore executable versions of the code must be available both in the server (for OQL) and on the clients. This introduces risks that the two versions of the code will get out of step. This may, for example, occur if a developer modifies and recompiles the code on a client but forgets to do the same on the server. Also a malicious user might develop, compile and execute methods on the client specifically so as to access object properties not available through the methods installed in the server. Standardising on Java or another interpreted language for writing methods might appear to be a solution to these problems because they have a standard, portable executable representation to which a method could be compiled once and stored in the database before execution on either a client or server. However, the relatively poor performance of interpreted languages when compared to a compiled language such as C++ is a deterrent because the main reason for utilising a parallel server is to provide high performance.

Secondly, the cost of executing a method may be a significant component of the overall performance of a database workload, and may greatly vary in cost depending on its arguments. This could make it difficult to statically balance the work of executing a query over a set of nodes. For example, consider an extent of objects that is perfectly partitioned over a set of nodes such that each node contains the same number of objects. A query is executed which applies a method to each object but not all method calls take the same time, and so, if the computation is partitioned on the basis of object locality, some nodes may have completed their part of the processing and be idle while others are still busy. Therefore it may be necessary to adopt dynamic load balancing schemes which delay decisions on where to execute work for as long as possible (at run-time) so as to try to evenly spread the work over the available nodes. For workloads in which method execution time dominates and not all nodes are fully utilised (perhaps because the method calls are on a set of objects whose cardinality is less than the number of nodes in the parallel system) then it may be necessary to support intra-method parallelism, in which parallelism is exploited within the execution of a single method. This would require methods to be written in a language which was amenable to parallelisation, rather than a serial language such as C++ or Java. One promising candidate is UFO [20], an implicit parallel language with support for objects.

4.4 Performance Management

Section 3 outlined the requirements for performance management in a parallel database server. In this section we describe how we intend to meet these requirements.

In recent years, there has been an interest in the performance management of multimedia systems [21]. When a client wishes to establish a session with a multimedia server, the client specifies the required Quality of Service and the server decides if it can meet this requirement, given its own performance characteristics and current workload. If it can, then it schedules its own CPU and disk resources to ensure that the requirements are met. Ideally, we would like this type of solution for a parallel database server, but unfortunately it is significantly more difficult due to the greater complexity of database workloads. The Quality of Service requirements of the client of a multimedia server can be expressed in simple terms (e.g. throughput and jitter), and the mapping of these requirements onto the usages of the components of the server is tractable. In contrast, whilst the requirements of a database workload may be specified simply, e.g. response time and throughput, in many cases the mapping of the workload onto the usage of the resources of the system cannot be easily predicted. For example, the CPU and disk utilisation of a complex query are likely to depend on the state of the data on which the query operates.

We have therefore decided to investigate a more pragmatic approach to meeting the requirements for performance management in a parallel database server which is based on priorities [11]. When an application running on an external client connects to the database then that database session will be assigned a priority by the server. For example, considering the examples given in Section 3, the sessions of clients in an OLTP workload will have high priority, the sessions of clients generating complex queries will have medium priority while the sessions of agents will have low priority. Each unit of work - an object access or an OQL query - sent to the server from a client will be tagged with the priority and this will be used by the disk and CPU schedulers. If there are multiple units of work, either object accesses or fragments of OQL available for execution, then they will be serviced in the order of their priority, while pre-emptive round-robin scheduling will be used for work of the same priority.

5 Conclusions

The aim of the *Polar* project is to investigate the design of a parallel, ODMG compatible ODBMS. In this paper we have highlighted the system requirements and key design issues, using as a starting point our previous experience in the design and usage of parallel RDBMS. We have shown that differences between the two types of database paradigms lead to a number of significant differences in the design of parallel servers. The main differences are:

- rows in RDBMS tables are never accessed individually by applications, whereas objects in an ODBMS can be accessed individually by their unique OIDs. The choice of OID to PID mapping is therefore very important. A number of schemes for structuring OIDs and mapping them to PIDs were presented and compared. None is ideal, and further work is needed to quantify the differences between them.
- there are two ways to access data held in an object database: through a query language (OQL), and by directly mapping database objects into client application program objects. In a relational database, the query language (SQL) is the only way to access data. A major consequence of this difference is that parallel RDBMS can

utilise either task shipping or data shipping, but task shipping is not a viable option for a parallel ODBMS with external clients.

- object database queries written in OQL can contain arbitrary methods (unlike RDBMS queries) written in one of several high level programming languages. Mechanisms are therefore required in a parallel ODBMS to make the code of a method executable on the nodes of the parallel server. This will complicate the process of replacing an existing serial ODBMS, in which methods are only executed on clients, with a parallel server. It also raises potential issues of security and the need to co-ordinate the introduction of method software releases. Further, as methods can contain arbitrary code, estimating execution costs is difficult and dynamic load-balancing, to spread work evenly over the set of nodes, may be required.

We have also highlighted the potential role for parallel ODBMS in distributed systems and described the corresponding features that we believe will be required. These include performance management to share the system resources in an appropriate manner, and the ability to accept and load-balance requests from CORBA clients for method execution on objects.

5.1 Future Work

Having carried out the initial investigations described in this paper we are now in a position to explore the design options in more detail through simulation and the building of a prototype system. We are building the prototype in a portable manner so that, as described in Section 3 we can compare the relative merits of a custom parallel machine and one constructed entirely from commodity hardware.

Our investigations so far have also raised a number of areas for further exploration, and these may influence our design in the longer term:

- More sophisticated methods of controlling resource usage in the parallel server are needed so as to make it possible to guarantee to meet the performance requirements of a workload. The solution based on priorities, described in Section 3, does not give as much control over the scheduling of resources as is required to be able to guarantee that the individual performance requirements of a set of workloads will all be met. Therefore we are pursuing other options based on building models of the behaviour of the parallel server and the workloads which run on it. Each workload is given the proportion of the systems resources that it needs to meet its performance targets.
- If it is possible to control the resource usage of database workloads then, in the longer term, it would be desirable to extend the system to support continuous media. Currently multimedia servers (which support continuous media) and object database servers have different architectures and functionality, but there would be many advantages in unifying them, for example to support the high-performance, content-based searching of multimedia libraries.

- The ODMG standard does not define methods of protecting access to objects equivalent to that found in RDBMS. Without such a scheme, there is a danger that clients will be able to access and update information which should not be available to them. This will be especially a problem if the database is widely accessible, for example via the Internet. Work is needed in this area to make information more secure from both malicious attack and programmer error.

Acknowledgements

The *Polar* project is a collaboration between research groups in the Department of Computing Science at the University of Newcastle, and the Department of Computer Science at the University of Manchester. We would like to thank Francisco Pereira and Jim Smith (Newcastle), and Norman Paton (Manchester) for discussions which have contributed to the contents of this paper. The *Polar* project is supported by the UK Engineering and Physical Science Research Council through grant GR/L89655.

References

1. Watson, P. and G.W. Catlow. *The Architecture of the ICL Goldrush MegaServer*. in *BNCOD13*. 1995. Manchester, LNCS 940, Springer-Verlag.
2. Loomis, M.E.S., *Object Databases, The Essentials*. 1995: Addison-Wesley.
3. Cattell, R.G.G., ed. *The Object Database Standard:ODMG 2.0*. , Morgan Kaufman.
4. Watson, P. and P. Townsend, *The EDS Parallel Relational Database System*, in *Parallel Database Systems*, P. America, Editor. 1991, LNCS 503, Springer-Verlag.
5. Watson, P. and E.H. Robinson, *The Hardware Architecture of the ICL Goldrush MegaServer.* Ingenuity- The ICL Technical Journal, 1995. **10**(2): p. 206-219.
6. Watson, P., M. Ward, and K. Hoyle. *The System Management of the ICL Goldrush Parallel Database Server*. in *HPCN Europe*. 1996: Springer-Verlag.
7. Bancilhon, F., C. Delobel, and P. Kanellakis, eds. *Building an Object-Oriented Database System : The Story of O2*. 1992, Morgan Kaufmann.
8. van den Berg, C.A., *Dynamic Query Processing in a Parallel Object-Oriented Database System*, 1994, CWI, Amsterdam.
9. Carey, M. et al., *Shoring up persistent applications*. in *1994 ACM SIGMOD Conf.* 1994. Mineapolis MN.
10. Day, M., *et al.*, *References to Remote Mobile Objects in Thor.* ACM Letters on Programming Languages and Systems, 1994.
11. Rahm, E. *Dynamic Load Balancing in Parallel Database Systems*. in *EURO-PAR 96*. 1996. Lyon: Springer-Verlag.

12. Tamer Ozsu, M. and P. Valduriez, *Distributed and parallel database systems.* ACM Computing Surveys,, 1996. **28**(1).
13. Hilditch, S. and C.M. Thomson, *Distributed Deadlock Detection: Algorithms and Proofs*, UMCS-89-6-1, 1989, Dept. of Computer Science, University of Manchester.
14. OMG, *The Common Object Request Broker: Architecture and Specification.* 1991: Object Management Group and X/Open.
15. Gerlhof, C.A., *et al.*, *Clustering in Object Bases*, TR 6/92. 1992, Fakuly for Informatik, University Karlsruhe.
16. Eickler, A., C.A. Gerlhof, and D. Kossmann. *A Performance Evaluation of OID Mapping Techniques*. in *VLDB*. 1995.
17. Gruber, O. and P. Valduriez, *Object management in parallel database servers*, in *Parallel Processing and Data Management*, P. Valduriez, Editor. 1992, Chapman & Hall. p. 275-291.
18. Singhal, V., S. Kakkad, and P. Wilson, *Texas: An Efficient, Portable Persistent Store*, in *Proc. 5th International Workshop on Persistent Object Stores*. 1992. p. 11-13.
19. Franklin, M.J., M.J. Carey, and M. Livny, *Transactional client-server cache consistency: alternatives and performance.* ACM Transactions on Database Systems, 1997. **22**(3): p. 315-363.
20. Sargeant, J., *Unified Functions and Objects: an Overview*, UMCS-93-1-4, 1993, University of Manchester, Department of Computer Science.
21. Adjeroh, D.A. and K.C. Nwosu, *Multimedia Database Management - Requirements and Issues.* IEEE Multimedia, 1997. **4**(3): p. 24-33.