Polar: An Architecture for a Parallel ODMG Compliant Object Database

Jim Smith
and Paul Watson
Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU UK
Jim.Smith@ncl.ac.uk
Paul.Watson@ncl.ac.uk

Sandra de F. Mendes Sampaio
and Norman Paton
Department of Computing Science
University of Manchester
Oxford Road
Manchester, M13 9PL UK
sampaios@cs.man.ac.uk
norm@cs.man.ac.uk

ABSTRACT

Object database management systems (ODBMS) are now established as the database management technology of choice for a range of challenging data intensive applications. Furthermore, the applications associated with object databases typically have stringent performance requirements, and some are associated with very large data sets. However, despite the demands made on object databases by applications, there has been surprisingly little work on parallel object databases. This paper presents the architecture and some preliminary performance results for the Polar ODMG compliant parallel object database. The architecture described has been implemented in a shared-nothing environment on a network of PCs. The paper describes how OQL queries are compiled, parallelized and executed in this environment, and includes some preliminary performance results for OQL queries using the 007 benchmark.

1. INTRODUCTION

There are some fields in which ODBMS have been used for many years, such as Geomatics and various forms of engineering design where they are preferred for their ability to manage data with complex structure [20]. As the volume of data stored increases and the demands of complex analyses grow, the need for scaleable object database servers seems sure to become ever greater. Additionally, several interesting new areas are currently emerging. For example, ODBMS are now being adopted for the storage of vast amounts of scientific information, such as in the Digital Sky [23] (astronomical information) and CERN RD45 [24] (high energy physics) projects which are anticipated to require the ability to store and manipulate up to Petabytes of object data, something that is clearly not feasible with current, single site ODBMS. In newly developing applications it is hard to anticipate the requirements but the dual navigational and declarative access capability seems intuitively attractive. Other areas where ODBMS are being seen as well suited include bioinformatics, e.g. [22]. The use of object technology to build Internet applications will also create demands for the storage and manipulation of huge numbers of objects. The need to support data mining for extracting useful information from this data will add to the performance requirements.

Surprisingly, while a number of research groups and companies now provide object database servers designed to run on uniprocessors, there has been very little work done towards the use of parallelism to provide scalable performance in ODBMS. The growth of an ODBMS industry has however led to the generation of the ODMG standard [7]. Such a standard not only allows the Polar project to focus on research into parallelism, rather than on the ODBMS interfaces, but also eases the transition to a parallel ODBMS for existing users of ODMG compliant databases.

Over the last 15 years, there has been a great deal of effort expended in designing parallel relational database servers (RDBMS), and these now have the largest share of the entire commercial parallel systems market. In designing a parallel ODBMS we wanted to build on the experience of the existing parallel RDBMS research where possible. A key lesson arising from parallel RDBMS development is that the shared nothing architecture, where neither disk nor memory is common between processors is highly scalable [10]. Therefore, Polar currently focusses on the shared nothing environment and on the realisation of a parallel algebra to fully exploit it. However, this does not prohibit exploitation of shared memory parallelism within each node of the parallel machine in the future.

Our work has shown that the significant differences between the object and relational database paradigms lead to significant differences in the designs of parallel servers to support these two paradigms. The description of the Polar system presented in the rest of this paper shows how these differences impact the design of a parallel ODBMS. Significant differences include:

1. An ODBMS automatically allocates each separately persistent object an OID which allows direct pointer style accesses as well as the value based accesses familiar in an RDB. OIDs are fundamental to support-

ing navigation, but they also support path expressions within queries and so imply a need for new algebra operators. In a parallel setting, traversing a relationship can imply traversing between processors of the parallel machine which has implications for performance.

2. An ODBMS differs also from a traditional RDBMS in supporting collection valued attributes, which are similar to nested relations in extended relational systems but may contain references to implement many to one and many to many relationships. Such collections also necessitate enhancements in the algebra and query planning.

3. Relational databases can only be accessed through a query language, whereas object databases can be accessed both through a query language, and also by mapping database objects into applications in support of navigation. The presence of the two different styles of access inevitably complicates both concurrency control and load management.

4. Unlike traditional relational systems, object relational systems [26] include support for user defined functions, implemented either as subqueries in the declarative language or in a procedural language. Calls to such functions may be included in queries. An ODBMS has similar functionality in its support for objects having methods except that in the case of an ODBMS a method can navigate arbitrarily through other objects, as can code executing in a navigational client. As described above however, traversing a relationship in a parallel setting can imply the cost of traversing between processors of the parallel machine. The widespread appeal of OO languages, which operate in the navigational paradigm, suggests that benefits are seen in organising complex applications as objects with navigational methods. However supporting calls to such methods which navigate in declarative queries must be expected to increase the difficulty of planning and executing those queries.

Initial work in Polar focusses on the realisation of parallel execution of OQL queries without method calls, but using components that enable enhancement to include other attributes of an ODBMS.

In the rest of this paper we describe the design of the Polar parallel ODBMS which is novel in supporting ODMG on a shared nothing architecture with disk based data. We present performance results for a set of parallel queries measured on a PC cluster that show reasonable speedups can be obtained in this low cost environment. Amongst these results we present measurements that demonstrate the complexity of issues surrounding the processing of relationships; an important issue in ODBMS. Section 2 describes how our work relates to that of others. The Polar architecture is then described in Sections 3 to 5. Section 6 gives the results measured on the Polar prototype and analysis described above. Finally in Section 7 we draw conclusions from our work so far.

2. RELATED WORK

The previous parallel RDBMS projects that have most influenced our work are EDS and Goldrush. In the 1980s, the EDS project [30] designed and implemented a complete parallel system including hardware, operating system and a database server that was basically relational, but did contain some object extensions. This ran efficiently on up to 32 nodes. The ICL Goldrush project [29] built on these results and designed a parallel RDBMS product that ran parallel Oracle and Informix. Issues tackled in these two projects that are relevant to the parallel ODBMS include concurrency control in parallel systems, scalable data storage, and parallel query processing.

Both of these projects used custom-built parallel hardware. In Polar we are investigating an alternative, which is the use of lower-cost commodity hardware: a cluster of PCs.

Research in parallel object databases can probably be considered to belong within two principal areas – the development of object database models and query languages specifically for use in a parallel setting, and techniques to support the implementation of object models and query processors in a parallel setting. A thorough discussion of the issues of relevance to the development of a parallel object database server is given in [28].

An early parallel object database project was Bubba [4], which had a functional query language FAD. Although the Bubba model and languages probably influenced the later ODMG model, FAD provides more programming facilities than OQL. There has also been a significant body of work produced at the University of Florida [27, 8], both on language design and query processing algorithms. However, the language on which this is based [27] seems less powerful than OQL, and the query processing framework is significantly different; it is not obvious to us that it can be adapted easily for use with OQL. Another parallel object-based database system is PRIMA [13], which uses the MAD data model and a SQL-like query language, MQL. The PRIMA system's architecture differs considerably from Polar's, as it is implemented as a multiple-level client-server architecture, where parallelism is exploited by partitioning the work associated with a query into a number of service requests that are propagated through the layers of the architecture in the form of client and sever processes. However, the mapping of processes onto processors is accomplished by the operating system of the assumed shared memory multiprocessor. Translating the PRIMA approach to the shared nothing environment assumed in Polar appears difficult.

There has been relatively little work on parallel query processing for mainstream object data models. The one other parallel implementation of an ODMG compliant system we are aware of is Monet [3]. This shares with Polar the use of an object algebra for query processing, but operates over a main-memory storage system based on vertical data fragmentation which is very different from the Polar storage model. As such, Monet really supports the ODMG model through a front-end to a binary relational storage system.

There has also been work on parallel query processing in object relational databases [21]. However, object relational databases essentially operate over extended relational storage managers, so results obtained in the object relational setting may not transfer easily to an ODMG setting.

3. ARCHITECTURAL OVERVIEW

The Polar architecture involves a number of query compiler/optimiser systems running on a subset of the processors in a parallel machine. These systems share an ob-

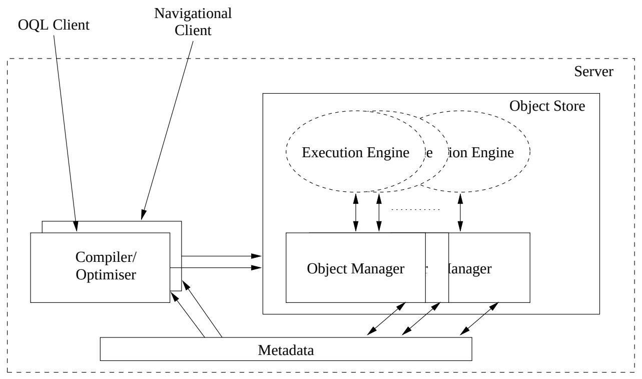

Figure 1: Overview of query processing.

ject store which is distributed over the remaining processors of the parallel machine. Persistent objects are distributed across the disks mounted on the the processors comprising the object store. Within the object store, each processor runs an object manager. The object manager provides services to a local copy of the execution engine which implements the physical algebra operators. The compiler/optimiser and object store share a common description of both system and application level metadata.

Figure 1 shows compilers processing OQL queries input in the one case via a command line interface and in another case generated by a program implemented in a bound language. In either case, the compiler/optimiser produces a query plan which specifies the distribution and interconnection of operators over the available processors. A query plan is comprised of a number of partitions, one for each processor which is to participate in the query. The partitions are communicated to the object store where they are instantiated as threads executing within the query execution engine on each appropriate processor.

The operators are implemented according to the *iterator model* [15] whereby each implements a common interface comprising the functions *open*, *next* and *close* allowing creation of arbitrary pipelines. As well as performing manipulations of the data propagating up the query tree, operators themselves call on runtime services to perform various tasks. These tasks include spawning concurrent threads of execution, transmitting data to and from other operators and accessing local application data.

Inter query parallelism may be obtained by running independent queries concurrently in separate partitions of the parallel object store. *Intra query* parallelism is obtained through the parallel execution of a single query on more than one processor. In this work the focus is on the exploitation of *intra query* parallelism in the execution of a single OQL query. *Intra query* parallelism may be classified further into: *inter operator* parallelism which is obtained through parallel execution of separate algebra operators in a single query tree on multiple processors, and *intra operator* parallelism which is obtained through executing a single operator on multiple processors in data parallel fashion. Polar exploits both forms of parallelism.

Two general approaches for parallelising a database query execution engine are described in [15], namely the *bracket model* and the *operator model*. In the bracket model, a generic process template is used by the physical operators for receiving and sending data. In the operator model, parallelism related operators are inserted into a sequential plan, transforming it into a parallel plan. Polar adopts the operator model with the intention of achieving a cleaner separation between data manipulation and control functionality at the level of the physical operators. The exchange operator, modelled after that described in the Volcano system [14], is used to implement a partition between two threads of execution, and a configurable data redistribution, the latter implementing a flow control policy.

4. QUERY OPTIMISATION AND PARALLELISATION

As illustrated in Figure 2, a declarative query is passed

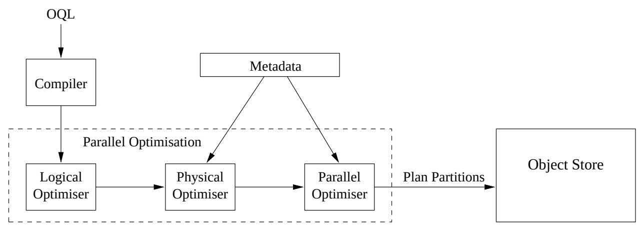

Figure 2: Functional components of the parallel optimiser.

to the server where, the compiler/optimiser produces a plan which specifies the distribution and interconnection of operators over the available processors and passes this plan, as a set of partitions, to the object store for instantiation and execution. The compilation of queries is discussed in detail in the rest of this section. Their execution is described in the section 5.

The key design decisions that have shaped the parallel OQL query compiler for OQL are:

1. The parallel optimiser is an extension of an existing non-parallel optimiser for OQL [11]. It uses a two phase approach, in which non-parallel aspects of optimisation take place before parallel aspects, allowing reuse of well established techniques for non-parallel optimisation.

2. The optimiser is modular, in that components have been designed with well defined interfaces, so that alternative approaches can be experimented with during the project. The main internal representation used for queries is an object algebra, which is refined during the optimisation process to include increasingly detailed execution descriptions.

The principal components of the query compilation architecture are illustrated in Figure 2. The parallel optimiser is further divided into components that perform partitioning and processor allocation. The following sections describe each component of the parallel optimisation system.

4.1 Logical optimiser

During compilation, the input OQL query expression is translated into a *Monoids* calculus [12] expression for normalisation. The main advantage of the Monoids calculus is

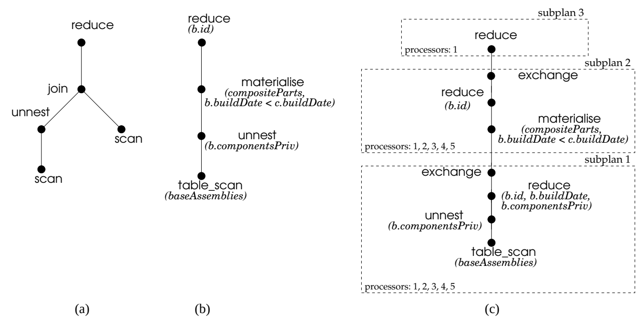

Figure 3: An example query: (a) Logical algebra. (b) Physical algebra. (c) Parallel algebra.

that it allows a uniform treatment for collections and scalar types, simplifying the design of the logical algebra [16]. The normalisation of the Monoids calculus includes query unnesting, fusion of multiple selection operations into a single selection operation, normalisation of predicates by applying DeMorgan's laws, etc. Subsequently, the normalised calculus expression is mapped into a logical algebra expression. A detailed description of the algebra, which is an extension of that proposed in [16] is provided in [9]. Figure 3(a) shows a logical plan for the query shown below. This query corresponds to Query 5 of the OO7 benchmark (described in section 6).

select distinct b.id
from b **in** baseAssemblies, c **in** b.componentsPriv
  where b.buildDate $<$ c.buildDate;

Optimisations over the logical algebra expressions include reordering of joins and moving of selections as close as possible to the scan operators. The logical optimiser is heuristic, rather than cost based, and generates several logical expressions as output for each query, though many fewer than all possible expressions.

4.2 Physical optimiser

The physical optimiser transforms each logical expression into a physical plan, by choosing an execution strategy (i.e. physical operators) for each logical operator. Examples of physical operators include nested_loop, hash_join and materialise for logical join, table_scan and index_scan for logical scan, etc. The physical optimiser selects one or several of the physical operators available for each logical optimiser, and generates several plans using cost-based techniques. Note that the cost model used by the physical optimiser does not at this stage take into account the effects of parallelism on the cost of the different plans. Figure 3(b) shows a physical plan for the logical plan in figure 3(a).

Table 1 indicates the relationship between the logical and physical operators.

4.3 Parallel optimiser

The parallel optimiser transforms a sequential physical plan into a parallel plan, by rewriting the plan in a parallel algebra. As mentioned earlier, this is achieved through the insertion of instances of the exchange operator. Since this operator is associated with data communication among different processors of a parallel machine, the insertion of this operator in a conventional query plan divides the plan into different sets of operators, i.e. different partitions, or subplans.

Table 1: Logical and physical algebras.

Logical Operators	*Physical Operators*
get	table_scan, index_scan
join	nested_loop, hash_join, materialise
nest	nest, groupby
unnest	unnest
	sort
reduce	reduce
union	union
map	map

There are many possible ways in which the exchange operator can be inserted into a query plan, and, therefore, there are many ways in which to partition a query plan. Moreover, there are many possible ways of allocating machine resources for executing each subplan of a parallel plan. Partitioning and processor allocation are described below.

4.3.1 *Partitioning of Plans*

A simple criterion for partitioning a query plan into a number of subplans that can be executed in parallel using some type of parallelism is by placing two data dependent operators in different nodes of the parallel machine if *data repartitioning* is required for the proper execution of the consumer operator. [18] distinguishes *attribute sensitive* operators and *attribute insensitive* operators. An attribute sensitive operator, when executed on multiple processors, requires its input data to be partitioned by a distinguished attribute, whereas an attribute insensitive operator does not. An exchange operator must be placed before an attribute sensitive operator if any of its child operators use different partitioning attributes, so that the data is partitioned by the required attributes.

Grouping operators and valued-based join operators such as nested_loop are examples of attribute sensitive operators, as they require partitioning by the joining and grouping attributes, respectively. Table 2 classifies the parallel algebra operators as attribute sensitive or attribute insensitive.

Table 2: Operators classification.

Operator Characteristic	*Operator*
Attribute Sensitive	nested_loop, hash_join, nest, groupby, materialise
Attribute Insensitive	reduce, sort, unnest, union, map, exchange

Figure 3(c) illustrates a possible way of inserting the parallelism related operator into the plan shown in figure 3(b). The operators table_scan and index_scan are usually parallelised based on the partitioning of the data being read. Thus, even if an attribute sensitive predicate is specified in any of these scan operations, no exchange operator is required to do data repartitioning. The operator materialise, first proposed in [2], is mainly responsible for materialising path expressions by bringing into scope objects of collections referred to in a path expression. The operator reduce is considered as attribute insensitive, as its role is simply to structure the results of a query. However, an exchange operator may be necessary to channel data from different nodes to the node that hosts the reduce operator, so that aggregate operations are performed and duplicates removed.

The parallel optimiser may choose to insert additional reduce operators onto the plan in order to discard data that is not relevant to the next steps of execution, saving the cost of distributing unnecessary data over multiple processors of the machine. For example, a reduce operator is placed between unnest and exchange in figure 3(c), to discard attributes of the objects of baseAssemblies that are not relevant to subsequent operators in the query execution.

The partitioning strategy described above is naive in the sense that exchange operators are placed right before attribute sensitive operators, without any previous analysis about exactly in what position before an attribute sensitive operator it is least costly to perform data repartitioning. In some cases, a less costly repartitioning can be obtained by moving an exchange operator to a position right above an operation that discards data, so that fewer data values are moved during repartitioning. A partitioning strategy that analyses the best position for performing data repartitioning in a plan tree is described in [17].

4.3.2 Processor Allocation

Depending on the number of processors in a parallel machine, there may be many possible ways of assigning a set of processors to execute each partition of a plan. A compile-time parallel optimiser does not consider runtime information such as current load on processors when assigning processors to subqueries. Thus, such optimisers have to rely on other criteria to make their decisions. The criteria used for processor allocation in the current version of the parallel optimiser are:

- Allocate scan operations based on data location and data distribution, so that only processors with relevant data are assigned to execute scans.
- Try to partition the other (non-scan) operators over the processors, in such a way that the processors receive approximately the same load. At the moment, the load resulting from other running tasks is not taken into account.
- Try to use the same set of processors when data repartitioning is required, increasing the chance of a number of tuples (output from the producer operators) not having to be moved across nodes (to the consumer operators) and, therefore, saving communication costs.

Figure 3(c) shows a possible processor allocation for the plan in figure 3(b), assuming a parallel machine with five processors.

In the example shown, the baseAssemblies extent is partitioned over processors 1, 2, 3, 4 and 5. Following the criteria itemised above, the subplan with a scan operator is placed with its extent, in this case processors 1, 2, 3, 4 and 5 for table_scan over baseAssemblies. The subplan with the materialise operator is placed on the union of all processors allocated for its input subplan (processors 1, 2, 3, 4 and 5 in this case). Finally, reduce can be placed on any of the five processors.

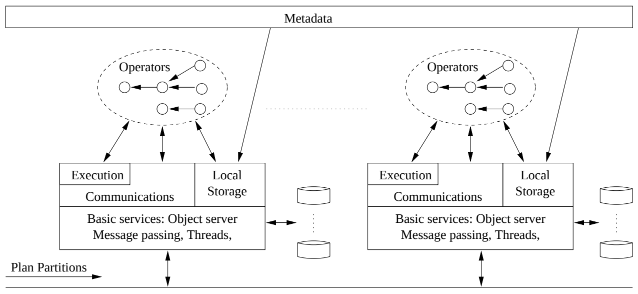

Figure 4: Functional components of the object store.

In the experimental work described in this paper all data is partitioned over all processors, so the processor allocation analysis simplifies accordingly.

5. QUERY EVALUATION

The query support services within the object store are illustrated in figure 4. A partition of the parallel plan may contain a number of independent trees of operators, which execute concurrently. A description of each such partition of the parallel plan is transported to its destination where it is instantiated using the appropriate physical operators in the execution engine and its execution initiated.

The services provided by the object manager to the execution engine include the management of execution threads, flow-controlled data transmission between operators, and access to local application data, which is organised as partitions of the extents defined in the overall schema.

A buffer is defined as the unit of transfer between processors, and tuples are packed into and unpacked from buffers within the operator exchange. Buffers have a (configurable) size limit to ensure a steady flow, but can grow to accommodate an arbitrarily large tuple if necessary, though the query optimiser takes account of the cost of any data redistribution. The current policy is that after the first tuple is packed into a buffer, that buffer's size is fixed to the greater of its current size and the configured limit, and it is this size which then determines when the buffer is full, and thereby ready for transmission. The exchange operator is parameterised in the query plan with a set of destinations and the specific destination for a specific tuple decided by an arbitrator which is again configured within the query plan, for instance to distribute tuples round robin, to copy to all destinations or to distribute according to a specified component of the tuple, such as an object's home node.

Support for communications builds on the message passing facilities of MPI [25] to control resource usage in the flow of data between operators. Space utilisation is controlled by allocating buffers from and returning them to a pool, whose size is configurable. A manager thread handles incoming traffic, queueing buffers for the local operators, and monitors completion of non-blocking output requests so as to reclaim buffers. This manager implements a flow control policy in cooperation with communication managers in other processors.

Each node stores objects locally in an instance of the Shore object manager [5]. A number of basic access mechanisms are defined to support access to an extent partition, returning for each object an unstructured block of data. To support the creation of a structured tuple from this record, an object walker [1] interface is implemented. This defines a number of *virtual* handler functions and a function to initiate traversal of a record containing the data of an object. During the traversal, the function calls appropriate handler functions to pass control to the application, such as a tuple factory which is implemented as a class derived from the object walker class. It is in the child class that the tuple is actually created in redefined handler functions.

6. PRELIMINARY RESULTS

A well known benchmark for OODBs is OO7 [6]. OO7 defines a schema which is intended to typify a CAD/ CAE database, with a hierarchy of design objects. At the largest grain are modules which comprise a number of assemblies. Assemblies are either complex assemblies which comprise other complex assemblies or base assemblies which are composed of composite parts, themselves composed of atomic parts. The construction of composite parts from atomic parts is represented by a random collection of interconnections between the atomic parts. Each composite part has a document and, in addition, a module has a manual.

OO7 defines various example database configurations, characterised by the number of atomic parts, the depth and spread of the hierarchy of assemblies, the number of interconnections between atomic parts and the size of documents and manuals. Also defined are a collection of traversals and queries; the traversals are suited to navigational access while the queries suggest use of a declarative query language. The implementation of OO7 used in this work follows that defined in the original paper, using the names of classes, etc, so the schema is not reproduced in this paper.

The results reported here are measurements of the performance of a selection of associative queries closely based on those in the OO7 benchmark for instantiations of the OO7 database on Polar within a distributed memory parallel environment. This environment is a cluster of 233MHz Pentium II machines running RedHat Linux version 6.2, each with 64MB main memory and a number of local disks, connected via a 100Mbps Fast ethernet hub. For each experiment data is partitioned in "round robin" style over some number of processors but located on one disk per processor, this being a MAXTOR MXT-540SL. The OO7 configurations used are medium and large, in all cases having fan-out 3.

6.1 Selection of Queries from OO7

The simplest query selected for these initial experiments is Query 7 of the benchmark; a scan over all atomic parts, which is expressed in OQL as:

```
        select a.id from a in atomicParts;
```

An example of a join is Query 8 of the benchmark, which prescribes a search for pairs of atomic parts and documents where the `id` of the document matches the `documentId` attribute of the atomic part. Clearly the number of values returned will be equal to the total number of atomic parts. This query may be expressed in OQL as:

```
select struct(A: a.id, D: d.id)
from d in documents, a in atomicParts
    where a.docId = d.id;
```

Of interest particularly in an OODB is the potential for direct traversal between objects via relationships, i.e. navigation. Query 5 of the benchmark, introduced already in section 4.1, suggests such a path traversal. This query corresponds to a single step of the build date comparison tests required by a `make` utility, selecting all base assemblies that use a composite part with a `buildDate` later than the `buildDate` of the base assembly. In the absence of duplicate detection, a slight modification of the query introduced earlier may be expressed as:

```
select struct(B: b.id, C: c.id)
from b in baseAssemblies, c in b.componentsPriv
    where b.buildDate < c.buildDate;
```

However, to increase the volume of data accessed, an alternative query, referred to as query 5a, is defined:

```
select distinct c.id
from c in compositeParts, a in c.parts
    where c.buildDate < a.buildDate;
```

The rationale for this modification is that a complete `make` operation, represented by query 5 would ultimately require a check of the build dates of all design objects contained in a module, implying a need not just for the recursive check as originally proposed in query 6 but also that the build dates of atomic objects should be compared against the build dates of containing composite parts.

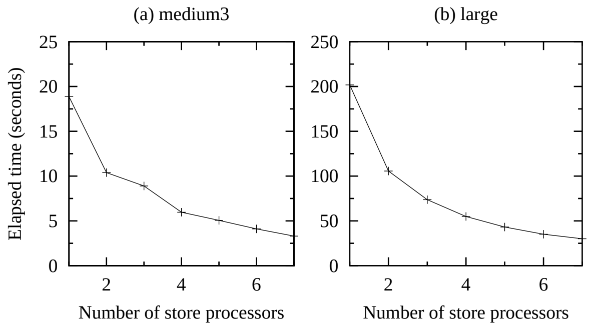

Figure 5: Performance of query 7 (Sequential Scan) over medium and large OO7.

6.2 Discussion of Results

Figure 5 shows cold performance times for query 7 on medium and large OO7 configurations, with respectively 100000 and 1 million atomic parts and fan-out 3. As expected, the absolute execution execution time does appear to increase by a factor of 10.

Figure 6 shows cold performance times for query 8 on the same medium and large OO7 configurations. While Polar implements a `hash_join` operator, the local value join operator used in these experiments is `nested_loops`. Similarly while Polar supports both *replicate and partition* and *symmetric partition* policies for redistribution [15], it is the former which is used in these initial experiments. The cost of redistribution is minimised through the use of the `reduce` operator to filter out the attributes which are not required. Thus, the texts of the documents for instance are not replicated. Clearly it is the CPU intensive processing in the nested loops algorithm itself which dominates performance here. In the large database, the number of both documents and atomic parts increases over the medium database by a factor of 10, so it is then not suprising that the elapsed times for the large database are 100 times those observed for the medium database; hence only a few shorter duration experiments were performed!

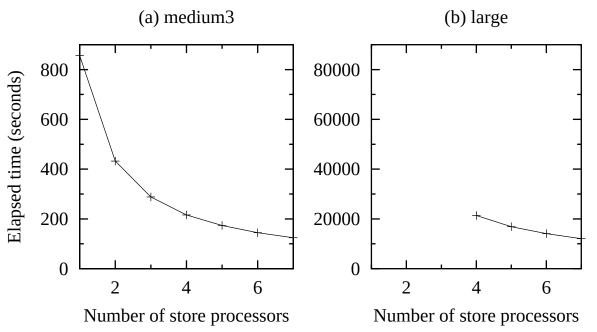

Figure 6: Performance of query 8 (Value join) over medium and large OO7.

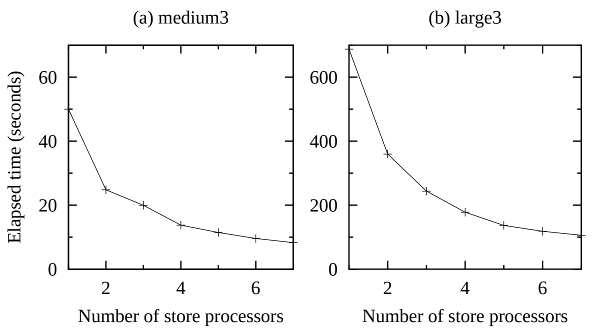

Figure 7: Performance of query 5a, based on query 5 (Unnest) over medium and large OO7.

Figure 7 shows cold performance times for query 5a on the medium and large OO7 configurations. The volume of data accessed by the query increases by a factor of 10 between medium and large databases, suggesting a corresponding increase in execution time. One possible explanation for the greater increase is the reduction in the rate of hits to the object store disk buffer cache in the materialise operator, in the presence of random accesses to larger extent partitions.

For comparison, figure 8 shows cold performance times for two alternative value based join realisations of query 5 on the medium configuration. The original query 5 is used here because the join used is again nested_loops so the execution times tend to be high. The sizes of the two collections joined in query 5 are 500 and 3×729 giving a total of about 1 million comparisons. In the case of query 5a, the corresponding number is 100000^2, i.e. 10000 times greater.

In both cases, the materialise is replaced by a nested_loops join and second table_scan. The first experiment represented in the left hand graph uses *replicate and partition* policy for redistribution while the second directs elements in the collection arising from unnesting base assemblies by OID each to their home processor, as suggested in [19]. The speedup in the latter case is proportional to $1/n^2$ (n being the number of store processors), since (as one of the collections is not moved) the number of objects involved in the local join on a particular processor falls with the number of processors.

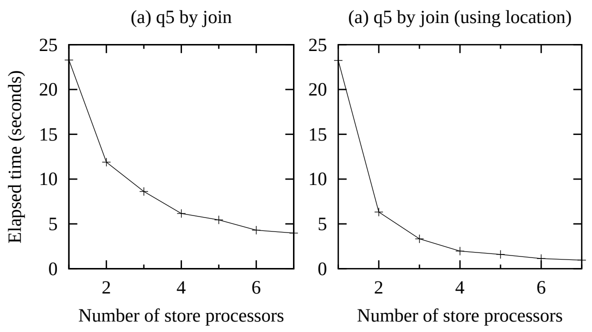

Figure 8: Performance of value join based realisations of the original query 5 oo medium OO7.

Clearly the direct relationship to the size of the inner collection disappears if the nested loops algorithm is replaced by say a hash based alternative, but the saving in redistribution costs remains.

7. CONCLUSIONS

Focussing on the parallel execution of queries, we have described the architecture of the Polar parallel object database server which claims novelty through supporting ODMG on a shared nothing architecture with disk based data. The performance results obtained show that parallel speed-ups can be achieved for queries defined in the OO7 benchmark, even when the parallel platform consists of a low-cost cluster of PCs, rather than a custom parallel system with tightly coupled nodes. Polar's modular construction, which links together a number of established components, is intended to improve portability as well as allowing experimentation with alternative components. A port to a massively parallel processor (MPP), an ICL Goldrush, is nearing completion.

It is clear from our work so far that the differences between relational and object databases lead to interesting differences in the design of parallel servers to support the two paradigms. Initial experiments have uncovered the need for further experimentation on the efficient evaluation of queries that explore relationships between objects, and ongoing work is investigating a wider range of parallel pointer based join strategies over a more diverse collection of queries. In addition, future work will focus on other other key differences, including parallelising queries that contain method calls, and support for mixed workloads that include both parallel queries and navigational clients that map persistent objects into client programs. The overall goal is to be able to support the increasing demands for performance and capacity being generated by the growing number of large-scale applications that require access to persistent objects.

8. ACKNOWLEDGEMENTS

The work reported here has been supported in part by research grants from the Engineering and Physical Sciences Research Council; numbers GR/L89655 and GR/L24618. The development of the query execution engine was greatly assisted through reuse of an earlier implementation by Pedro R. Falcone Sampaio.

9. REFERENCES

[1] *POET Generic Programming Guide*. San Mateo, California, 1997. Version 5.0.

[2] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proceedings of the ACM SIGMOD*, pages 287–296, Washington, DC, USA, 1993.

[3] P. A. Boncz, A. N. Wilschut, and M. L. Kersten. Flattening an Object Algebra to Provide Performance. In *International Conference on Data Engineering*, pages 568–577. IEEE Press, 1998.

[4] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):5–24, March 1990.

[5] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 383–394. ACM Press, 1994.

[6] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, US, May 1993. ACM Press.

[7] R. G. G. Cattell, D. K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gameran, D. Jordan, A. Springer, H. Strickland, and D. Wade, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, 1997. ISBN: 1-55860-463-4.

[8] Y.-H. Chen and S. Su. Implementation and evaluation of parallel query processing algorithms and data partitioning heuristics in object-oriented databases. *Distributed and Parallel Databases*, 4:107–142, 1996.

[9] S. de F. Mendes Sampaio, N. W. Paton, P. Watson, and J. Smith. A parallel algebra for object databases. In *Workshop on Parallel and Distributed Databases, in conjunction with DEXA'99*, Florence, Italy, Aug. 1999.

[10] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[11] L. Fegaras. An experimental optimizer for OQL. Technical Report TR-CSE-97-007, CSE, University of Texas at Arlington, 1997.

[12] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.

[13] M. Gesmann. Mapping a Parallel Complex-Object DBMS to Operating System Processes. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *International Euro-Par Conference*, pages 852–861, Lyon, France, Aug. 1996. Springer-Verlag Inc.

[14] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 102–111, 1990.

[15] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[16] T. Grust, J. Kröger, D. Gluche, A. Heuer, and M. H. Scholl. Query evaluation in CROQUE - calculus and algebra coincidence. In *15th British National Conference on Databases*, volume 1271, pages 84–100. Springer, 1997.

[17] W. Hasan. *Optimization of SQL Queries for Parallel Machines.* PhD thesis, Stanford University, Department of Computer Science, Dec. 1995.

[18] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *Proceedings of the 21st VLDB Conference*, 1995.

[19] D. Lieuwen, D. DeWitt, and M. Mehta. Parallel pointer-based join techniques for object-oriented data bases. In *International Conference on Parallel and Distributed Information Systems*, pages 172–181. IEEE Computer Society Press, Jan. 1993.

[20] M. E. S. Loomis and A. B. Chaudhri, editors. Prentice Hall, Dec. 1997. ISBN: 013899725X.

[21] M. Olson, W. Hong, M. Ubell, and M. Stonebraker. Query processing in a parallel object-relational database system. *Data Engineering Bulletin*, 19(4):3–10, 1996.

[22] N. W. Paton, S. A. Khan, A. Hayes, K. Eilbeck, C. Goble, S. J. Hubbard, and S. G. Oliver. Conceptual modelling of genomic information. In preparation, 2000.

[23] K. Ramaiyer, R. Brunner, I. Csabai, A. Connolly, M. Goodrich, G. Szokoly, and A. Szalay. Designing a terabyte database for astronomical data mining. Technical report, 1995.

[24] J. Shiers. Building a multi-petabyte database: The RD45 project at CERN. In M. E. S. Loomis and A. B. Chaudhri, editors, *Object Databases in Practice*, pages 164–176. Prentice Hall, 1997.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference.* The MIT Press, Cambrideg, Massachusetts, 1998. ISBN: 0-262-69215-5.

[26] M. Stonebraker and P. Brown. *Object Relational DBMSs Tracking The Next Great Wave.* Morgan kauffman, Sept. 1998. ISBN: 1-55860-452-9.

[27] A. Thakore and S. Su. Performance Analysis of Parallel Object-Oriented Query Processing Algorighms. *Distributed and Parallel Databases*, 2:59–100, 1994.

[28] P. Watson. The design of an ODMG compatible parallel object database server. In *International Meeting on Vector and Parallel Processing (VECPAR), LNCS 1573.* Springer, June 1998.

[29] P. Watson and G. Catlow. The architecture of the ICL Goldrush MegaServer. In C. Goble and J. Keane, editors, *13th British National Conference on Databases*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[30] P. Watson and P. Townsend. The EDS parallel relational database system. In P. America, editor, *The PRISMA Workshop on Parallel Database Systems*, Lecture Notes in Computer Science, Noordwijk aan Zee, 1990. Springer-Verlag.