A New General Purpose Parallel Database System

M. Afshar, J. Bates, G. Bierman, K. Moody
Computer Laboratory
University of Cambridge
Cambridge, CB2 3QG

Abstract

This paper is concerned with the transparent parallelisation of declarative database queries, based on theoretical principles. We have designed an entire database architecture suitable for use on any general-purpose parallel machine. This architecture addresses the shortcomings in flexibility and scalability of commercial parallel databases. A substantial benefit is that the mathematical principles underlying our framework allow provably correct parallel evaluations and optimisations, using compile-time transformations. We address parallelism in a language-independent way through the choice of monoids as a formulation for expressing and modelling queries. Queries expressed in our declarative language are transformed into applications of a higher–order function, the monoid homomorphism. The evaluation of this function is partitioned at run-time, giving a tree-like processor topology, the depth and breadth of which can be varied with a declarative execution plan. Leaf nodes within the evaluation tree operate on their own data partitions and forward results to the appropriate interior nodes. Due to the nature of our language, the functions that are necessary to combine results from independent parallel evaluations are generated automatically at compile–time from a monoid definition dictionary, additions to which can be made to extend the system's data types. We have built a complete prototype of our system, which uses Swiss Radio Corporation's entire recorded music catalogue, on a general–purpose AP1000, 128–cell parallel computer at the IFPC.[1]

1 Introduction

Our research is motivated, in particular, by opportunities to improve the efficiency of database queries. We propose an abstract declarative method for query expression, removing the need to break down the problem into parallel tasks, handle internal communications or assign a procedural processor–task topology.

[1] Imperial College/Fujitsu Parallel Computing Centre, London

This paper is structured as follows: firstly we detail the motivations for our work. Section 1.2 gives an overview of related work. Section 2 describes our approach and the high-level design of our system and is followed by a theoretical grounding of our work in section 3. We then discuss our prototype implementation in section 4 before going on to present preliminary results in section 5. We make some concluding remarks in section 6.

1.1 Motivation

Parallel databases are suitable for tackling very large data sets or applications with high transaction volumes. Those applications typically involve databases containing many gigabytes of information.

Current commercial approaches lack flexibility, which is mainly due to the fact that they are targeted specifically at SQL and they lack extensible data types. Scalability is also a problem, mainly due to the use of a *shared disk* approach. Moreover, they lack a theoretical basis, i.e. a provably correct model for their mode of parallel query evaluation. We address all of these shortcomings in our work.

A parallel database can be defined as a system in which more than one process is used to perform database queries. However, we shall assume that a parallel database involves the following sub–processes: splitting database queries into parallelisable sub–queries; performing these sub–queries in parallel; collecting and amalgamating the results.

In a tightly coupled shared-disk system, processor nodes in a parallel database system will be assigned to carry out the above functions. At the bottom level are parallel data managers built on processors with attached disks. Each data manager can evaluate sub–queries on its local data set. The total data is partitioned in a suitable manner across these data manager nodes (see section 4 for details). Other nodes are responsible for splitting up queries, passing the sub–queries to the data management nodes and then collecting and amalgamating the results.

In commercial parallel database systems, such as Or-

acle, standard SQL queries are supported, the parallelisation of which is made transparent. Businesses can thus upgrade to powerful parallel database engines without changing their existing software infrastructure. These systems are tuned for maximum transaction throughput.

In the research domain, related projects are addressing both *parallel object–oriented systems* and *specialised parallel database machines*, the latter becoming less fashionable. In the former, research is at present focussed on the parallelisation of individual operations in OQL in order to address the inefficiency of current object query evaluation strategies. These efficiency problems are brought about by the semantic richness and powerful data modelling features of OQL.

In contrast, we base our entire database systems architecture on a theoretical foundation which we claim covers all current approaches.

1.2 Related Work

Although the specific target area of this paper is databases, the underlying model is of general relevance to parallel data processing. It unifies several related research strands involving parallel programming, theoretical database and algorithms.

Research into parallel programming languages/systems can be broadly categorised into three areas: those which address the parallel processing problem by adding features to traditional sequential programming languages; those that provide standard library functions which can be used to build parallel applications; and those that take a declarative path concentrating on defining stereotypical patterns of parallel computation which can be put together. We briefly look at each of these in turn.

In the first category lie systems which have extended a procedural language with *explicit* constructs for parallelism such as *C** [8] and High Performance Fortran [10], which is the latest language that combines Fortran 90 with enhanced user defined data distribution. A similar idea is the provision of standard message passing library extensions, such as MPI and PVM, which enable much flexibility through facilitating the use of existing sequential languages (currently C/C++ and Fortran). Both supply a set of routines to allow tasks on each processor to communicate.

Although suitable for some application classes, difficulties still exist in both of these approaches such as task division and allocation, expressing massively parallel programs, and reasoning about programs.

Within the database community, an alternative approach has been to exploit implicit parallelism within database queries by use of high level declarative languages such as logic or functional languages [5, 9, 13]. One advantage of the functional framework is that program transformation is based on mathematical principles. Higher order functions allow recursion patterns over structured types to be "abstracted out", leading to concise programs with no explicit recursion and hence greater potential for program transformation. Although such languages offer the potential for much implicit parallelism, most of this is too fine–grain for efficient parallel implementation.

A derivative of this third approach has been to concentrate on the development of a programming methodology for parallel machines which allows portability through the use of, for example, high level specifications, or *skeletons*. Skeletons have received attention both from the theoretical view and in practical implementations [6, 4, 2]. A skeleton is a high level template description of some parallel operations in which part of the functionality is parametrised. Skeletons can be expressed naturally through their equivalent higher order functions in functional languages. Much complexity (process topology, data distribution and amalgamation and control) can be hidden inside skeletons, easing considerably the task of the compiler. Furthermore, by using performance models for each skeleton, a compiler can make accurate decisions about resource allocation. The use of skeletons allows the programmer to structure the parallel computation by means of the composition of a set of well defined patterns. Common examples of skeletons are processor farms, process pipelines and divide and conquer trees, which exhibit independent parallelism.

Notable implementations of the declarative approach include NESL[3], which allows the definition of arbitrary functions in a pseudo–functional language, and P3L [2], which allows skeletons to be used as a glue for sequential C++ code and includes a performance model for the evaluation. A key feature of NESL is that functions can be applied in parallel to achieve nested parallelism.

Independent parallelism, as exhibited by functions such as `map` and `filter`, is very effective for decomposed forms of data and for divide–and–conquer computations, which categorises many database operations. Structural recursion, which is a development of these ideas has been investigated, both over single collection types [12, 11], and over multiple collection types [7].

Our work draws upon the work in database query modelling [7] and the `reduce` skeleton in [1]. `reduce` is a derivative of `divide–and–conquer` and `map` skeletons and resembles the higher–order function *fold* in

functional languages. The evaluation of this function proceeds in a tree–based fashion.

2 General Approach

2.1 Approach

Queries are expressed in a declarative language environment in which we use program transformations on a stereotypical function (capable of divide–and–conquer computation), which is then subject to evaluation in parallel.

The algebraic properties necessary for parallelism form the basis for our choice of monoids to model collection (e.g. lists, bags) and primitive types (e.g. max, sum, product).

Expressions expressed as a (potentially nested) monoid comprehension are translated to the application of a single function, the monoid homomorphism, which resembles the `reduce` skeleton. The evaluation of expressions involving this function are then subject to parallelism at run–time. Monoid comprehensions offer an intuitive and declarative basis for defining, translating and evaluating DBMS query languages. We view monoid comprehensions as a high-level form for monoid homomorphisms, which we see as abstract machine code.

Monoid comprehensions can capture most collection and aggregate operators currently in use for relational and object algebra [12], and can be used to express queries which simultaneously deal with more than one collection type and also naturally compose in a way that mirrors the allowable query nesting in OQL. They also permit easy integration of functional subexpressions, as used in [7]. The following example shows a query in SQL, monoid comprehensions and monoid homomorphisms.

```
SQL           Monoid Comprehensions          Monoid Homomorphisms
SELECT NAME                                  hom[SET,SET].λcd.
FROM CDs      bag{name(cd)|cd←CDs,           (if (age(cd) >3)
WHERE AGE>3     age(cd)>3}                    then {cd}
                                              else {}) CDs
```

The use of monoid comprehensions is based on the following factors:

Firstly, queries expressed in the monoid comprehension are translated into applications of monoid homomorphisms, which are amenable to parallel evaluation since they capture a `reduce` computation.

Secondly, the information that influences the amount of parallelism that can be extracted from the evaluation of a query is dependent upon a number of algebraic properties (i.e. associativity, commutativity, idempotency and existence of a "unit" for the merge function). Whilst it is desirable to elucidate whether particular functions exhibit such algebraic properties by inspection, the problem is, in general, undecidable. Hence, we have taken the approach of allowing the database maintainers to provide this information for the system. This information is made available to the system through a monoid dictionary, at the time when each monoid is defined. This means that the combiner functions that are required to combine the results from parallel evaluations of a particular reduction, are in fact, part of the natural definition of each monoid involved in the reduction. For example, in the definition for the monoid $set(\alpha)$, the user–defined function to combine instances, which is part of the monoid definition, is the well known set–union operator $\cup$ (the implementation of which is in the monoid dictionary). The information held in the monoid dictionary contributes to making the data types that the system can deal with extensible and also enables us to carry out the kind of type–checking which is necessary to verify the correctness of our mode of parallel evaluation.

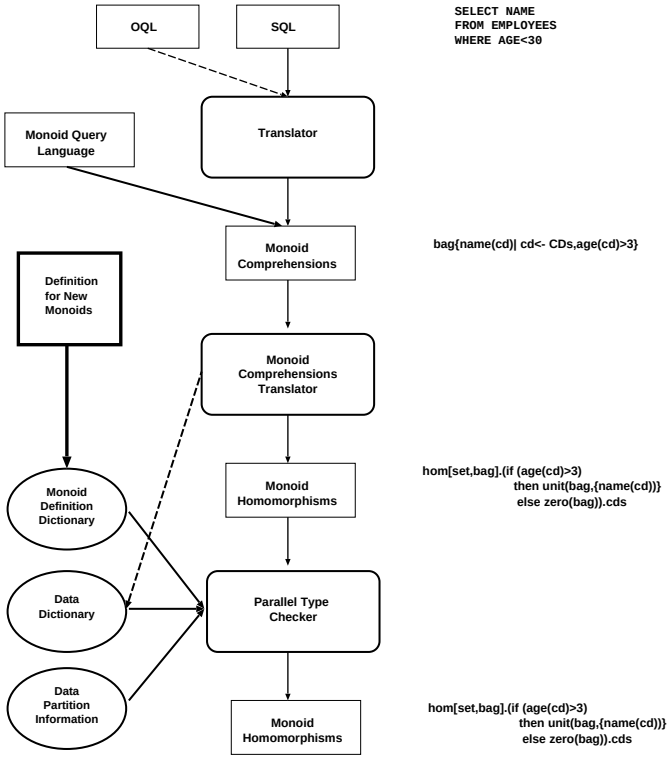

Figure 1: Software Architecture

2.2 Design

Figure 1 represents our software architecture.

Queries may either be expressed in terms of monoid comprehensions or in SQL (for which a translator produces monoid comprehensions). For example, a query to select "the name of all compact discs produced more

than three years ago" is shown in section 2.1.
Expressions containing monoid comprehensions are fed into a translator that produces monoid homomorphism code on which optimisations are carried out. At this stage, the types of each collection monoid which represent a database object (e.g. `cds` in a musical compact disc database), are obtained from the data dictionary. The result of this process is then fed into a type–checker which uses the monoid dictionary to ensure that correct parallel evaluation can be achieved using our methodology (section 3.3). This parallel type-checker requires a *data dictionary*, which includes descriptions of the data in the database, a *monoid definition dictionary* which includes definitions for all monoids (table 1), and data partitioning information which stipulates how the data has been partitioned between processors. This information has a direct influence on the type-checking process (in section 3.3). At the end of this process, code expressed in terms of monoid homomorphisms is suitable for evaluation on the parallel evaluator.

3 Theoretical Basis

Our high–level language is based on monoid comprehensions. In this section we discuss the theoretical basis for this and also for the lower–level language, monoid homomorphisms, which is executed by the back–end parallel query evaluator.

3.1 Monoid Comprehensions

Our query language is based on monoid comprehensions, which are translated into parallel operations expressed in terms of the *monoid homomorphism* which operates on multiple monoid types.

Definition 1. (Monoid). A data type T is expressed as a monoid $\mathcal{M}$ with a unit function $\mathcal{M} = (T, zero, unit, merge)$, where the function *merge*, of type, $\mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$, is *associative* with left and right identity *zero*.

If $\forall x, y : merge[T](x, y) = merge[T](y, x)$ then we say that the monoid is **commutative**.

If. $\forall x : merge[T](x, x) = x$, we say that the monoid is **idempotent**.

Definition 2. (Free Monoid). *Let T(α) be determined by the type parameter α (i.e. T is a type constructor) and (T(α,zero[T],unit[T],merge[T])) be a monoid. The quadruple (T(α),zero[T],unit[T], merge[T]) where unit[T] is a function of type* $\alpha \rightarrow T(\alpha)$*, is a free monoid.*

We divide monoids into two sorts: *collection* and *primitive* monoids, as in table 1. Collection monoids which capture bulk types, such as lists, sets and bags, and can be freely nested. Primitive monoids capture primitive types such as integers and booleans.

Collection Monoids					
$\mathcal{M}$	$\mathcal{T}$	**zero**	**unit(x)**	**merge**	**Com/Idem**
set	set(α)	{}	{x}	$\cup$	C&I
bag	bag(α)	{{}}	{{x}}	$\uplus$	C
list	list(α)	[]	[x]	++	-

Primitive Monoids					
$\mathcal{M}$	$\mathcal{T}$	**zero**	**unit(x)**	**merge**	**Com/Idem**
sum	int	0	x	+	C
max	int	0	x	max	C&I
all	bool	true	x	$\wedge$	C&I

Table 1: Examples of Collection and Primitive Monoids

A comprehension over the monoid $(T, zero[T], unit[T], merge[T])$ is defined by the following inductive equation:

$T\{e \mid \} = unit[T](e)$
$T\{e \mid x \leftarrow u, r\} = hom[S, T](\lambda\ x\{e \mid r\})\ u$
$T\{e \mid pred, r\} =$ **if** $pred$ **then** $T\{e \mid r\}$ **else** $zero[T]$

where r is a possible (empty) sequence of terms (of the form $x \leftarrow u$ or $pred$) separated by commas, $pred$ is a predicate, and u is an expression of type S(α), where S is a free monoid with $S \preceq T$. The expression at the head of the comprehension (to the left hand side of the | sign) may contain other comprehension terms.

3.2 Monoid Homomorphism

A monoid homomorphism $hom[T, S]$ from the free monoid $(T(\alpha), zero[T], unit[T], merge[T])$ to a monoid $(S, zero[S], merge[S])$ is defined by the following inductive equations:

$hom[T, S](f) zero[T] = zero[S]$
$hom[T, S](f) unit[T](a) = f(a)$
$hom[T, S](f) merge[T](x, y) =$
$\quad merge[S](hom[T, S](f)x, hom[T, S](f)y)$

where T$\preceq$S, where $\preceq$ is the obvious ordering on monoid properties, e.g. $\{associate\} \preceq \{associative, commutative\}$. For example, a monoid homomorphism `hom[set,sum]` that captures the cardinality of a set is not valid since set $\not\preceq$ sum (set is commutative and idempotent, whereas sum is just commutative). Conformance to this rule is checked by the type–checker for parallel evaluation (section 3.3), since the commutativity and idempotency properties of a monoid are specified explicitly when the monoid is defined and are stored in the *monoid definition dictionary*.

3.3 Type-Checking

In addition to checking that queries are well formed with respect to the database schema as defined in the data dictionary, we incorporate a type–checker for parallel evaluation which is built into our system. The function of this type–checker is to ensure that queries expressed in terms of monoid–homomorphisms make legitimate transitions and hence that our methodology for parallel evaluation yields a correct result. In order to make such decisions, the type–checker needs two types of information: the monoid definition table, and information on data partitioning. The first type–checking stage is concerned with ensuring that the homomorphisms in the query follow acceptable transitions (i.e. that $T \preceq S$ is satisfied). For queries that fail first type–check, we use data partitioning information to ascertain whether the query is can still be evaluated correctly.

For example, the query given below selects all tracks from a database of tracks on CDs:

```
sum{1 | t ← tracks, date_recorded(t)<=1992}
```

The homomorphism in the above query takes a set (of tracks) and produces a sum from it (i.e. gives the total number tracks in the database which were recorded before 1992). Although this query may appear to be legitimate, in the absence of information regarding data partitioning, it is rejected. A closer inspection of the properties of the collection monoid, *set* and the primitive monoid, *sum*, reveals that *set* is a commutative, idempotent monoid whereas *sum* is only commutative (i.e. the condition $T \preceq S$ is not satisfied). Hence, the query is passed to the second phase of the type–checking process.

The given query can lead to a correct parallel evaluation if the argument data structures, which are the *set* of `tracks` and `CDs`, are distributed across the nodes such that there are no duplicates across nodes (i.e. the collection can be reconstructed without performing any duplicate purging). Since data of collection type *set* is partitioned across nodes in this way, i.e. there are no duplicates across nodes, then the evaluation can be accepted by the second phase type–checker, since there are no intermediate partial results produced which violate this condition.

However, the following query which returns the number of tracks that have been produced in the same country as the CD requires further analysis:

```
sum{1|t←tracks, cd←cds,
      recorded_in(track)=recorded_in(cd)}
```

The information in the data dictionary regarding each collection that has been split across multiple nodes holds information about the attribute upon which such partitioning has been carried out (if appropriate). In our database of `CDs` and `tracks`, information is stored for each table noting that it has been partitioned on the join attribute, namely `cd_id`. Clearly, since none of the filters in the above query contain an equality on the join attribute, the query is rejected. This can be traced to the observation that the query asks for the tracks that have been recorded in the same town as *any* CD, not the particular CD title on which the track appears. Since there may indeed be cases where potential results are lost, e.g. a track appearing on node 1 is recorded in London and so is a CD on node 6 but this does not contribute to the final result since it does lie across a node boundary, the query is rejected. Our check detects when our mode of evaluation would be incorrect, however contrived the query example is! Indeed, an efficient parallel implementation could only be produced if the argument set can be merged without performing duplicate purging.

The type-checker also has at its disposal the meta data detailing the way in which each database table has been partitioned. If, for example, the tracks database were to be split up on the `recorded_in` field, then the type–checker would infer that the monoid type for the `tracks` table is in fact *partitioned–set* if the query were to involve an `recorded_in` filter. The query would thus have been verified to be correct.

4 Implementation and Experiments

We have implemented a prototype of our system on a single-user 128–node Fujitsu AP1000 parallel computer. It would be possible to port this to any parallel machine on which a subset of nodes have access to disks. Each node (cell) consists of one processor running a UNIX–like operating system with no pre–emptive scheduling. Local disks are attached to 32 cells, although all cells are able to access all 32 of the local disks. The machine is SIMD in principle, although it is possible to obtain MIMD behaviour, a feature which we make use of in our evaluations. We use the native message–passing libraries, as, at the time of implementation, no local I/O could be carried out on the MPI implementation.

Our experimental data set consists of the *Swiss Radio Corporation's* CD acquisitions catalogue. This database consists of two tables: a compact discs table, and a tracks table with about 30,000 CDs and on average about 9 tracks per CD. Both tables have been partitioned horizontally on the key attribute `cd_id`, so that all tracks relevant to a CD can be found on

the same node. This is required to address potential problems with nested queries involving idempotent monoids (e.g. set, which is commutative idempotent), as discussed in section 3.3.

As shown in figure 1 database queries are expressed in terms of monoid comprehensions, or SQL — in which case they are subject to translation to monoid comprehension first — which are translated into monoid homomorphisms at which stage suitable type checking is carried out. During this process optimisations may also be carried out, to produce optimised monoid homomorphism queries.

The prototype system requires a declarative execution plan in order to define the form of the evaluation–tree since, due to the operating system limitations, we are not able to create processes dynamically on the cells of the AP1000. This means that the process topology is defined at compile–time in the form of a declarative evaluation map, which defines the number of processors at each level within the evaluation tree. The depth and breadth of the evaluation tree influences the efficiency of the parallel evaluation. Changes to a declarative evaluation map may require re–organisation of the data residing on local disks. If the number of leaf nodes which process data directly off their local disks is varied, (e.g from 32 to 16), some reorganisation of the database partitions is required. The run–time system takes care of this reorganisation.

The prototype parallel evaluator requires both the monoid homomorphism code and a declarative evaluation map. Its software architecture is given in figure 2. The server running on the host broadcasts the monoid homomorphism program together with a declarative evaluation map to the 128 cells on the machine. Each cell then calculates its position in the evaluation tree using the plan, which always consist of one cell acting as the root gathering results at the top. Cells which are at the leaf nodes of the evaluation tree are necessarily equipped with disks on where their data resides. Each leaf node interprets the homomorphism code and sends its results to its sole destination cell periodically during evaluation. Internal cells in the evaluation tree combine results from a number of leaf–nodes (or other internal nodes) and forward their results to internal nodes higher up in the evaluation tree. The functions which are required to combine results from other cells at the internal nodes are available in the monoid definition dictionary, which is part of the run–time system of each cell. The evaluation is complete when the root cell has finished combining results from its communication partners.

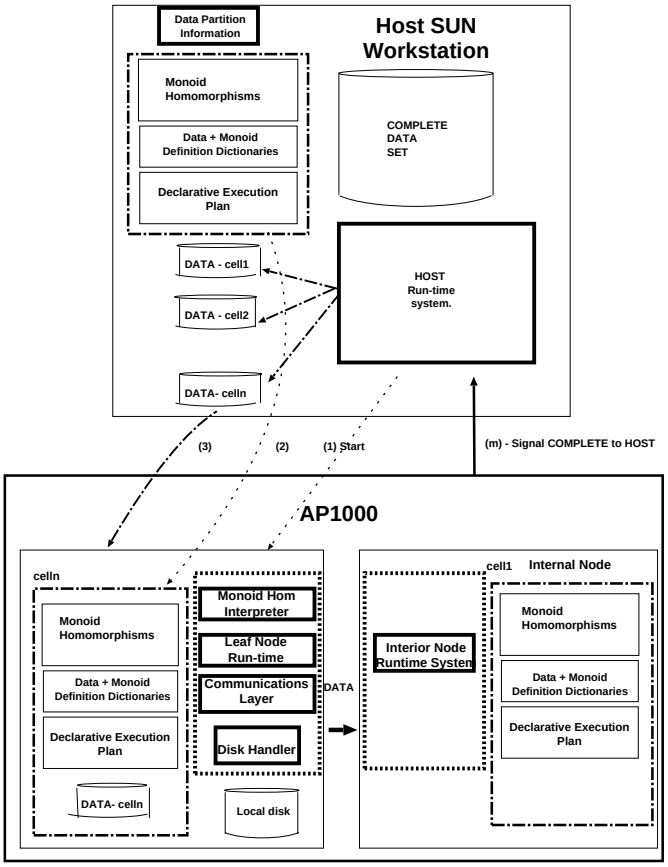

Figure 2: Run–time Software Environment

5 Initial Results

The graph below shows the speedup for typical query runs on a single table (CDs table), with configurations containing up to 20 cells. Initial inspections show that the query with the lowest selectivity (3% of the data inspected is returned in the result) has the fasted evaluation time, whereas the other two evaluations which are less selective (with selectivities of 14.8% and 85%) take longer. As would be expected, the speedup is greater with the query returning fewest results.

The results show that the runs with 19 and 23 cells do not show a relative performance improvement over the runs with 17 cells. The information which we have omitted from the graph gives us an insight into why this may be so.

Runs with 19 and 23 cells have configurations of (1,2,16), and (1,2,4,16), where these numbers indicate the number of cells at each level from the root down to the leaves within the evaluation tree. The run with 17 cells, which has a configuration of (1,16), does not

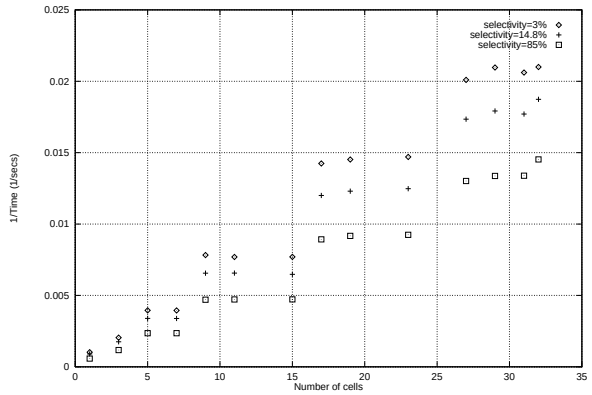

Figure 3: Speedup graph

perform significantly worse than runs with 19 and 23 cells since the extra-level(s) within the evaluation tree in the latter runs is/are degrading rather than enhancing performance. Put another way, the evaluation tree is too deep for the amount of data and cell configuration in the runs with 19 and 23 cells.

6 Conclusions

This paper has described the theoretical basis, design and implementation of our model for parallel data processing. We have presented a framework for parallelising database queries based on monoid comprehensions as a declarative high–level language. Expressions involving monoid comprehensions are translated into applications of the monoid homomorphism, which is subject to parallel evaluation in a tree–structured topology. Monoid definitions are kept in a monoid definition dictionary, additions to which can be made in a modular, organised, simple and provably correct way, without any need to change any existing code. The definition of monoids includes functions required to combine partial results. This enables experimentation with different implementations of the same combiner operations. The parallel evaluation of monoid homomorphisms is more scalable than that of a master–slave arrangement, which is sometimes used for parallel database query evaluation. We have implemented our framework on a Fujitsu AP1000 128–cell parallel machine using a large commercial dataset.

References

[1] M Afshar and H Khoshnevisan. Mechanical parallelisation of database applications. In *Proceedings of The 1994 ACM Symposium on Applied Computing*, pages 436–441. March 1994, Phoenix, Az.

[2] Bacci, Danelutto, Orlando, Pelagatti, and Vanneschi. P3L: a structured high level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[3] G E Belloch. Programming parallel algorithms. *Communications of the ACM*, March 1996.

[4] M Cole. *Algorithmic Skeletons: Structure Management of Parallel Computation*. Pitman/MIT, 1989.

[5] S Danforth and P Valduriez. A fad for data intensive applications. *IEEE Transactions on Knowledge and Data Engineering*, 4(1):34–51, February 1992.

[6] J Darlington et al. Parallel programming using skeleton functions. In *Proceedings, Parallel Architectures and Languages, Europe*, Lecture Notes in Computer Science, number 694, 1993.

[7] L Fegaras. A uniform calculus for collection types. Technical Report 94-030, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, OR 97291-1000, 1994.

[8] P Hatcher, W F Tichy, and M Philippsen. A critique of the programming language C*. *Communications of the ACM,*, 35(6):21–24, June 1992.

[9] M L Heytens. *The Design and Implementation of a Parallel Persistent Object System*. PhD thesis, MIT Laboratory of Computer Science, 1992.

[10] High performance fortran language specification, version 1.0. Technical Report, May 1993. Rice University

[11] T Sheard and D Stemple. Automatic verification of database transactions. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[12] V B Tannen, P Buneman, and L Wong. Naturally embedded query languages. In *Proceedings of International Conference on Database Theory, Berlin, Germany*, number 646 in lncs, October 1992.

[13] P Trinder. *A Functional Database*. Prg-82, Oxford University Programming Research Group, 1989. DPhil.