An Open Parallel Architecture for Data–intensive Applications

Mohamad Afshar

Kings College

A dissertation submitted for the degree of Doctor of Philosophy
in the University of Cambridge

December 1998

to my family

Abstract

Data–intensive applications consist of both declarative data–processing parts and imperative computational parts. For applications such as climate modelling, scale hits both the computational aspects which are typically handled in a procedural programming language, and the data–processing aspects which are handled in a database query language. Although parallelism has been successfully exploited in the data–processing parts by parallel evaluation of database queries associated with the application, current database query languages are poor at expressing the computational aspects, which are also subject to scale.

This thesis proposes an open architecture that delivers parallelism shared between the database, system and application, thus enabling the integration of the conventionally separated query and non–query components of a data–intensive application. The architecture is data–model independent and can be used in a variety of different application areas including decision–support applications, which are query based, and complex applications, which comprise procedural language statements with embedded queries. The architecture encompasses a unified model of parallelism and the realisation of this model in the form of a language within which it is possible to describe both the query and non–query components of data–intensive applications. The language enables the construction of parallel applications by the hierarchical composition of platform–independent parallel forms, each of which implements a form of task or data parallelism. These forms may be used to determine both query and non–query actions.

Queries are expressed in a declarative language based on *monoid comprehensions*. The approach of using monoids to model data types and monoid homomorphisms to iterate over collection types enables mathematically provable compile–time optimisations whilst also facilitating multiple collection types and data type extensibility. Monoid comprehension programs are automatically transformed into parallel programs composed of applications of the parallel forms, one of which is the *monoid homomorphism*. This process involves identifying the parts of a query where task and data parallelism are available and mapping that parallelism onto the most suitable form. Data parallelism in queries is mapped onto a form that implements combining tree parallelism for query evaluation and dividing tree parallelism to realise data partitioning. Task parallelism is mapped onto two separate forms that implement pipeline and independent parallelism. This translation process is applied to all comprehension queries including those in complex applications. The result is a skeleton program in which both the query and non–query parts are expressed within a single language. Expressions in this language are amenable to the application of optimising skeleton rewrite rules.

A complete prototype of the decision–support architecture has been constructed on a 128–cell MIMD parallel computer. A demonstration of the utility of the query framework is performed by modelling some of OQL and a substantial subset of SQL. The system is evaluated for query speedup with a number of hardware configurations using a large music catalogue database. The results obtained show that the implementation delivers the performance gains expected while offering a convenient definition of the parallel environment.

Preface

I would like to thank my supervisor, Ken Moody, for his thoughtful advice and practical assistance during the course of this research. I greatly benefited from the persistence and patience of Gavin Bierman — his continuous encouragement made the greatest obstacle seem surmountable. I thank all those who read earlier drafts of this dissertation: Giles Nelson who not only read and commented on numerous copies of this thesis and helped to reduce the number of long sentences in it, but also offered much (remote) practical assistance; Kaveh Moazami Goudarzi who is a master of chapter (re)organisation and the tribulations thereof; and Scarlet Schwiderski who continually willed me on. I owe much to John Bates who in addition to offering practical help, has been a constant source of reassurance — he played a special role. Lewis Tiffany provided expert Librarianship with a smile. Graham Titmus, Martyn Johnson, and most often, Chris Hadley, helped with various hardware and software problems. I am grateful to John Darlington who allowed me to use the parallel computing facilities at Imperial College. I also appreciate the help provided by Martin Khöler and Adrian Chung and the other members of the IFPC. I thank Helmut Kaufman, formerly at ETH, Zurich, who provided the SR-CD database.

I owe much to the friends with whom I shared many enjoyable experiences. Reza Gandjei who introduced me to intellectual rigour, jogging and tequila shall be sadly missed. Masoud's encouragement lead me to pursue a long research career. During this time Amir introduced me to the wonders of the "magic box". Thankfully, Cuddly Clea cooked some mouth–watering meals. Siamak, Keith, Scott, Mehrdad, Mehdi ($\times 2$), Narmin, Nasrin and Bahman provided much relief from Cambridge. My friends in Iran filled my holidays with fond memories. Pleasant coffee breaks were shared with Fox, Roberto, Christos, Dimitra and Andrea. Many a mug of herbal tea, meal, trip up and down the M11, and ten hour–long stints at the lab were shared with Kaveh. Sadly, he never benefited from the relief of sleeping on cardboard. I am grateful to Rozi and Nicola for accommodating me.

I owe much to all of my family: my mother and father, sister, uncles and aunts. I owe my pursual of academic pursuits to my uncle Mohammad–Taghi, who from my early childhood, infused me with the desire to understand. My family's continuous encouragement and their unceasing emphasis on education has benefited me greatly. I am indebted to them.

I am grateful to the Engineering and Physical Sciences Research Council for providing three years funding.

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of any work done in collaboration. I declare that this dissertation is not the same as any other dissertation I have submitted for a degree, diploma or other qualification at any university. Furthermore, no part of this dissertation has been or is currently being submitted for any such qualification. This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Contents

List of Figures **xiv**

List of Tables **xv**

1 Introduction **1**

1.1 Background . . . 3
1.2 Hardware Architecture . . . 3
1.3 Forms of Parallelism . . . 4
1.4 Abstract Models of Parallelism . . . 5
1.5 Research Aims . . . 5
1.5.1 Parallelism . . . 6
1.5.2 Language Support . . . 6
1.5.3 Query Optimisation . . . 6
1.5.4 Data type Extensibility . . . 7
1.5.5 Implementation . . . 7
1.6 Sample Database . . . 7
1.7 Dissertation Outline . . . 7

2 Related Work **9**

2.1 Structure of Analysis . . . 9
2.2 Models for Parallel Computation . . . 10
2.2.1 Criteria for Categorisation . . . 11
2.2.2 Task Parallel . . . 12
2.2.3 Data Parallel Models . . . 15
2.2.4 Hybrid Task and Data Parallel Models . . . 17
2.2.5 Functional Language Derivatives . . . 20
2.2.6 Dataflow . . . 29
2.3 Parallel Database Management Systems . . . 30

2.3.1 Criteria for Categorisation 30
2.3.2 Parallel Database Systems: Commercial 31
2.3.3 Parallel Database Systems, Research 34
2.4 Query Modelling Techniques 40
2.4.1 Sheard 41
2.4.2 SVP 42
2.4.3 Comprehensions 43
2.4.4 Structural Recursion 43
2.4.5 Fegaras 45
2.4.6 CROQUE 46
2.4.7 Comparison of Query Modelling Techniques 47
2.5 Summary 48

3 Requirements and Analysis **49**
3.1 Requirements for a Model of Parallelism for Data-intensive Applications . . 49
3.2 Application-level Parallelism 50
3.2.1 Motivation 50
3.2.2 Treatment in Related Work 51
3.2.3 Thesis Approach 54
3.3 Ease of Programming 57
3.3.1 Motivation 57
3.3.2 Treatment in Related Work 58
3.3.3 Thesis Approach 59
3.4 Architecture Independence 59
3.4.1 Motivation 59
3.4.2 Treatment in Related Work 59
3.4.3 Thesis Approach 60
3.5 Extensibility 60
3.5.1 Motivation 60
3.5.2 Treatment in Related Work 61
3.5.3 Thesis Approach 61
3.6 Correctness 61
3.6.1 Motivation 61
3.6.2 Treatment in Related Work 62
3.6.3 Thesis Approach 63
3.7 Efficiency & Cost 63

3.7.1 Motivation 63
3.7.2 Treatment in Related Work 64
3.7.3 Thesis Approach 64
3.8 Summary 65

4 Framework **67**
4.1 Approach 67
4.2 Decision Support Queries 67
4.3 Complex Applications 70
4.4 Summary 72

5 Application Level Parallelism **73**
5.1 Algorithmic Skeletons 73
5.1.1 `pipe` 74
5.1.2 `farm` 74
5.1.3 `hom` 76
5.1.4 `split` 78
5.1.5 `seq` 78
5.1.6 Application Interface 79
5.2 Examples 79
5.3 Summary 80

6 Theory and Query Model **83**
6.1 Introduction 83
6.2 Approach 83
6.3 Theoretical Basis 84
6.4 Data Modelling 87
6.5 Data Manipulation 87
6.5.1 Monoid Homomorphism Language (MHL) 88
6.5.2 Monoid Query Language (MQL) 89
6.5.3 An Example — average calculation 91
6.6 Extensibility 91
6.6.1 Grouping 92
6.6.2 Ordering 97
6.7 Parallel Evaluation Model 99
6.8 Query Optimisation 101
6.8.1 Example Proofs 102

6.9 Summary 106

7 Compilation Process **109**

7.1 Interaction Model — Outline 109

7.1.1 Summary 111

7.2 Monoid Definition Dictionary 111

7.3 Data Partition Dictionary 112

7.4 Monoid Dictionaries 112

7.5 Monoid Homomorphism Rewrite Database 113

7.6 SQL to MQL 113

7.7 MQL to MHL 115

7.8 MHL Rewrite System 115

7.9 Parallel Type Checker 117

7.9.1 Monoid Transition Type Checker 117

7.9.2 Symbol Table Construction 119

7.9.3 Resolution 121

7.9.4 Data Requirements Generator 123

7.10 Object Code Generation 124

7.10.1 Program Generator 124

7.10.2 Resource Allocator 127

7.10.3 VPT to Machine Node Mapping 128

7.11 Summary 131

8 Runtime Architecture and Implementation **133**

8.1 Parallel Machine Software Architecture 133

8.1.1 Master Runtime System 133

8.1.2 Worker Runtime System 135

8.2 Prototype Implementation 146

8.2.1 The AP1000 and Related Implementation Issues 146

8.2.2 Discussion 151

8.3 Summary 152

9 Performance Evaluation **153**

9.1 Introduction 153

9.1.1 Parallel Performance Metrics 153

9.1.2 Terms of Reference 154

9.1.3 Structure 155

9.2 Buffer size variation . . . 155
9.3 Two–level Processor Topology . . . 156
9.4 Three–level Processor Topology . . . 162
9.4.1 Fixed Number of Disk–Cells . . . 162
9.4.2 Varied Number of Disk–cells . . . 167
9.5 Join Queries . . . 169
9.6 Transformed Queries . . . 170
9.6.1 Ax 7 . . . 170
9.6.2 Ax 9 . . . 173
9.7 Summary . . . 174

10 Conclusions **175**
10.1 Summary . . . 175
10.2 Future Work . . . 178

A Notation **181**

B Alternative Formulation — grouping **183**
B.1 The Grouped Monoid . . . 183

C Transformations **187**
C.1 Introduction . . . 187
C.2 Legend . . . 187
C.3 Production of Partial Results . . . 188
C.3.1 Qualifier Interchange . . . 189
C.3.2 Qualifier and Filter Interchange . . . 189
C.3.3 Filter Promotion, Simple Filters . . . 189
C.3.4 Filter Promotion, Complex Filters . . . 190
C.3.5 Filter Promotion – from inner to outer comprehension . . . 190
C.4 Redundant Qualifier Elimination . . . 190
C.5 Join . . . 191
C.5.1 Filter and Map Promotion through Join . . . 191
C.5.2 Partial Result Generation . . . 191
C.5.3 Join Generator Elimination . . . 191
C.5.4 Join Filter Demotion . . . 191
C.6 Rules for Primitive Boolean Monoids . . . 192
C.6.1 Eliminating Redundant Comprehensions . . . 192

C.6.2 All . . . 192
C.6.3 Reordering Terms within Monoids Over $\mathbb{B}$ . . . 192
C.7 Optimisations based on Collection Identities . . . 193
C.7.1 General Collections . . . 193
C.7.2 Set Identities . . . 193
C.8 Miscellaneous Identities . . . 194
C.8.1 Cascading Map Operations . . . 194
C.8.2 Cascading Filters . . . 194
C.8.3 Cascading Map and Filter . . . 195
C.9 Implementation Based Improvements . . . 195
C.9.1 Use of Indexes . . . 195
C.10 Discussion . . . 195
C.10.1 Vertical Loop Fusion . . . 196
C.10.2 Horizontal Loop Fusion . . . 196

D Compilation Process **199**
D.1 Symbol Table Construction Algorithm . . . 199
D.2 VSL Normalisation Algorithm . . . 203
D.3 Treatment of Idempotent Predicate Terms . . . 204
D.4 Data Requirements Generation Algorithm . . . 206

E Sample Database — SR–CD **207**
E.1 cd Element . . . 207
E.2 distributor element . . . 207
E.3 track Element . . . 208

F Evaluation Queries **209**
F.1 Main Queries . . . 209
F.1.1 `1.112bag*all+atts` . . . 209
F.1.2 `2.98bag*all+atts` . . . 209
F.1.3 `2.98bag*1+atts` . . . 210
F.1.4 `15.3bag*1+atts` . . . 210
F.1.5 `17.3bag*1+atts` . . . 210
F.1.6 `100bag*1+atts` . . . 211
F.1.7 `100bag*3+atts` . . . 211
F.1.8 `100bag-and-max100*1+atts` . . . 211
F.1.9 `min100*1+atts` . . . 212

F.2 Join Queries 212
F.2.1 `CDS100*TRACKS100bag-1+att` 212
F.2.2 `CDS100*TRACKS100bag-1+att` 212
F.2.3 `CDS100*TRACKS100sum` 213
F.3 Optimised Queries 214
F.3.1 Qualification, Ax 9 214

List of Figures

4.1 Decision Support Interaction Model 69
4.2 Complex Applications Interaction Model 71

5.1 `pipe` Process Pipeline 74
5.2 Pure `farm` 75
5.3 Independent `farm` 76
5.4 Guarded `farm` 76
5.5 `hom` 77
5.6 Implementation of Join Using the `hom` Skeleton 78
5.7 `split` 79

6.1 Homomorphism Evaluation 100

7.1 Decision–Support Interaction Model 110
7.2 PTC and Object Code Generator 118
7.3 VPT for a `hom` skeleton 127
7.4 Annotated skeleton program 129
7.5 *reuse(i)* VPT to machine cell grid mapping for `hom` 131

8.1 Parallel Machine Software Architecture 134
8.2 Worker Runtime Software Environment 136
8.3 Skeleton Graph Depicting a Two–way Join 142
8.4 Skeleton Code Segment with a `pipe` within a Nested `hom` Expression 143
8.5 Hash–join Skeleton Tree 144
8.6 The Fujitsu AP1000 — External View 146
8.7 The Fujitsu AP1000 — Internal Structure 147

9.1 Amdahl's Measure 154
9.2 TRACKS Bag100*1+atts-(364KB) 156
9.3 TRACKS sum100*1+atts-(4B) 157

9.4 Two–level Process Graph 157
9.5 1/t vs Processors with Configurations: 1 , 1_2 , 1_4 , 1_6 , 1_8 , 1_12 , 1_14 , 1_16 , 1_20 , 1_24 159
9.6 Speedup vs Number of Processors with Configurations: 1 , 1_2 , 1_4 , 1_6 , 1_8 , 1_12 , 1_14 , 1_16 , 1_20 , 1_24 160
9.7 Inverse Time Graph for Configurations: 1_12, 1_2_12, 1_3_12, 1_4_12 163
9.8 Inverse Time Graph with Configurations: 1_24, 1_2_24, 1_3_24, 1_4_24, 1_6_24 . . 164
9.9 Speedup vs Number of Processors with Configurations: 1_12 , 1_2_12 , 1_3_12 , 1_4_12 165
9.10 Speedup vs Number of Processors with Configurations: 1_24, 1_2_24 , 1_2_24 , 1_3_24, 1_4_24 , 1_6_24 166
9.11 Inverse Time Graph for Configurations: 1_30 (900), 1_2_28 (964), 1_3_27 (1000), 1_4_26 (1038), 1_5_25 (1080) 167
9.12 Speedup Graph Configurations:1_30 (900), 1_2_28 (964), 1_3_27 (1000), 1_4_26 (1038), 1_5_25 (1080) 168
9.13 Inverse Time Graph for Join Queries 169
9.14 Speedup Graph for Join Queries 170
9.15 Unoptimised and Optimised Skeleton Programs for Ax 7 171
9.16 Inverse Time and Speedup Graphs for Application of Ax 7 171
9.17 Inverse Time Graph for Application of Ax 7 – Query 2 172
9.18 Inverse Time Graph for Application of Ax 9 173

List of Tables

2.1 Comparison of Models for Parallelism 13
2.2 Comparison of Parallel Database Systems 32

5.1 Format of the Final Result 80
5.2 Complex Application 81

6.1 Examples of Free and Ground Monoids 88
6.2 MHL BNF 88
6.3 MQL BNF 90

7.1 SQL and equivalent MQL forms 114
7.2 MQL to MHL translation 116
7.3 Sample symbol table 120
7.4 VSL 122
7.5 Cartesian Product – Source 125
7.6 Cartesian Product – Processed 126

8.1 Summary of Allocation Algorithms 138

9.1 Breakdown of Evaluation Times with a Configuration of 1_24 — 1 158
9.2 Breakdown of Evaluation Times with a Configuration of 1_24 — 2 161
9.3 Breakdown of Evaluation Times with a Configuration of 1_2_24 164
9.4 Breakdown of Evaluation Times with a Configuration of 1_4_24 166

A.1 Notation 181

D.1 Algorithm for the Construction of the Symbol Table 201
D.2 Symbol Table Key 202
D.3 VSL Normalisation Algorithm 203
D.4 Data Requirements Generation Algorithm 206

Chapter 1

Introduction

Parallel processing denotes the concurrent independent execution of two or more co-operating processes in solving a problem. Parallelism makes more computational performance available than in a single processor. Even if single-processor speed improvements continue on their upward trend, parallel computation is likely to be more cost-effective for many applications than using leading edge single-processors. This is especially evident with respect to the availability of cheaper parallel computers based on industry–standard components such as the $\times 86$ processor. These machines are now beginning to compete with state–of–the–art uniprocessor configurations. This has fuelled renewed interest in parallel computing as a vehicle by which general–purpose applications may experience significant performance improvements.

In spite of the amount of interest since its advent, parallel computing has not proved to be a panacea in delivering cost–effective performance. Providing software support to realise the potential benefits of massively parallel computing has proven more elusive than fabricating ever more powerful and cheaper parallel computers. Programming languages developed expressly to harness the power of parallel hardware have been plagued by difficulties with programmability, poor portability, poor scalability, and in the majority of cases, lack of a formal foundation for parallelism. Major factors that have hindered the widespread adoption of parallel computing include:

Historical Parallel computer manufacturers have targeted high–performance scientific and numerical applications that are readily parallelisable.

Software Environment The lack of suitable software support tools at a high-level of abstraction to aid the programmer in solving the problems of data partitioning, identifying and coding parallel tasks, mapping these to real processors and managing communications, means that parallel application development still requires careful design and implementation by experienced programmers. Furthermore, it is still a time–consuming and error–prone procedure for anything other than simple tasks.

Efficiency Some problems are inherently difficult to parallelise. Even for problems that are amenable to parallelisation, dividing the problem up in a sensible and efficient manner between processors is critical to achieving good performance on a parallel machine.

In practice, improvements in performance commensurate with increases in resources are difficult to achieve due to overheads such as communication time and purely sequential parts of code.

Theory The theory behind sequential computation preceded the computers themselves, whereas for parallel computation it has had to develop alongside the technology. As a result, the theory required for parallel computation is primitive. In addition, it is significantly more difficult to ascertain the correctness of parallel software compared to that of sequential software.

Portability In order to produce portable applications which run across different uniprocessor machine architectures, developers carry out separate source code compilation for each architecture. Although the process of porting code is complicated by the presence of different operating systems, there is little need for complete software re–engineering. Since there are many more varieties of parallel architectures,[1] the task of producing portable code for parallel machines is significantly more difficult and costly than the uniprocessor case.

Parallel computing has been confined to readily parallelisable, computationally bound scientific and engineering problems that can only be solved with an acceptable response time through parallel computation. However, within the last decade, it has also been applied to the parallel evaluation of relational database queries in data–intensive applications.[2] Application areas that constitute an important part of industrial and commercial computing such as database, data–warehousing and data–mining applications have benefited from improved response times for decision–support queries and increased transaction throughput for on–line transaction processing applications that would be difficult to achieve with uniprocessor hardware platforms. This success has been facilitated by the easily parallelisable set operations in the relational data model, upon which SQL is based. A factor that has also contributed to this success is the use of a standard database query language, the declarative SQL language, which has delivered both ease of programming and portability.

A data–intensive application utilising parallel relational database technology benefits from the parallel evaluation of the database queries at the server. The non–query components of the application however run sequentially on a client machine since they are usually developed in a procedural language. The development of multi–threaded operating systems running on small configuration multi–processor client PCs has enabled cost–effective client–side parallelism to be extracted from the non–query application components. Although this supplements the query parallelism within the parallel database server, it requires the developer to identify and re–code program segments that are independent and can be parallelised using threads. Further, the underlying model of client–side parallelism is not scalable or suited to data–intensive applications which typically consist of dependent operations that synchronise frequently. Aside from multi–threading, no software support that extracts parallelism from non–query components is available. Additionally, software

[1] Processors, memory and disks may be interconnected in different ways.

[2] A data–intensive application is defined as an application a significant part of which is occupied with the manipulation of large amounts of structured homogeneous or heterogeneous data.

support that enables parallelism to be exploited across both the query and non–query components of a data–intensive application in a *unified* way is absent.

This thesis proposes a model that supports unified parallelism for data–intensive applications. The model is based on a number of theoretical principles and allows the extraction of multiple forms of parallelism across both the query and non–query components of those applications that are currently supported by parallel relational database management systems. This leads to improvements in the running times of data–intensive applications.

1.1 Background

A brief overview of the essential concepts of parallel application development is carried out in the following sections. A computational problem is implemented as a parallel application using a specific application development environment that consists of a hardware platform and a software layer. The software layer provides a computational language that is based on an abstract model of parallelism that supports a number of different forms of parallelism. This exposes three inter–related areas which are reviewed in the next section:

- *parallel hardware architectures;*
- *forms of parallelism that may be exploited;*
- *abstract models of parallel computation.*

1.2 Hardware Architecture

Parallel computers consist of processors, memory, an interconnection network and in some cases, secondary storage usually in the form of magnetic disk. The way in which these components are put together determines the main characteristics of each parallel machine. Collections of one or more processors are bundled with communications to form *nodes* that are connected through an interconnect. The interconnect may take one of a number of forms such as a *mesh*, in which a constant number of connections to the network is required for each node; or a *hypercube*, in which the number of connections per node is a logarithmic function of the total number of nodes within the computer.

Parallel computers are commonly divided into two classes: *Single Instruction Multiple Data* (SIMD) and *Multiple Instruction Multiple Data* (MIMD). In SIMD architectures, a single processor fetches a single instruction stream but broadcasts this to a number of data processors. In MIMD architectures, groups of processors co-operate to solve *different* parts of the same problem, concurrently. MIMD computers form the larger proportion of parallel computers in use. There are two categories of MIMD architectures, *distributed–memory* MIMD and *shared–memory* MIMD. If each node has its own local memory and does not access values in other memories in the network, then the class of architecture is termed *distributed–memory*. Distributed–memory machines are classified by availability of secondary storage facilities at nodes. Those in which each node also consists of secondary storage are termed *non-shared disk* (or sometimes, *shared–nothing* in parallel database literature), and those in which each node has access to non–local secondary storage, termed *shared–disk*. In *shared-memory* systems, in contrast, all nodes have uniform access to all memory modules and all disks through a global main memory. In this case, inter–node communication is facilitated through shared–memory techniques as opposed to *message-passing*, which is normally used in distributed–memory MIMD architectures. Shared–memory systems offer simplicity of program development and good performance when the number of processors is small, however conflicting accesses to the shared-memory rapidly degrade performance for larger configurations. Shared-memory and shared-disk systems are limited by interconnect bandwidth or I/O channel bandwidth restrictions, which results in scalability problems.[3] Shared–nothing systems have been shown to grow more gracefully by the addition of more secondary storage or processing power at the expense of greater conceptual difficulties in breaking down problems.

1.3 Forms of Parallelism

The typical sources of parallelism that are exploited in massively parallel computing come from either *task parallelism* or *data parallelism.* Task parallelism is exploited through the concurrent execution of (different) functions over different data elements belonging to the same input stream. Typical forms of task parallelism are *pipeline parallelism*, as in *process pipelines*, and *farm parallelism*, in which a controlling processor farms out work to *slave* processors. Data parallelism on the other hand is exploited by the concurrent computation of portions of an output result relative to the computation of a single function by multiple processors.

Data–intensive applications are commonly implemented using database management systems. Database systems support two forms of parallelism: these are termed *inter–query* parallelism and *intra–query* parallelism. *Inter–query* parallelism, a flavour of task parallelism, is a direct result of the ability of database management systems to service multiple requests (transactions) concurrently. *Intra–query* parallelism refers to parallelism within the evaluation of a single query. Two forms of *intra–query parallelism* are normally identified: *intra–operator* and *inter–operator* parallelism. Inter–operator parallelism (a form of task parallelism) is obtained through the parallel execution of different operators within the same query. It can be used to execute producer and consumer pipelines or to execute independent subtrees in a complex query execution plan concurrently. Intra–operator parallelism (a form of data parallelism) may be utilised when a single operator in a query plan is executed using multiple processes, typically on disjoint pieces of the problem and/or disjoint subsets of the data. Intra–operator parallelism is based on fragmentation or partitioning, and is facilitated through parallel implementations of set operators such as selection, join and set intersection.

Presently, parallelism is exploited in database systems between transactions within different applications, and within the constituent queries of each application. However, levering multiple forms of parallelism *across* the whole of each transaction within the *same* application is not possible due to the lack of a unified model of query and non–query parallelism.

[3] Shared–memory machines with more than 100 processors are rare.

1.4 Abstract Models of Parallelism

Parallel applications are developed using a development environment based on a particular parallel programming language. A parallel programming language is the realisation of a model of parallelism. An abstract model of parallelism is an *abstract–machine* that provides certain operations at the programming level and is designed to separate software development from parallel execution detail. Realising an abstract model of parallelism that can support different classes of applications and that is efficiently implementable on each different class of parallel architecture is a hard problem.

The models of parallelism that incorporate features determined by the specific characteristics of a particular parallel architecture provide least abstraction. The developer is forced not only to contend with the task of extracting parallelism, but also deal with particular machine features. Applications developed using such paradigms require code re–engineering on a per application basis to realise portability. In order to ease the task of the developer, abstract models of parallelism abstract away machine details and concentrate instead on the core machine–independent features that are required to enable the development of efficient parallel applications.

The declarative language SQL that is implemented by parallel relational database systems realises an *implicit* abstract model of parallelism. The application developer does not need to be aware of the way in which queries are parallelised, and the characteristics of the parallel server machine. The use of SQL delivers portability — a number of parallel implementations of the language exist on different parallel hardware. However, data–intensive applications that are limited by the query and data modelling facilities provided in SQL must be deployed using less abstract models of parallelism, even if they would benefit from the application of parallel relational database technology. The lack of availability of a suitable abstract model at a high–level of abstraction forces developer to use a language that is based on a much more explicit model of parallelism. This complicates the development process since there exist few declarative and flexible parallel data–processing languages suited to the task.

It is interesting to note that the success of Java in delivering (sequential) application portability is largely attributed to the existence of a *virtual machine* that is based on a well–defined abstract machine model. The benefits of such a paradigm have still to be extended to parallel computing.

1.5 Research Aims

This thesis proposes a unified abstract model of query and application parallelism and a systems architecture and implementation for the support of data–intensive applications. In order to harness parallelism across the whole range of data–intensive applications, it is not adequate to exploit opportunities for parallelism within query and non–query components individually. This thesis argues that a unified approach to application parallelism is necessary in order to exploit all opportunities for application parallelism.

The rest of this section outlines the contributions of this thesis in a number of areas which are central to the development and parallel evaluation of data–intensive applications. These areas include a framework and language model for the support of both query and non–query parallelism, a declarative extensible query language for decision–support queries that integrates seamlessly with the unified model of parallelism, and an implementation of the decision–support part of the architecture. Expressions in the declarative language proposed in this thesis are amenable to source to source optimisations, a property that has proven necessary for efficient parallel evaluation of database queries in SQL.

1.5.1 Parallelism

A unified model for application–level parallelism that accommodates both query parallelism and non–query parallelism in data–intensive applications, is presented.

Query Parallelism

The model proposed for query parallelism is appropriate for decision–support queries. The query model is based on sound theoretical principles and well–defined parallel operational semantics. It is centered around the use of *monoid* structures to model collection–types, such as list or set. The basis for performing operations over instances of all monoids is limited to a single function, the *monoid homomorphism*, which performs a reduction of a collection through the element–wise application of a unary operator followed by combination with a binary operator.

Application–level Parallelism

In order to provide support for cross application parallelism for complex applications, the approach taken is to structure parallel programming using the composition of skeleton functions. Each of these provides a specified form of parallelism. Since the monoid homomorphism has well–defined parallel semantics,[4] the decision–support model integrates seamlessly with the model for exploiting application–wide parallelism in complex applications.

1.5.2 Language Support

In order to provide a declarative interface similar to the SQL language, a high–level language, the *Monoid Query Language* (MQL), is presented based on the decision–support model. Through examples it is shown that MQL can naturally and uniformly express operations on various collection types including set, bag and list. MQL is a high–level form for the core operational language, which is based on the monoid homomorphism. A language based on the monoid homomorphism, *Monoid Homomorphism Language* (MHL), is also proposed together with a suitable parallel hierarchical evaluation scheme. This language is more expressive than the relational calculus, upon which SQL is based.

1.5.3 Query Optimisation

In relational database management systems the query optimiser performs a vital task due to the declarative nature of SQL queries. Originally, few thought that relational systems

[4] And defines a stereotypical parallel form.

would be able to deliver response times comparable with the network database systems in use at the time. The reason why this has proven not to be the case is advancements in query optimiser technology. Some of the algebraic optimisations which are at the heart of the query optimiser in relational (and object–oriented) database management are equally applicable to the model presented in this thesis. In order to show this, a representative sample of these optimisations are proved formally.

1.5.4 Data type Extensibility

A core feature of the model is data type extensibility which enables parallel evaluations to be carried out on a *user–defined* data type. A parallel implementation is realised from the definition of a user–defined data type. This feature is largely a repercussion of the regime that is used to define a new data type.

1.5.5 Implementation

The complete implementation of the model for decision–support queries is presented. This consists of a compiler which produces MQL from SQL input, a translator from MQL to MHL, a type checker that ensures correct parallel evaluation, a source to source optimiser for MHL expressions, and a code generator. Optimised MHL expression may be executed on the parallel back–end machine, which is a 128–processor Fujitsu AP1000 parallel computer (a distributed memory MIMD machine). The back–end run–time software architecture consists of a master process, which is run on the parallel machine's host computer, and a number of worker programs, which executes on each node on the parallel machine.

The decision–support implementation is capable of supporting applications which currently use relational database technology and SQL in particular. The models which have been developed for decision–support and complex applications are abstract–machine independent in that neither is reliant upon any particular class of parallel machine.

1.6 Sample Database

An application and data from the Swiss Radio Corporation's compact disk (CD) catalogue (*SR-CD*) is used to evaluate the implementation for decision-support queries. Swiss Radio has a large collection of CDs, the details of which, track by track, are recorded in a database which DJs search in real time to satisfy callers to live chat shows. The main retrieval method is via individual title words, which are searched in combination. The database consists of around 100,000 CDs with accessions of 7,000–10,000 per year. Each CD has 10 tracks on average.

1.7 Dissertation Outline

Chapter 2 reviews the previous literature in the three key areas directly relevant to this thesis: *models for parallelism*, *parallel database management systems* and *query modelling*.

Chapter 3 presents a critique of the related work and determines the main requirements that a core model directed at parallelism for data–intensive applications must satisfy.

Chapter 4 briefly describes the overall proposed architecture for both decision–support and complex applications.

Chapter 5 outlines the constituents of the model which supports complex applications by describing a core set of skeleton functions. An example of a simple database application is also presented.

Chapter 6 presents the theoretical background upon which the decision–support model is based. This chapter defines both the user–level MQL language and the operational MHL language.

Chapters 7 and *8* describe the front compiler and back–end runtime implementation of the decision–support model respectively. Special attention is paid to the task of semantic type–checking of MHL expressions to ensure correct *parallel* evaluation (in Chapter 7). The back–end run–time environment which executes on the parallel machine is presented in Chapter 8.

Chapter 9 presents an quantitative evaluation of the model using queries from the sample application.

Chapter 10 describes a summary of this research and indicates areas of possible future work.

Chapter 2

Related Work

Chapter one outlined the need for an abstract model of parallelism that unifies the parallelism in both query and non–query components of data–intensive applications. This chapter presents a critical view of the work pertinent to this thesis. The shortfalls in the work and related technologies are collated in chapter 3 which identifies a set of requirements for such a unified model of parallelism.

2.1 Structure of Analysis

Data-mining and statistical applications require support for complex collective tasks which consist of a number of inter–related database queries bundled with segments of specific code that implement analysis and display functions. Server–side parallelism in such applications has been achieved through the parallel evaluation of queries on parallel relational database management systems. Since support for parallelism within the client can only be achieved through the development of multi–threaded code in the non–query component of the application, a review of research into models for general parallel computing is carried out. Hence, the first part of this survey (§2.2) deals with abstract models of parallelism that are applicable to data–intensive applications. This part gives a non-exhaustive review of the approaches taken to incorporating typical forms of parallelism into existing language models.

Since server–side parallelism is used in evaluating database queries, a review of related work in commercial and research implementations of parallel database systems is carried out. The second part of the survey (§2.3) deals with work in the area of parallel database systems, both in the commercial domain and in the research domain. Work in this area is important since a significant part of the target applications under review require the processing of large amounts of data; a process which has lent itself well to the exploitation of parallelism by parallel database management systems and query engines.

Query processing and query optimisation in both sequential and parallel relational database systems are usually treated in an ad-hoc way. One of the reasons cited for the relatively slow adoption of parallelism in data–intensive applications, especially large–scale[1] parallelism, is the lack of abstract well-defined models of parallelism that are based on

[1]More than 100 processors.

theoretical principles. To address both the need for an abstract model of parallelism for queries, and the need for modelling queries for post-relational database systems, the third section of this survey (§2.4) deals with query modelling techniques. This review is also relevant to query language extensibility, query processing and query optimisation; areas that are also largely treated in an ad–hoc way in parallel relational database management systems.

The structure of this chapter is given below:

- Models for Parallel Computation (§ 2.2)
  - Criteria for Categorisation (§ 2.2.1)
  - Task Parallel Models (§ 2.2.2)
  - Data Parallel Models (§ 2.2.3)
  - Hybrid Task and Data Parallel Models (§ 2.2.4)
  - Functional Language Derivatives (§ 2.2.5)
  - Dataflow (§ 2.2.6)
- Parallel Database Management Systems (§ 2.3)
  - Criteria for Categorisation (§ 2.3.1)
  - Commercial Database Management Systems (§ 2.3.2)
  - Research Database Management Systems (§ 2.3.3)
- Query Modelling Techniques (§ 2.4)
- Summary (§ 2.5)

2.2 Models for Parallel Computation

An abstract model of parallelism is an abstract–machine which provides certain operations at the programming level and is designed to separate software development from parallel execution detail. Abstract models of parallelism fall into two main categories: those that are realised as explicit parallel programming languages, and those that are realised as implicit parallel programming languages. The other main approach to parallel programming is through the application of parallelising compilers.

Using a parallelising compiler is an attractive approach since it takes an existing sequential program and creates a parallel program from this techniques such as dataflow analysis, dependence analysis and control flow analysis. The programmer does not need to consider the properties of the target parallel architecture and is not burdened with the task of problem decomposition, processor allocation and dealing with communications. Parallelising compilers perform reasonably when the application can be easily mapped to a data parallel model. However, there are a number of drawbacks to parallelising compilers. Since the compiler is necessarily conservative about the way in which parallelism is exploited, it will usually not make use of the available parallelism within the application. In order to address this concern, some parallelising compilers allow the user to specify compiler directives through annotations. These directives are both compiler and language specific, and possibly even architecture dependent, thereby sacrificing portability. Another drawback is that parallelising compilers are forced to parallelise a sequential solution, whereas more efficient parallel solutions may exist. One notable, often neglected, area in which parallelising compilers have been particularly successful is that of parallelising relational database queries using parallel relational database systems.

This survey mainly deals with the remaining two approaches: implicit and explicit parallel languages. Explicitly parallel languages require the programmer to stipulate where and how task and data parallelism may be utilised in a program. Languages that are implicitly parallel have a high level of abstraction so that the programmer no longer needs to worry about decomposing the problem into distinct components and managing resource allocation, synchronisation and communication between components. Examples of implicit parallel programming languages are functional and dataflow languages which have no explicitly parallel constructs. The models for parallelism that are reviewed in this section form a cross–section of those that have a particular applicability to data-intensive applications. A survey of general models for parallelism is provided in [ST98].

2.2.1 Criteria for Categorisation

A categorisation of the models and languages for parallelism that are discussed in this section is presented in table 2.1. The criteria which have been chosen reflect the real requirements of data-intensive applications. These include:

- *task decomposition* – does the programmer have to analyse the problem and break it down into parallel components?
- *definition of granularity* – is the programmer required to fix the granularity of parallelism? If so, can this be altered dynamically depending on applications' dynamic behaviour?
- *support for multiple collection types* – many systems support lists but data processing applications require a richer set of collection types.
- *data partitioning* – does the system support data partitioning or does the responsibility for this lie with the programmer? Can automatic data partitioning be based on an application's own data distribution requirements?
- *processor allocation* – does the system require the programmer to break down the problem to the level of physical processors (or an abstraction thereof)? does the system deal with scheduling?
- *communications* – does the model require that the programmer handles communication between components on different processing nodes?

Two other indicators which do not feature on the table are a model for parallel I/O, and ways in which the correctness of applications can be tested. Considering the former, none of the models that are investigated includes a parallel I/O model that enables access to disk drives at nodes on the parallel machine, or even a parallel file system.

The task of proving, or simply illustrating the correctness of applications is an important factor since parallel programs are more prone to error than sequential programs due to the introduction of concurrency. Hence, models that encourage declarative ways of stipulating the problem enable the user to evaluate the correctness of application more easily than procedural models. The models which are based on formal semantics, such as the *Bird-Meertens Formalism* and *algorithmic skeletons* can thus help to reduce errors.

One other important issue that is not discussed in detail is the ability of a general model to inform the programmer of the relative costs of different parallel algorithms. A cost model is especially important for parallel applications since poor implementations of parallel algorithms may make inefficient use of available processing resources. The only one of the models/systems reviewed that has a notion of cost evaluation is NESL § 2.2.5. However, the algorithmic skeletons approach used in P^3L also takes relative costs of skeletons into account when deriving a concrete evaluation plan.

Each of the models is now explored.

2.2.2 Task Parallel

Task parallel languages are well suited to applications in which there are multiple actions that may be performed independently, possibly in the form of a process pipeline. Task parallel programming languages encompass approaches ranging from those using standard communication libraries in sequential programming languages on MIMD machines to those exploiting a distributed–object paradigm.

Message Passing

Early models of parallel computation were individually designed for a single architecture and did not attempt to hide much of the detail of problem partitioning, mapping logical tasks onto processors, co-ordinating tasks and handling inter–process communications.

Communication on distributed-memory MIMD machines is based on message passing. The interfaces between individual nodes are through *send* and *receive* operations which are incorporated into procedural programs as library calls. The realisation that many message-passing features are common to different classes of distributed-memory parallel machines and that many facilities that are required at the programming level are common to different application areas, has led to standardisation work which has culminated in the *Message Passing Interface Standard* (MPI) [mpi95]. In addition to providing a range of message passing primitives, including point-to-point and broadcast, MPI also provides primitives for collective communication, data scattering and synchronisation. Common communication patterns may also be prescribed through the definition of groupings of processors.

MPI-2 [mpi97], the successor to the MPI-1.1 standard, includes provisions for dynamic process creation and parallel I/O. The parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between processor memories and files. Data access to files is performed in typed units as opposed to the byte stream abstraction normally associated with flat files in UNIX. The parallel I/O system defined by the standard permits the implementation of a parallel file system in which both independent and collective access to a file are provided to logical groups of processors.

Property/ Model	Multiple Collection types	Data Distribution	Task Decomposition	Granularity Definition	Processor Allocation / Scheduling	Communications
MPI,PVM	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Mentat	$\times$	Data distribution manually coded within a class instance	Persistent Mentat Objects define grains of parallelism. Dataflow analysis used to split into tasks	$\surd$	$\times$ Logical processor to physical processor mapping done by system	Implicit function calls to other physically distributed Mentat objects (analogous to UNIX processes)
HPF/C*	$\times$	Array distribution structure defined by programmer. System handles partitioning	Programmer to specify which parts can be run in parallel (e.g. *parloop* and `independent` statement in HPF)	$\times$ number has to be defined for data parallelism	$\surd$ Location of tasks not required. Number of processors determined by programmer	$\surd$
Split-C	$\times$	$\surd$ - through spread arrays and distributed-"shared memory". Dynamic re-partitioning of data at run-time, bulk data "transfer" between tasks	$\times$ - data parallelism implicit. Task parallel parts coded as virtual threads through single piece of code	$\times$	$\times$ Dynamic creation of processes possible	$\surd$ Implicit through sync variables
CC++	$\times$ Although can be explicitly coded.	Specification of data splitting arrays given by programmer. No details required regarding individual processors	For data parallel operations, method coding using parallel loop constructs. For task parallelism, explicit constructs, e.g. `parBLOCK`. Supports dynamic task creation	$\times$ - logical processors at prog level mapped to physical	Explicit mapping of processors to enable locality to be expressed. Logical objects are assigned to physical nodes	$\surd$ Single assignment variables used $\Rightarrow$ cf. shared-memory
Braid	$\times$	Data partitioning specification for a class instance given by programmer	$\times$ Grain for task parallelism	Grain of data parallelism per object needs to be defined	Logical to physical performed by system	$\surd$
SISAL	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
NESL	$\times$	$\surd$ - programmer not aware of nature of partitioning	Parallelism exploited on built-in sequence functions and the *apply_to_all* function. Data parallelism on sequence elements, task parallelism through parallel evaluation within nested sequences	$\surd$	$\surd$	$\surd$
P^3L, Darlington	$\times$	$\surd$ - programmer not aware of nature of partitioning	Data parallelism through skeletons, e.g. *map*, *reduce*. Task parallelism through *pipe* and *farm* skeletons	$\surd$	$\surd$	$\surd$ Arbitrary communications patterns may be modelled using Darlington's *DMPA* skeleton

Table 2.1: Comparison of Models for Parallelism

Although there is no explicit support for *local* I/O, a particular MPI-2 implementation may make use of I/O facilities that are local to each processing resource. Such a facility may thus be used by application code.

Within the domain of heterogeneous workstation environments, PVM (Parallel Virtual Machine) [Sun90], has gained acceptance as a message passing standard. As with MPI, point-to-point *send* and *receive* operations are available, as are group communication and synchronisation primitives.

Using a model of parallelism that is based on the use of communications libraries such as these requires that the programmer does all of the decomposition, placement, synchronisation and communications explicitly.

Evaluation

Message passing is extremely flexible and is applicable to most application domains. Both task and data parallelism can be explicitly coded as required for each application. With careful design, it is possible to extract the appropriate level of parallelism suitable for a particular application. Since implementations of communications libraries such as MPI [mpi97] exist for a variety of hardware platforms software should be portable.

However, in general, it is extremely difficult to build software using models based on message passing, since both correctness and performance can only be achieved with painstaking effort. Further, data types such as set and bag are not supported at the programming level, so mechanisms for supporting these would have to be coded explicitly as required. Since the semantics of the application are embedded in the implementation, the task of verifying the correctness of a parallel implementation is not viable. Similar problems are encountered for predicting performance from code inspection, which is virtually impossible. The lack of a local I/O model means that it becomes difficult to predict the performance of applications which would benefit from I/O parallelism, namely those which retrieve and process large amounts of data from secondary storage. Although message passing is extremely flexible, developing complex applications requires a high level of expertise and is time–consuming and error-prone. Further, much of the time spent developing data–intensive applications using explicitly task parallel techniques is in duplicating many of the features that are already available in database management systems.

Use of Distributed Objects

The Mentat Programming Language (MPL) [Gro96] which is based on C++ is an example of a task parallel language that incorporates a shared distributed object space in which Mentat objects may invoke member functions on other (possibly remote) Mentat objects in order to read and modify their state.

The programmer uses application domain knowledge to specify those object classes that are of sufficient computational complexity to warrant parallel execution. Communication is implicit across method boundaries. The granularity of parallelism is the Mentat class member function, which is the only way to modify the state of a Mentat object. Parallelism is achieved through concurrent invocations on different methods that are all members of a

Mentat class. Data parallel computations can be mapped by hand onto the medium grain task parallel model supported by Mentat.

Objects, namely instances of Mentat classes or tasks, may be created dynamically. Shared state between Mentat objects can only be realised using other Mentat objects (there is no shared memory in the model). A class is defined as a *Mentat* class under any of the following circumstances: when a member function of the class is computationally expensive; when it performs a co-ordination role; or when it has shared state information, for example, shared queues. Mentat classes are either *regular, sequential* or *persistent*. A sequential Mentat object allows the user to control the order of invocation of member functions whereas in a Persistent Mentat Object member functions are invoked as soon as all of their arguments are available, irrespective of their order within the caller.

Mentat integrates a medium-grain data-driven computation model with the object-oriented paradigm, in which programs are represented as directed graphs. The language constructs are mapped to a macro dataflow model that is the computation model underlying Mentat. The vertices of the program graphs are computation elements that perform some function. The edges model data dependencies between the computation elements. The compiler generates code to construct and execute data dependency graphs. Thus inter-object parallelism is largely transparent to the programmer. However, the burden of managing the distribution of data and the generation of code to iterate over data elements rests with the programmer.

In Mentat, the programmer makes granularity and partitioning decisions and implements them using Mentat class definition constructs, while the compiler and runtime support manage communication and synchronisation.

Evaluation

Compared to the task parallel models based on the use of communications libraries, distributed object–based models relieve the programmer of the need to code communications explicitly. However, the programmer is not aware of the individual object accesses that result in communications, and hence, reduced concurrency. Further, the mapping from logical to physical processors is handled by the system. However, the programmer must use application specific knowledge to specify those object classes that make sufficient computational demands to warrant parallel execution. In such models, data parallel evaluations over the elements of the same class have to be coded explicitly.

2.2.3 Data Parallel Models

Data parallel programming is becoming an increasingly important tool for exploiting parallelism in data-intensive applications, especially on SIMD and vector computers. Existing data parallel programming languages are mainly extensions of sequential languages that are familiar to users, for example, C* [QH90] and High Performance FORTRAN (HPF) [hpf93, hpf97] are extensions of ANSI C and Fortran respectively.

A data parallel language provides constructs for expressing that a statement sequence is to be executed in parallel on different data. The programmer must analyse the algorithm to find parts which can be executed in parallel, and map them onto parallel hardware. Data parallel languages are well suited to applications where the same action is to be performed on many different data items, for example, image convolution where each pixel can be processed in parallel. Data parallel models of parallelism encompass those in which the existence of parallelism is understood, but the granularity of parallelism which is exploited at runtime, the mapping of grains to processors, and synchronisation is handled by the system that implements the model. The two approaches that are used to extract data parallelism from the (data parallel) sequential language derivatives are based on parallel loops and on monolithic operations. Most of these extended languages use a special data parallel data type which resembles the array.

Even though the data parallel languages in use today incorporate some elements of abstraction, their semantics are based on the SIMD execution model. Therefore, it is hard to use these languages to express algorithms in an abstract, machine-independent way. This is likely to make programming in such languages error-prone and lead to portability problems. In addition, extending a sequential language with new parallel data types and operators can make the language complex, thereby negating some of the benefits of familiarity.

C*

In C*, which was originally developed for the Connection Machine, data parallelism on arrays is exploited using *parallel variables*. An array of one or more dimensions can be defined to have one of its dimensions as the "parallel" dimension. For example, the rows of a 2×2 array may be processed in parallel and columns serially. Any number of multiple processors which operate in parallel on designated parallel variables can be created.

C* has a number of faults [HTP92], which include lack of orthogonality for constructs (e.g. pointers may not be stored in parallel variables), and the necessity to code different functions to operate on scalar and parallel variables. C* is cumbersome to program due to peculiarities that are influenced by the architecture of the Connection Machine in particular. Since task parallelism cannot be supported or coded manually by the programmer, C* is unsuitable as the basis for a model of parallelism for data–intensive applications.

Fortran Derivatives and HPF

High Performance Fortran (HPF) is the result of efforts towards the standardisation of Fortran language derivatives and contains a set of new constructs and extensions to FORTAN 90. It also has features such as compiler directives for informing the compiler that operations in a loop can be executed in parallel, the (explicit) allocation of data structures to processes, and data parallel operations such as system and user-defined reductions. HPF also requires the programmer to define the granularity of execution. However, the data communications necessary to interchange elements between parallel processes is handled by the system.

One of the main approaches used to capitalise on parallelism in many FORTRAN dialects utilises *parallel loops* in which conceptually independent iterations of a loop body are executed concurrently. However, ensuring that the loops do not actually reference the same locations, for example, indexing the same element of an array via a different index expression, is the responsibility of the programmer. The programmer is also required to give detailed information about distribution and alignment of arrays in addition to a specification of the independent parts of the program. Data types such as set and bag need to be explicitly coded into array operations since data parallelism may only be exploited within the array data type.

A number of derivatives of HPF have been developed, some, [SY97], integrating task parallelism into the data parallel model. However, user annotations are required in such approaches to flag independent data parallel tasks.

File I/O in Fortran has sequential semantics and a parallel I/O model for HPF has not yet been developed.

Evaluation

Data parallel languages simplify the programming of parallel computers since operations that require loops in low-level parallel languages can be written as single operations. The natural mapping of data parallel operations to architectures enables efficient implementations, at least for some architectures. In addition, many data parallel languages are extensions of sequential languages such as C or FORTRAN, which means that they are very flexible and familiar to programmers in the large.

The way in which data can be partitioned is at best physical, rather than data dependent. In addition, the choice of data parallel operations is made according to what can be efficiently implemented, so there is no basis for a formal software development methodology. Support for multiple data types, and hence, data parallel operations on them, is also weak. Languages based on types other than arrays include Parallel SETL [HK93], an imperative looking language which is based on data parallel operations on bags, and *parallel sets* [Kil92].

Data parallel languages eliminate the need for the programmer to map problems onto virtual or real processors, organise communications or synchronise processes, these being carried out by the compiler.

2.2.4 Hybrid Task and Data Parallel Models

Compositional C++

CC++, [CK93], is a parallel object-oriented notation that consists of C++ with six extensions. The aim of CC++ is to provide notations and tools for integrating data parallel and asynchronous computation models for both reactive systems executing on heterogeneous distributed environments and parallel platforms generally.

Data parallelism is provided through concurrent evaluations on a `distributed_class`. CC++ incorporates a number of constructs for specifying parallel execution: *parallel block* (for independent evaluation of statements giving task parallelism), *spawn* (fork), *parfor* (parallel for), *sequential*, and a mechanism by which parallel program components can communicate and synchronise: (`sync` objects). Task parallelism is provided through the abstraction of *Logical Processor Objects*, which are an abstraction of physical processing resources.[2] In order to implement an *independent farm*, for example, one of the logical processors is designated as the `initiator`, which can create additional processor objects by

[2]Such that each logical processor object is mapped onto a physical processing resource.

defining the code which the new logical processor is to execute, and the specific placement. Since each instance of a logical processor object is self-contained, existing C++ programs can be composed in parallel.

CC++ provides a general shared memory model which includes pointers to distributed memory. The effect of message passing and barriers can be obtained by using objects and sync variables (which extend to abstract and concrete data types in C++). CC++ reduces the overhead of managing communications by disguising communication in ways that fit more naturally into threads: communication is hidden by single-assignment variables. An attempt to read from one of these variables blocks the thread if a value has not been placed in it by another thread.

In CC++ distributed abstract data types may be defined, although data parallel operations on these have to made explicit, for example, by enclosure in a `parfor` (parallel for) statement. This is in marked contrast to mainly data parallel languages, such as Split-C (§ 2.2.4) [CDG+93], in which instructions are assumed to have data parallel semantics and the parts of a program to be executed by individual processors are flagged explicitly.

CC++ handles communications, synchronisation and mapping while the programmer manages problem partitioning and the selection of the granularity of exploited parallelism.

Braid

Braid [WG95] is a C++ derivative that aims to integrate task parallelism and data parallelism by adding data parallel extensions to Mentat [Gro96], which is task parallel.

The programmer specifies task parallelism by marking objects that are of sufficient computational complexity to warrant (independent) parallel execution, as in Mentat [Gro96]. In Braid, data parallel Mentat classes have to be labelled explicitly — they are designated by the prefix `data-parallel Mentat` in front of the C++ `class` keyword. Instances of data parallel objects must be created explicitly using the overloaded `new()` operator. The arguments to this function fall into three categories:

- *dimensions* of the data (the size of the data parallel object in the row and column dimensions respectively);
- *local communication category*, the type of local communication that occurs in terms of data parallel objects (for example, evaluating an average function over a number of neighbours);
- *non-local communication* which provides information about interactions between the data parallel object being created, and other data parallel objects within the application.

In addition, the dominant method of the object also needs to be defined. Based on this information, the compiler determines an object distribution pattern and passes information to the runtime system concerning the number and distribution of elements to be allocated for the object that is created.

To define a data parallel method within a data parallel class it is necessary to annotate the function to indicate the type of the particular data parallel operation and the subset size that Braid should use as the *unit* of the operation. The body of the member function must be written as if the operation is being performed upon a single subset, without reference to any *specific* element. The three types of data parallel member functions that may be defined are: *overlay* **(OVR)** functions, which are used to initialise instances of the data parallel object, *aggregate* **(AGG)** functions, which are applied to all elements of the data parallel object, and *reduction* **(RED)** functions. Facilities are provided for the cases in which data elements need to be split into sub–parts. This is achieved using *subset*–level data parallelism. In order to provide subset–level operations, the programmer is responsible for providing any iteration required within the subset while the compiler provides iteration across the subsets. Objects may be created dynamically and communication is implicit across method boundaries.[3]

In Braid, the programmer decides on the parts of the application that warrant parallel execution and indicates the form of parallelism that is to be exploited by the compiler. The compiler and runtime systems manage communication, data partitioning and synchronisation.

Split-C

Split-C [CDG+93] is a parallel extension of the C programming language that incorporates features from shared memory, message passing, and data parallel programming. Split-C is based on a SIMD model of parallelism in which each processor executes the same code image, although processors may follow distinct flows of control and join together at rendezvous points.

Split-C has a shared-memory feel in that it incorporates both local pointers and global pointers which address space on distributed memory multiprocessors. New data types can be explicitly coded to take advantage of the "virtual" shared-memory model. Split-C supports both static and dynamic allocation (or re-allocation) of objects/data, with facilities to vary the granularity and the way that the data is partitioned. Additions to an instance of a data structure can be performed from any processor, although the physical location of data cannot be prescribed. Data distribution is based on the array type and is of a physical, rather than a semantic, nature.

Task parallelism in Split-C has to be programmed explicitly on a per processor and configuration basis. A reference to message variables appearing in different threads invokes a message transfer from one processor to the others. "Global" variables may be updated explicitly (synchronously) or by using one of the asynchronous operations which handle inter-process transfer of typed data.

Split-C is extremely flexible: although task parallelism is not supported explicitly, it can be coded for individual processor and configurations. Nested data parallelism[4] can also be utilised in Split-C. Since the language is based on a low–level procedural language, the parallel behaviour of an application is embedded in the implementation. In addition, local I/O is not possible.

[3] Remote object access results in the communications which is handled by the underlying system.

[4] Also termed subset–level parallelism here.

Evaluation

CC++, one of the hybrid data and task parallel languages explored, is mainly task parallel, whereas the model underlying Split–C is mainly data parallel. Subset level data parallelism is incorporated into Braid and can be achieved in Split–C and CC++, in addition to both task and element-level data parallelism. Approaches that incorporate hybrid data and task parallel languages allow the definition of distributed abstract data types for which data placement has to be coded by the programmer explicitly at the structural level. Reasoning about the correctness of applications is still difficult since the application semantics are enveloped in a procedural language.

Using a hybrid task parallel and data parallel model the programmer identifies the parts of the program that would benefit from each form of parallelism using explicit constructs and maps these onto the target processors. Since locality will have a marked effect on communication performance, this almost inevitably requires knowledge of the interconnection network. Such software does not turn out to be portable, unless compilers are written for target architectures, or code is cross–compiled to a standard message passing interface, such as MPI [mpi97]. However, the compiler deals completely with synchronisations and communications and manages some aspects of mapping and data partitioning, thereby easing the burden on the programmer.

2.2.5 Functional Language Derivatives

An area in which implicit models of parallelism have been utilised is *functional languages.* Functional languages are based on Church's λ calculus. Functions are expressed as trees in which there are common subtrees for shared functions, which are evaluated using graph reduction [PJL92]. Computation rules select graph substructures (redexes), reduce them to simpler forms, and replace them in the larger graph structure until all computation rules have been applied. Parallel functional programming takes profit of the clear semantics of the model mainly through the exploitation of *referential integrity.* This allows different redexes to be evaluated concurrently, possibly on different computing agents. Thus, multiple processors may search for, and evaluate, redexes independently. The choice of evaluation rule influences the amount of parallelism that can be exploited. The speculative evaluation of redexes, some of which may not contribute to the final result, is wasteful of resources and may alter the semantics of non-strict functions. The actual structure of the graph changes during evaluation, so that load balancing may cause problems. Although implementations based on the functional model give respectable performance on shared memory parallel machines, in practice most implementations have been unable to avoid high overhead penalties in managing the high–degree of parallelism.

Of particular applicability to data–intensive applications is research in extracting parallelism from the evaluation of operations based on typical data structures. The *list* data type is a typical data structure in functional languages. Usually, the programmer has two ways of operating over lists: by scanning the list spine using *tail recursive functions*; or, by operating over lists with primitive operators that are included in the language and represent typical patterns of computation over lists. This second approach has led to work on the *Bird-Meertens Formalism* and *Algorithmic skeletons*, which will be reviewed shortly.

Tail Recursive Functions

Schreine [Sch93] considers the parallelism of functional programs and list processing functions in particular. List processing functions which return their results as streams of list cells are classified into four groups:

- *expanders*, which expand elementary objects whose evaluation into lists can be carried out in parallel;
- *folding*, which combines various elements of a list into a single object, the folding process itself is mainly sequential, even for commutative and associative operators;[5]
- *transformation*, which transforms a list into another in an element–wise fashion by iteratively creating the structures of the result list, the list elements being computed by parallel processes;
- *merge*, which is the element–wise merging of two lists into one in which both lists are processed element by element and "zipped" into one.

Tail–recursive forms of these functions are also considered. For example, for *expanders* the structure of the result becomes visible to the caller only when expand has reached the base case of recursion. Consequently, only finite list structures can be generated in a tail-recursive way. Further, cases of *downward communication* and *upward communication* are also considered. *downward communication* occurs when a level of recursion influences the next one. Information about the early processed part of the list may be propagated "down" the level of recursion, although this does decrease the amount of available parallelism. *upward communication* occurs when the current activation requires information from the recursive activation. Such classes of functions are difficult to transform into tail recursive form. Even though it is possible to express a wide range of tail-recursive functions that manipulate lists in Schreine's approach, the task of matching appropriate parts within program with these forms requires complex techniques.

Related work has been carried out by Heytens, [Hey92], who aims to extract parallelism from tail recursive functions within the context of "open-lists". An implementation of a distributed object store which uses dataflow evaluation techniques has been realised.

Maaßen [en92] follows a different approach. He proposes a choice of data structures and corresponding higher-order functions that enable the programmer to write programs which exhibit a *degree* of potential parallelism in order to overcome the inefficiencies which are due to extensive use of recursive function definitions. A number of data structures such as list, table and set are defined together with some type-specific functions. A number of primitive functions such as *construct*, single–element *delete*, *select* and higher-order functions such as *fold* and *map* are also defined. In addition to these, three other higher-order functions are also defined, namely *foldint* (for-loop), *iterate* (while-loop) and *fixed-point*.

Structure at a time

Efficient strategies for parallel evaluation of functions on lists can also be developed from the recognition that the forms used in the first approach (consideration of tail-recursive

[5] Although the folding of a list can usually overlap with its construction.

functions) are typical. This has approach has been investigated by Blelloch in NESL [Ble95], Skillicorn with respect to the Bird-Meertens Formalism [Ski90], and generally applied to Cole's paradigm of *structured parallel programming* [Col89] using *algorithmic skeletons*.

NESL

NESL [Ble95], is a strict first-order data parallel functional language which also supports fully nested data parallelism. Task parallelism is supported within nested vectors and across nested data parallel function calls. The data parallel *map*-like operator *apply-to-each* applies any function (built-in or user-defined) concurrently over each element of a sequence.

In addition to arithmetic and logic constructs, a number of other data parallel constructs based on sequences are also provided. An example is *sum*, which sums all the elements in a sequence. Constructs operating on nested sequences, such as those which perform flattening and nesting of nested sequences are also included. A two phase nesting operation that takes arbitrary nesting functions can be constructed by applying the function on all elements `x`, to give a sequence of pairs of the form `(f(x),x)`. Pairs with the same value for `f(x)` may then be nested using the `collect` function which takes a sequence of `(key,value)` pairs, and collects each set of values that have the same key together into a sequence (cf. the `GROUP BY` construct in SQL). Other collection oriented operations include a duplicate elimination function and sequence `union` and `intersect` functions.

A criticism of high-level data parallel languages that operate on collections is that it can be hard to determine the approximate running times a program by looking at the code. For example, in SETL [HK93] the time taken by set operations is strongly affected by the representation of the set. In the evaluation of programs, NESL incorporates the derivation of the asymptotic complexity (parallel complexity) of programs using two complexity measures for each function. These are a *work* complexity, which is defined as the running time if executed on a serial Random Access Machine (RAM), and a *depth* complexity, which represents the deepest path taken by the function – the running time if the function is to execute with an unbounded number of processors. Composition rules are used to combine the two complexity measures across expressions. For example, for the *apply-to-each* construct the work complexity for `sum` is the work complexity of the instantiations, the depth complexity is the maximum depth complexities of the instantiations. Using Brent's scheduling principle, [Bre74], the two complexities place an upper bound on the asymptotic running times for the parallel random access machine (PRAM). This information is used at compile-time to produce object code.

NESL also addresses portability concerns since it is compiled into VCODE [BC90], a stack-based intermediate language. Nested parallelism is flattened into the simpler flat data parallel VCODE using *segment descriptors* [BS90].

Evaluation

NESL incorporates a single data parallel data type, the sequence, although other parallel data types may be implemented in in terms of this. The declarative programming paradigm and extensive use of built-in data parallel operations eases the task of ascertaining program correctness for new data types. The language is flexible since facilities are provided to encode reasonably efficient parallel versions of some sequential algorithms, such as quicksort. However, the PRAM model, upon which the cost model in NESL is based, is not suitable for data-intensive applications which communicate large amounts of data. Although sequential I/O operations are available, there is no provision for parallel I/O.

In NESL, parallelism has to be explicitly coded, either with the use of the *apply-to-each* operator, in which case the function is applied in parallel to data elements, and/or with the use of the built-in functions, such as the parallel sequence cardinality evaluation function (#). The task of exploiting an efficient granularity of parallel tasks, co-ordination between parallel tasks and inter–process communication is all handled by NESL.

Homomorphic Skeletons and The Bird-Meertens Formalism

The homomorphic skeletons model is based on the Bird-Meertens formalism. The Bird-Meertens Formalism [Bir86, Ski90] consists of a set of theories built on a base algebra with unary and binary functions, each theory capturing the behaviour of a particular class of data structures. A Bird-Meertens theory begins with base types and extends them to new data parallel types using type functors, adding to the base algebra a set of second-order functions and laws which relate them to one another. The set of identities has a completeness property which guarantees that any property of the type can be expressed within the framework, thus enabling equivalent forms of a computation to be expressed. Furthermore, three additional results are of particular relevance to data parallel evaluation. The first states that a generalised *map* operation exists over the new type. For free types, the second additional results states that a generalised *reduction* operation is defined which computes the result of some operation by computing it first on the elements, and then recursively on larger aggregates.

Considering the theory of join or concatenation lists in the Bird-Meertens Formalism, any homomorphism on lists can be expressed as the composition of a *map* followed by a *reduce*. Put another way, the homomorphism can be structured as a parallel Algorithm consisting of two steps: a single parallel operation applied to each element of the structured type followed by a (logarithmic-depth) tree-like reduction. Similarly, for other type theories (graphs, sets) there are constructors such that homomorphisms on those types can be expressed as the composition of the *map* and *reduce* forms.

The control and communication aspects of the homomorphism can be encoded into a parameterised template which takes a unary function for element–wise application, and a binary function for combining the results produced from element–wise application.[6] This template for scheduling the individual computations and communications can be reused to compute many different homomorphisms by instantiations of the function parameters. Each implementation only needs to create an embedding of the communication pattern in the interconnection topology of each target architecture. For lists, if the reduction operation is linear time, the parallel algorithm will be $O(log\ n)$ where n is the length of the list.[7] Efficient implementations can be built for any target computer in which it is

[6] For a list homomorphism, the binary function is associative.

[7] Assuming the existence of a logarithmic binary tree in the target architecture interconnection network, such that all inter–node communications is between nearest neighbours.

possible to embed the standard topology in the physical interconnection topology.

An interesting development of the Bird-Meertens Formalism is BMF++ [Cro96], in which the parts of an application that can benefit from parallelism are written in terms of three (parallel) functions: *map, reduce and zip.* These function are used to implement parallelism in C++ programs.

Algorithmic Skeletons

Instead of searching for opportunities for parallelism everywhere within a function program, the *algorithmic skeletons* approach attempts to look for it in well-defined places. A number of parallel implementations of functional programming languages develop this approach by proposing the use of a set of *parallel forms* and/or well defined target architectures.

Algorithmic skeletons [Col89] are higher-order functions designed to capture a class of algorithmic parallel computations motivated by the Bird-Meertens Formalism. The approach of expressing parallel patterns as templates or higher order functions has received attention, both from the theoretical view and in practical implementation. A *skeleton* is a high level template description of some parallel operations in which part of the functionality is parameterised. Parallelism is expressed in a structure–qualitative way, by hierarchical composition of a restricted set of language constructs, corresponding to those forms of parallelism that are frequently encountered in parallel applications, and that can be efficiently implemented. This enables the separation of the meaning of a parallel program from its behaviour.

Skeletons are motivated by the problems associated with the formulation of static/dynamic process graphs for mapping over distributed memory architectures. Researchers have attempted to reduce the degrees of freedom associated with this problem, thereby making it more tractable. By taking into account just a small set of possible process graphs, and fixing the topology of the target architecture, the problems associated with mapping the process graph to a processing elements graph, load balancing, and handling changes in the process graph are eased. By fixing the architecture topology one degree of freedom is removed as the topology information can be included in language support tools once and for all. By fixing the process graphs taken into account when exploiting parallelism, problems can be studied and solutions found that could not have evolved in the general case due to the complexity of the problem.

Skeletons seem attractive as a basis for parallel programming because they provide a level of abstraction which succeeds in precisely separating semantics from parallel behaviour, allowing programmers to abstract away from the low-level implementation details. Much complexity can be hidden inside skeletons, easing considerably the compiler's task. Further, by using the performance models associated with each skeleton, a compiler can make decisions about resource allocation. Since skeletons abstract semantics from behaviour, it is more likely for the compiler to be able to produce an efficient implementation. Furthermore, these templates form the basis for a virtual machine.

Using transformations, it is possible to derive equivalences between templates once only and store them for future use. The parallelisation task thus now consists of tuning these templates, and transforming user programs into instances of these templates. A number of approaches that utilise skeletons are now reviewed.

Darlington *et al*

Darlington's approach [DGT93] is based on the observation that skeletons provide powerful abstractions over common patterns of parallel computation and can be expressed naturally through their semantically equivalent higher order functions (HOFs) in functional languages. Expressing skeletons as HOFs in a functional language is an attractive approach because HOFs are an integrated and natural part of functional languages, and not an ad-hoc feature that is bolted on. In fact, several HOFs found useful for describing skeletons are already widely used by functional programmers, not because of their content of implicit parallelism, but because they provide a concise and powerful way of expressing functionality. Examples of these higher order functions are `map`, `fold`, `filter` and function composition ($\circ$). In Darlington's approach a strong emphasis is placed on the use of program transformation techniques for transforming one skeleton or a group of skeletons into another in order to obtain a better mapping onto a particular architecture.

Many algorithms work by splitting a large task into several sub-tasks, solving the subtasks independently, and combining the sub-results. Trivial tasks (`t`) are solved (`s`) directly on the home processor: larger tasks are divided (`d`) into sub-tasks passed to other processors to be solved recursively. The sub-results are then combined (`c`) to produce the main result:

```
DC t s d c x = if t(x) then s(x)
               else c(map (DC t s d c) (d(x)))
```

Such patterns are captured by the `DC` skeleton (and similar ones in related works), which is a combination of two separate skeletons: `map` and `reduce`. Another skeleton is the `RaMP` — *reduce and map over pairs* — skeleton, which is an unusual form in that it defines an operation on two lists in which each object in one argument list can potentially interact with any other object in the second list. Each individual interaction is calculated and the results are combined to produce a result for each object. This skeleton mirrors the (non-indexed) nested-loop join algorithm. This skeleton is mainly intended for initial specification.

Also incorporated is a low-level skeleton which aims to capture general computational forms, the `DMPA` – *dynamic message passing architecture* – skeleton is used to realise inter–process message passing.[8] This skeleton is intended for dynamic algorithms where the interactions between the processors are determined at runtime. Recently, Darlington's group have followed the approach of using skeletons as a way of composing and co-ordinating concurrent code written in Fortran [DGTJ95].

Bratvold

Bratvold's [Bra93, MBBW94] approach is similar to Darlington's [D$^+$93] in that parallelism is expressed through the implicit use of HOFs in the functional program. The task specific code is written in a declarative, functional language which is amenable to transformation techniques. He considers the four main skeletons, which are `map`, `filter`, `mapandfilter` and `filterandmap`, in addition to the process pipeline, which is considered as a glue that hold various parts of a program together. Processor farms can be used for

[8] DMPA is *not* an algorithmic skeleton.

implementing arbitrary linear recursion (although not all functions will produce efficient farm implementations). Other forms of recursion may be implemented in a divide-and-conquer manner. Interestingly, [Bra92] presents a function *mapn* which applies an *f-ary* functions, f, *n-wise* over n data structures. This has not been presented elsewhere. It has applicability for modelling the combination of elements from corresponding data partitions as in parallel bucket–based implementations of the relational join operation.

Nelson and Snyder

Nelson and Snyder [NS88] provide a `Compute-Aggregate-Broadcast` skeleton. Algorithms are viewed as consisting of 3 basic steps, a *Compute phase* performing some basic operation, an *Aggregate phase* (usually tree-based computation) combining local data into one, or a few, global values, and a *Broadcast phase* returning global information, or a directive based upon it, back to each process. It may iterate on the three phases starting from the Compute phase. This skeleton may be seen to be particularly appropriate for cases in which the result of an aggregate operation is used. For example, consider a difference list in which each element in the result list is calculated by deducting the average of all elements within the argument list from each element.

Rabhi

Rabhi [Rab93] provides the most general *divide-and-conquer*-like skeleton, termed `recursively partitioned`. A problem is dynamically solved by recursively dividing it into sub–problems which are solved independently, in parallel. The final result is obtained by recursively combining the solutions to the sub problems. This form exploits speculative parallelism by building partial solutions and testing all improvements in parallel, discarding a partial solution when no further improvements can be made.

P^3L

In P^3L [DMO+92, BDO+95, DPP97], the language developed as part of the Pisa Parallel Processing Project, a program may be structured as the hierarchic composition of a fixed set of skeletons that can be efficiently implemented. The leaf nodes of the composition are written in a sequential host language (currently C++) such that an application becomes a composition of many parallel components. In P^3L, the compiling tools work out an optimised implementation for a specific architecture [BDJ+91] and output a static process graph implementation in C. This is then compiled for execution on a parallel machine.

Specifically, the approach is based on:

- a structured parallel language, called P^3L (*Pisa Parallel Programming Language*) which provides a collection of distinct constructs for pre–defined forms of parallelism. For each construct there is a set of implementation templates for each target architecture. An implementation template consists of a process graph model, which supplies information about the way in which some specific processes and channels must be arranged and mapped, and a performance model. Each implementation template does not correspond to a unique static network of processes.

- a set of compilation tools that realise portability by exploiting the structured features of P^3L. Using a limited number of constructs enables compiling optimisations that can achieve greater efficiency in the implementations. P^3L uses *construct tree optimisations* stored in a library of absolute rewrite rules,[9] timed rewrite rules and topological rewriting rules (which depend on architecture and processor interlinking).

- an abstract machine, P^3M, which exports features of the underlying architecture to the compiling tools which use this information to tune applications to a target parallel platform. P^3L does not require programmers to specify machine dependent factors such as process activations, the actual degree of runtime parallelism, scheduling, or interprocess communications. The target class of architectures are of (distributed-memory) k-ary n-cubes.

In addition to the `map, pipe, farm` (process farm), and `reduce`-like skeletons which are found in other work, P^3L incorporates a `loop` skeleton. This skeleton consists of an input channel, an output channel and a body like the other skeletons. However, the implementation of the *loop* construct allows the tasks coming from both the input and the output (*feedback*) channels of the loop to be concurrently computed. The output process receives the results from the loop body and depending on the value of the loop condition sends its result either over the feedback channel, or over the output channel of the loop if the final condition of the loop is reached. Input/output ordering is preserved using the tags which are assigned to new tasks by the input process. Feedback gives the state information which is necessary for each iteration of the loop. This allows tasks over the feedback channel to be integrated with the tasks over the input channel. The loop condition and the loop body may be subject to pipeline parallelism. Individual iterations of the loop may also be assigned to other processors, thereby giving the concurrency.

Recently, the Pisa group [DPP97, BCD$^+$97] have introduced some data parallel skeletons. The approach taken here is lower level with explicit collective data parallel operations. The distinguishing feature about this work is that it has been brought to implementation.

Skil

An alternative to P^3L's approach of using skeletons as a structuring mechanism for chunks of sequential code, is to embed algorithmic skeletons into a sequential language (C). This approach is taken in *Skil* [BK96], in which `C` is enhanced with higher-order functions, currying and a polymorphic type system.

Skil incorporates a flavour of the *map* and *reduce* skeletons which operate on the distributed array parallel data structures. Elements from an array may be explicitly distributed (block-wise, cyclic, block-cyclic) onto a number of virtual interconnection topologies. Promising comparative performance results are presented [BK96] for a number of applications which have been both coded in Skil and conventionally–coded in C on a parallel machine.

[9] Always *true* and optimising.

Evaluation of Algorithmic Skeletons

Algorithmic skeletons are simple and abstract structures for which efficient implementations are possible. However, since programs must be expressed as compositions of a fixed set of skeletons, the expressiveness of the abstract model is open to question. The exploitation of parallelism within skeleton-based programs lies mainly with the programmer who has to transform the high-level specification of an application into a set of lower-level skeletons relative to a particular target machine. However, some performance models and rules of transformation aid in this process.

There are skeletons that are more or less common to most of the skeleton–based approaches.[10] Most algorithmic skeletons include some kind of *divide-and-conquer* skeleton: Cole's `Fixed Degree Divide and Conquer` [Col89], Rabhi's `Recursively Partitioned` [Rab93], P^3L's `tree` [BDO+95] and Darlington's `DC` [D+93], of which Rabhi's is the most general. Variations on the *pipeline* and *farm* skeletons can also be found in these works: Rabhi's [Rab93] `Recursively Partitioned`, Cole's [Col89] `Task Queue`,[11] `pipe`, and `Crowd` [Gab90] (in which algorithms are defined as sets of co-operating independent processes, each an independent process communicating and synchronising with its peers).

The approach advocated by Darlington *et al* [D+93] is general, as skeletons are defined as higher-order functions in functional programs. This makes the specification of an application independent of any particular implementation. The higher-order functions considered correspond to both data and control parallel computations, although the `DC` skeleton may lead to too fine a grain of parallelism since the level of granularity is effectively a trivial task (t). Those considered in the Bird-Meertens Formalism [Ski90] are relative only to the data parallel model, which restricts the control flow to only a single thread expressed as a sequence or function composition.

Two approaches which use a host sequential languages have been reviewed: P^3L in which skeletons structure sequential code, and *Skil*, in which skeletons are embedded into a sequential programming language. P^3L includes a range of skeletons, including one based on parallel evaluations of a loop. This exhibits the same form of parallelism that is found in HPF's [hpf97] parallel loops. Each construct in P^3L is thought of as a dataflow module. The compiler tools have multiple implementations for each skeleton and perform optimisations on the whole of the process graph, whereas in *Skil* optimisations are not performed.

Although there exists a rapidly growing body of skeleton related work, performance results have been presented for few (exceptions [Bra94, BDO+95, BK96]). In addition, there is still very little experience with the approach applied in practice.

Algorithmic skeletons are used by the programmer to define the parallel semantics of the application. Some of the skeletons have equivalents in functional languages, so it is arguable whether the use of skeletons requires the programmer to define patterns of recursion over data types, or explicit parallel forms.

Skeletons abstract the implementation details from the programmer. The task of partitioning the application into parallel components, extracting a suitable grain of parallelism,

[10] A complete categorisation of algorithmic skeletons is proposed in [Cam96].

[11] A generalisation of the *farm* skeleton, except that it allows slave processes to produce new work to be performed by other slaves.

assigning grains to physical processors, prescribing communications patterns among processors and co-ordinating the complete computation are handled automatically. Hence, *algorithmic skeletons* provide a high-level declarative means of stipulating parallel computations. However, the skeletons work does not include facilities for parallel I/O. In addition, few data types are supported.

An alternative to requiring the programmer to expressly write code in terms of a fixed set of skeletons is to look for forms which can be reduced to these in a program. *PUFF* [Bus93] automatically extracts processor farm parallelism from linear recursive programs, namely ones for which the total number of recursive calls made grows linearly with the size of the top level argument. The principle used is that at each recursive call, the farmer should do only enough work to determine the arguments of the recursive call while all other work is packaged into a task and sent to workers. Since linear recursion is generally plentiful in functional programs, *PUFF* relies heavily on high quality instrumentation data to decide when parallelism is worthwhile.

2.2.6 Dataflow

Dataflow languages were originally developed to program dataflow architectures, which provide parallelism at operator level and require appropriate languages. Dataflow expresses computations as small operations. The execution of these operations depends solely on their data dependencies — an operation is computed after all of its inputs have been computed. The difference between dataflow and purely functional languages is that dataflow languages are not based on the λ-calculus. They do not allow the use of higher order functions, since these functions cannot be translated into a flow graph at compile time.

Operations that do not have a mutual data dependency may be computed concurrently. The operations of a dataflow program are considered to be connected by paths, expressing data dependencies along which data values flow.

Decomposition is implicit, since the compiler can divide the graph representing the computation in any way. The cut edges become the places where data moves from one processor to another. The occurrence of a name as the result of an operation is associated, by the compiler, with all those places where the name is the input of an operation. Processors execute operations in an order that depends solely on those that are ready at any given moment. Decomposition can be carried out automatically by decomposing programs into the smallest operations and then clustering to get pieces of appropriate size for the target architecture's processors. Communication is not made explicit in programs. Communication is always unsynchronised. A dataflow language, SISAL, is now reviewed.

SISAL

SISAL (Streams and Iteration in a Single Assignment Language) [sis97, KGP96] is a dataflow language. The SISAL parser translates the source program into the IF1 dataflow representation, from which the SISAL compiler generates C code which is compiled by the native compiler of the target parallel machine. Nodes in IF1 denote operations such as add or divide, while edges represent values that are passed from node to node. Functions are graph boundaries that surround groups of nodes and edges.

SISAL is one of the few applicative languages that has respectable performance. With its optimisations, most code run with comparable performance to the fastest versions in traditional imperative languages, although for distributed memory machines this is not true. Dataflow languages are simple to program and tend to work well for some application domains. Languages such as SISAL promote the construction of correct parallel programs by isolating the programmer from the complexities of parallel processing. SISAL exposes implicit parallelism through data independence and guarantees deterministic results.

Although dataflow languages shield the programmer from dealing with problem partitioning, granularity of parallel selection, mapping of parallel tasks to processors, and dealing with communication and co-ordination tasks, in practice, finding an efficient granule of parallel computation has proven to be difficult. Furthermore, dataflow languages are best suited to applications in which task parallelism can be exploited to a large extent. They have limited applicability to data-intensive applications in which the same operations are applied to many elements.

The next section deals with parallel database technology.

2.3 Parallel Database Management Systems

Exploiting parallelism from queries in relational database management systems (DBMSs) has long been seen as a possible solution to the need to provide real-time responses to complex queries involving large volumes of data. Often, a high-level query interface results in a number of complex DBMS queries, which together must have a short response time. Hence, in developing a unified model for application–level parallelism in data–intensive applications, which often consist of complex queries, it is necessary to consider the related work in the area of extracting parallelism from database queries.

The main reasons parallelism is relatively easy to exploit in relational query processing systems are: query processing is performed using a tree of operators that can be executed in separate processes and processors connected with pipelines (process parallelism); and, each operator consumes and produces sets that can be partitioned or fragmented into disjoint subsets to be processed in parallel (data parallelism). Parallelism is specifically exploited through *function shipping*, which involves evaluating tasks on processing nodes where data is available on local disks; and *data shipping*, in which remote data is shipped for processing.

2.3.1 Criteria for Categorisation

Table 2.3.2 shows a comparison of the parallel database systems that are reviewed in this section. The criteria for this evaluation are:

- *architectural considerations* - distributed memory MIMD machines are considered the most scalable. Does the system use a shared-memory, shared-disk or a shared–nothing model?
- *task parallelism* - is task parallelism exploited within queries? can task parallelism be exploited within the non–query components of applications?

- *hierarchical evaluation* - does the system use a hierarchical evaluation model which has been shown to be effective for some classes of architecture and queries [BV96]?
- *data model/language* - what data model is used? what language is used?
- *extensible data types* - is the system extensible with additional data types?
- *theoretical foundation* - is the model of parallel evaluation based on sound theoretical principles? are optimisations used proved correct within the parallel model? Does it include ad-hoc implementations of individual operations?

One of the criteria used is that of exploiting task parallelism within the application as a whole. Most relational database systems provide an API through which SQL queries are posed to the database. Object-oriented systems have C++ bindings and queries are integrated into the type system of the language. However, the only part which is subject to parallel evaluation, in both commercial and research systems, is the database component.

In the following two sections § 2.3.2 and § 2.3.3, commercial and research implementations of parallel database systems are explored.

2.3.2 Parallel Database Systems: Commercial

DB2 Parallel Edition

The IBM DB2 Parallel Edition [BFG$^+$95] has a shared–nothing architecture and runs on the RS/6000 SP hardware platforms and clusters of RS/6000s. Data placement issues can be prescribed through the DDL. These include selecting full or partial clustering across all or some of the available nodes, overlapped/non-overlapped assignment,[12] and the form of partitioning (e.g. hashing) across a set of logical nodes.

DB2 Parallel Edition features a cost-based query optimiser which takes into account different parallel algorithms based on query and partitioning information. For example, for cases where the data partitioning does not allow local joins to be performed the optimiser chooses between a broadcast join which performs well when one table contains many fewer elements than the other, and a repartitioned join. Performance results presented in [BFG$^+$95] show scalability up to 64 processors.

Tandem NonStop SQL/MP

NonStop SQL/MP [CC93, EGH95] is a commercial database system from Tandem computers based on the shared–nothing architecture, mainly intended for online transaction processing applications. Tables are horizontally partitioned (equally) across *disks* by key range values. Disk processes are associated with each disk and perform scan operations. Data is scanned by disk processes that implement operations such as table scans, selection, and some aggregate operations. The file system has knowledge of partitioned files and accordingly routes messages for the appropriate disk. NonStop SQL/MP supports

[12] Overlapped assignment refers to the case in which two tables share a subset of the nodes across which each of them is partitioned.

Property/System	Architectural Considerations	Task Parallelism: Query	Task Parallelism: Application	Hierarchical Evaluation	Data Model/ Language	Extensible Data types	Theoretical Foundation
Research							
DBS3	$\times$ - shared memory implementation, although compiler more general	$\surd$ - pipelined	$\times$	$\times$ - dataflow model, dynamic process allocation	(ESQL) - Relational + objects and rules	$\surd$ - new data types can be implemented by programmer	$\times$
XPRS	$\times$ - shared disk	$\surd$	$\times$	$\surd$	Extended Relational (as in Postgres)	$\times$	$\times$
EDS (developed into ICL's Goldrush)	$\times$ - shared disk	$\surd$ -pipelined	$\times$	$\surd$	(E)SQL + PROLOG, LISP	???	$\times$
Gamma	$\times$	$\surd$	$\times$	$\times$	Relational - SQL	$\times$	$\times$
Bubba	$\surd$	$\surd$	$\surd$ - limited through *do-end* block	???	Mainly relational + objects. Functional statements	$\times$	$\times$
Shore	Network workstation implementation. Objects may be suitably partitioned for a shared–nothing model	$\times$	$\times$	$\times$ - flat master-slave and between Shore servers for object exchange	Object-Oriented. C++ OML includes `apply`, `reduce`,`select`	$\times$ - can be manually coded. no support	$\times$
OMEGA	$\surd$	$\surd$	$\times$	$\times$	Object-Oriented	Can be explicitly coded	$\times$
Volcano	$\times$ - shared-memory implementation	$\surd$	$\times$	$\surd$	No particular data model. Individual operations/methods for manipulating individual data elements to be provided by programmer	$\surd$	$\times$
Commercial							
Tandem NonStop SQL	$\surd$ - shared–nothing	$\surd$	$\times$	$\times$	Relational - SQL	$\times$	$\times$
NCR Teradata	$\surd$ - shared–nothing	???	$\times$	$\times$	Relational - SQL	$\times$	$\times$
DB2 Parallel Edition	$\surd$ - shared–nothing	$\surd$	$\times$	$\times$	Relational - SQL	$\times$	$\times$
Oracle	$\times$ shared-disk and shared-memory	$\surd$	$\times$	$\times$	Relational - SQL + optimiser hints	$\times$	$\times$
Informix	$\surd$ - shared–nothing	$\surd$	$\times$	$\times$	Relational - SQL	$\times$	$\times$
Sybase	$\surd$ - shared–nothing	???	$\times$	$\times$	Relational - SQL	$\times$	$\times$

Table 2.2: Comparison of Parallel Database Systems

data parallelism and may use data replication in addition to partitioning. Producer–consumer pipelining is also exploited in the evaluation of query trees.

Joins are performed using nested-loops, sort-merge, or hybrid hash join algorithms and *may* require repartitioning or replication of data. Hash joins are used which perform better than a sort-merge when a small table is joined to a very large table. For join operations which require a different organisation of data, on-the-fly hashing techniques are used to achieve the required data distribution. No data re–organisation is performed for grouping instead, a separate grouping is performed for each partition followed by a combination of the results. NonStop SQL/MX is the latest development of the SQL/MP for both online transaction processing environments and data warehouses. It uses both a cost-based and a rule-based query optimiser. It is said to scale to 4080 processors with built-in fault tolerance, although no performance results for such large configurations have been reported.

NCR Teradata

The NCR Teradata [DSB89, TC88, BF97] is mainly intended for decision supports applications (for example, sales, purchasing, inventory, budget) and is also based on a shared–nothing architecture. However, NCR also provide platforms for Oracle's release 7.1 SMP based parallel database system. Teradata features full data de-clustering using hash-based techniques. The V2R2 release features a synchronised table scan feature that lets multiple queries execute full table scans on the same table at the same time.

Informix

The Informix-onLine Extended Parallel Server [inf96, Dav92] is based on Informix's On-Line XPS software architecture using a shared–nothing model on Hewlett-Packard Enterprise Parallel Servers.

Data partitioning is specified by the database administrator. A configurable pool of database server processes - termed virtual processors — is available for work. Each virtual process specialises in different resource management tasks such as CPU work, client/server communication, disk I/O, and database administration. Each virtual processor can spawn other threads to service new clients. Informix can access indexes in parallel, so it does not have to resort to table scans for each parallel query which is the case for Oracle (§ 2.3.2). Hash-based join techniques are used.

Sybase MPP

Sybase MPP [Cor97] is based on a shared–nothing model and uses the IBM RISC System/6000 SP architecture. For each database table, a number of partition options are available, such as no-partition, hash-based partition, and attribute key-range partition.

Oracle Server Release 7.1

Oracle's Parallel Server [ora96, ora86] runs on very large SMPs, and is hence of the shared-memory model. Platforms include the NCR series 5100 32 processor SMP product. More recently, MPP versions that utilise a shared–disk model have been available.

Data parallelism and pipeline parallelism are utilised. Data is partitioned on disk using a hash-based approach. Oracle uses nested loop and sort-merge joins only. Each query is split into several processes across multiple CPUs where each CPU is capable of performing independent disk operations. Oracle cannot use indexes when performing parallel queries. Oracle is a process-based database server that has incorporated a simulated threading architecture. Each user request is placed in a queue and Oracle starts up one or more database server processes to service these requests.

The Oracle interface allows the programmer to give explicit hints to the DBMS regarding configuration options and precise control of query decomposition and parallel execution.

Evaluation

Problems with existing commercial solutions include the following:

- the entry-level parallel database management systems in use today are based on a shared-memory approach, usually in the form of a number of Intel-based processors based on the SMP model. There are still scalability concerns to be resolved for large systems. MPP implementations have proved to be more cost-effective for larger configurations.
- implementations lack a theoretical basis for query evaluation or the application of optimisations. Where optimisations have been carried out, some have not been proven correct.
- the lack of extensible collection types – parallel relational database systems are limited by SQL which only supports set and bag.
- the lack of language flexibility - all commercial solutions are targeted specifically at SQL. There is restricted support to model other languages or other approaches, for example, Object Query Language (OQL).
- the lack of scalability - employing a flat evaluation structure instead of a hierarchical one and targeting an implementation at a specific architecture places a limit on the scalability of a solution.
- the lack of portability - most commercial parallel database systems are tuned to specific architectures.
- the significant data administration effort required to achieve good performance.

A review of parallel database systems research is conducted in the next section(§ 2.3.3).

2.3.3 Parallel Database Systems, Research

The lack of software support on parallel computers has made parallel database research a significant task. Conducting meaningful experiments has traditionally involved enormous effort and many researchers have been forced to perform simulations. The work presented in this section is a brief review of a number of parallel database projects.

DBS3

DBS3 [BDV95] is a parallel database system for shared-memory multiprocessors which has been developed on a KSR 1 with 72 processors in which memory is physically distributed. Memory itself is physically distributed even though it is virtually shared through hardware mechanisms. DBS3 exploits both data and task parallelism. DBS3 supports ESQL [GV92], an extension of SQL with objects and rules is supported.

The compilation process produces a parallel execution plan that is expressed in *Lera-Par*, which captures both operations and control. Lera-par is a dataflow language in which a program is represented by dataflow graphs whose nodes are operators (such as filter, map and join) and its edges are *activators*. Activators are the sequential unit of work, which denote a single tuple (data activation) or a control message (control activation). Pipelined execution is expressed by using a data activation between a producer node and a consumer node.

Each node of the execution plan is described by an operation structure, which uses a table of activation queues and a table of threads to consume those activations. In order to control start-up and load balancing time, DBS3 implements dynamic allocation of processors to operations, independent of the degree of static partitioning. A pool of threads are allocated for each kind of operation. This is done by allocating the queues of an operation's instance in a shared-memory segment such that all threads of a pool can access all queries associated with a particular operation. Therefore, the threads can execute code for any activation, which improves load balancing.

DBS3 statically partitions data by hashing on one or more attributes, relation fragments being distributed onto disks in a round-robin fashion. This is used for compile-time parallel query parallelisation and to reduce runtime interference.

SHORE

SHORE [Cea94] integrates file system and object-oriented database concepts and services, and has been implemented using the *Parallel Virtual Machine* (PVM) message passing interface [Sun90] on workstation clusters. A binding exists for C++.

SHORE's approach to obtaining data parallelism is based on the concept of *ParSets* ("Parallel Sets"). Objects (and their sub-objects) within *Primary Parsets* are subject to data partitioning, whereas *Secondary Parsets* represent logical collections of objects for which there is no implication of physical location. Support for *hash*, *range*, *random* and *no* partitioning is provided. Set-oriented queries over ParSets are parallelised based on parallel implementations of the relational algebra operators, the same technique employed in parallel relational database systems. Iteration within ParSets is provided through the *set-apply* operator, which applies a method to every element of a ParSet in parallel (*map*). A `select` operation which applies a predicate to a ParSet and a `reduce` operation, which can be used for aggregate evaluation are also provided.

The software architecture of each processing node consists of a SHORE server, a master, and a number of (multi-threaded) slave processes which are created and await queries from the master process. The master and slaves share a global object identifier space giving a flavour of a shared-memory system. During the execution of a query, a slave thread may require access to an object which is remote. The SHORE server which handles requests for objects handles communications necessary to retrieve the object from the remote SHORE server hosting the object. The master process combines results from multiple slave processes.

SHORE only deals with sets, so other collection types are not supported. In addition, data parallel evaluations are not scalable since there is only a flat master-slave hierarchy. Furthermore, it is not clear how the system would perform with complex objects. The lack of support for complex objects means that nested parallelism is not exploited. Moreover, it seems hard to drive this system to advantage.

XPRS

XPRS [SAS89] is a shared-memory software database machine which exploits intra-query parallelism. Its rationale is to implement Postgres on top of RAID disk arrays and the Sprite operating system. XPRS supports an extended relational model (as in Postgres [SK91]).

The XPRS optimiser produces an heuristically optimised sequential plan. At runtime, a query is decomposed into the desired number of multiple parallel queries, each exploring a portion of the search space required to solve the query. Hence, it relies on concurrent execution of different subtrees in a complex query execution plan, in what is termed *bushy* parallelism. Work is automatically allocated to the first available processor.

XPRS uses dynamic intra-query parallelism. The number of parallel plans into which a query is broken can be varied over wide ranges without compromising CPU or disk load balance and without incurring messages between separate computer systems. The optimiser decides between possible execution plans and decomposes the plan into a collection of *plan fragments*, which is the basis for parallelism. The XPRS execution engine schedules plan fragments to collections of processes at runtime based on several considerations including available memory, load average and synchronisation overhead, as in DBS3 [BDV95]. The query engine is mainly sort-based with the use of hash joins when appropriate.

The other major component of the XPRS design is a two-dimensional file system called the FTD running on top of RAID disks, which automatically balances arm activity among multiple disks. Due to the striping techniques used files are spread maximally across disks so as to reduce potential collisions for disk arms when exploiting I/O parallelism. Hence load balancing between processor and disk arms should be achieved. Performance results for XPRS have not been reported to date.

GAMMA

GAMMA, [DGH$^+$88, DGS$^+$90], is a shared–nothing relational database machine that exploits dataflow query processing techniques and uses hashing and pipelining of data between processors. It was originally developed on 17 VAX 11/750's connected to each other and the VAX 11/750 host via an 80 Mb/s token ring. Eight GAMMA processors have local secondary storage which is accessed through WiSS (The Wisconsin Storage System, [CDKK85]). Data partitioning may be either hash, attribute-range or round robin across these processors with disks. The other nine disk-less processors are used for join processing. GAMMA has been ported to an Intel iPSC/2 hypercube with 32 nodes, each with a local disk drive. GAMMA's main form of parallelism is intra-operator parallelism based on partitioned data sets.

GAMMA supports a purely relational model and query language. Intra-operator parallelism based on partitioned data sets is used, as in other parallel relational database systems. GAMMA uses traditional relational techniques for query parsing, optimisation [JK84], and code generation. The optimisation process is somewhat simplified by the use of hash–based algorithms for joins and other complex operations: GAMMA relies heavily on hashing for joins and aggregations. To enhance performance indexes are widely used. Selection predicates are also compiled into machine language procedures to minimise their execution time.

The operator design used in GAMMA gives each operator control within its own process, leaving it to the networking and operating systems software to synchronise multiple operators in producer-consumer relationships. This design lends itself to single-process query evaluation with the use of lightweight processes (threads). The data types which GAMMA can handle are fixed; data type extensibility is not explored.

EDS

EDS [WT90] is a relational parallel database system based on a shared-disk model which is influenced by the Bubba project (§ 2.3.3) [DV92]. Configurations consists of up to 256 processing nodes (PEs), each with a RISC processor a multi-threaded operating system executing, and a CISC processor which carries out the Operating System functions (including message passing). Simple queries run data parallel, whilst more complex queries are decomposed, thereby achieving both data and process parallelism in the query evaluation.

EDS supports PROLOG and LISP in addition to ESQL (see DBS § 2.3.3) — a superset of SQL that incorporates user defined abstract data types, objects, rules, integrity constraints, triggers and user defined aggregate functions. The RDBMS architecture consists of a *request manager* which compiles the database commands into machine code, and the data manager, which provides the runtime library for the execution of those commands. The request manager carries out logical and physical optimisations in addition to parallelisation during which the optimised query is converted into a set of localised processing element operations with explicit communication. The output of this last stage of optimisation is used as the input to the code generator, which in turn outputs executable C++ code. The data manager on each PE implements the relational execution model and executes the object programs produced by the request manager and supports the data distribution, access methods and indexing techniques required by the optimiser.

At runtime relations are horizontally fragmented over a set of processing elements, with the size of each partition, and the method of partitioning being programmer defined. Data parallel operations, such as scans, are carried out by a subset of the PEs, each of which forwards its results in a pipelined fashion to a collator.

The follow-up phase, EDS II, concentrates on integrating the parallel machine with its EMEX operating system and the EDS RDBMS, in addition to demonstrating and evaluating this with the PLANES commercial geographic information system. The METAL automatic translation application and the parallel Lisp language sub-system will also complete their integration and evaluation phases on the EDS platform. The results of this project have been incorporated into ICL's Goldrush [WC95] product.

OMEGA

OMEGA [GCKL93] is a parallel object-based system based on a shared–nothing architecture that supports a functional data model. An implementation of OMEGA has been carried out over a workstation cluster in which a front-end is connected to eight networked HP workstations communicating over Ethernet.

OMEGA consists of four types of processes: a *Schema Manager* (SM) which maintains database meta information (on the host machine); *Query Manager*, (QM) of which there is one associated with each active user and which supports a variant of OSQL [WLH90] (on host machine); *Scheduler Processes*, for scheduling multiple-processor queries; and *Operator Process* such that there is a single operator process employed at each processor participating in the execution of the query. Given a query, the QM parses the query and constructs a query tree in which user-defined names are replaced by their corresponding object identifiers. The query tree contains information such as the file each operator is to iterate over and the nodes holding the relevant fragments. The QM then sends the query tree to a scheduler process and awaits query completion. The scheduler on the host machine (of which there is one) traverses the query tree which is produced and schedules each operator for execution on an appropriate processor. The QM receives the final results of the query. Note that OMEGA does not employ a query optimiser.

OMEGA uses the Wisconsin Storage Structure (WiSS) [CDKK85] for object storage and object-based constructs. WiSS provides file services that include structured files, byte stream files, B^+ and hash indexes, a sort utility, and a scan mechanism. In WiSS, each record has a unique WiSS identifier, termed RID. An object identifier (or SID for sub-objects), is defined as the pair $\langle$processor_id,RID$\rangle$.

In OMEGA, the software that implements the query-processing functionality is independent of the physical location of the input data. Data parallelism arising from the concurrent execution of the same operation(s) over different data is utilised along with data pipelining through different operators. In addition to the standard relational operators, there are two other operators in the dataflow graph: *compute-and-route(OID)*, an operator to retrieve remote objects; and the multi-valued function or stored method operator. The input to each relational operator is either a record identifier, RID, or a SID.

Objects are horizontally partitioned across multiple nodes in such a way that an object is physically independent from its sub-objects in different classes, and the objects it references or contains. It is claimed that this flexibility is useful since each application may require a different physical database organisation in order to obtain the best performance. Since OMEGA allows application-dictated data placement strategies, it is susceptible to skew: it is the application programmer's responsibility to ensure that the object distribution strategy is suited to each application. The average number of object references that span inter-node boundaries impacts the scalability of the system due to inter-process communications and potential skew.

Bubba

Bubba [BCC+90] is a multi-programmed shared–nothing parallel database system with a dataflow model of computation. Bubba incorporates a virtual memory operating system that supports a single level store (into which the persistent address space is mapped), page level locking of the persistent space, and transaction-private address spaces for updated persistent data.

Bubba's programming interface is provided through FAD [DV92] — a declarative language originally based on FP [Bac81]. FAD allows arbitrary nested structures including sets and tuples, and supports arbitrary nesting of data structures.

In addition to the actions necessary for building, accessing, and manipulating data, FAD provides a fixed number of "action constructors" which include constructs for qualification, conditional, explicit sequencing of actions and a loop construct without updates. The construct action iteratively loops through applications of the loop function until the result is null. The loop state is the non-null result from the previous application of the loop function. A `begin-end` and `do-end` construct are respectively provided for sequential and independent parallel evaluation.

A number of declarative set operations that are coded as primitives form the backbone of FAD's query processing capabilities. These resemble algorithmic skeletons (§ 2.2.5). The main operator is the pump operator, `pump(unary-func, binary-func, identity, set_expr)`. This takes a unary operator abstraction which is for application to individual elements, a binary function that is associative and commutative and which is used to combine elements produced from the application together with the identity of the binary function. Although FAD constructors accept functions as arguments, user–defined functions in FAD may *neither* accept functions as arguments *nor* return functions as result. Another of the primitives is the `filter` operator that applies a filter to the Cartesian product of its argument sets. There is also an explicit nesting construct `group(f,S)`, which returns pairs representing the equivalence classes of the set `S` under the application of the function abstraction `f`.

Three techniques are used for horizontal data placement: key ranges, hashing based on key values and random placement. In order to reduce the performance impacts of delays due to message interchange synchronisation and data exchange, Bubba does not partition data across all processors. This is decision is intended to reduce the total number of message exchanges.

Volcano

Volcano, [Gra94], is a parallel dataflow query execution system for which a shared memory implementation exists. Data parallelism is supported through partitioned data, pipeline parallelism and process parallelism.[13] The `exchange` operator supports data parallelism on partitioned data–sets and both pipeline and process parallelism, translating between demand-driven dataflow within the process and data-driven dataflow between processes. In Volcano, parallelism is made explicit through the use of the `exchange` operator, which may be used to parallelise any operator.

Volcano's design does not presume any particular data model: the only assumption is that query processing is based on transforming sets of items using parameterised operators. Volcano provides mechanisms for query evaluation to allow selection and experimentation with policies. To achieve data-model independence, set processing control (which is provided by Volcano's own operators) is separated from interpretation and manipulation of

[13] Also known as task parallelism.

data items. The software architecture consists of two layers: the file system layer and the query processing layer, the latter of which is reviewed here.

The operators in Volcano's query algebra are processing algorithms. The tree-structured query execution plan is used to execute queries by demand-driven dataflow. Algebra operators are implemented as, possibly nested, iterators — empty algorithm shells that support an *open-next-close* protocol. A state record containing each iterator's arguments and state facilitates multiple iterator invocations. Manipulation and interpretation of data objects is passed to the iterators by function pointers. All iterators are implemented on a uniprocessor environment and are parallelised using the `exchange` operator.

Examples of iterators are `scans, functional joins` and `filter`. The `one–to–one match` and `one-to-many match` operators realise set-matching functions. These when used in conjunction with the `filter` operator implement all relational operators in which an item is included in the output depending on the result of a comparison between pairs of items. Examples include relational join, intersection, aggregation and duplicate elimination. An operator is also included for sorting. To incorporate a new algorithm for any operation, e.g. `nest` or `unnest` operations, the algorithm can be coded in the iterator paradigm.

In Volcano, all issues with respect to parallelism are localised in one operator – the `exchange` operator – which is inserted into the query execution plan. The `exchange` operator provides a standard interface to the operators above and below in a query tree. The exchange operator can be inserted into different parts of the tree, effectively giving independent parallelism. For example, to facilitate independent parallelisation of the two separate inputs of a merge join operation, an `exchange` operator is inserted between each of the inputs of the merge-join. Flat data parallelism can be exploited within scan operations by the insertion of an `exchange` operator above the scan in the query execution tree. The `exchange` operator in the consumer process behaves like a normal iterator, the only difference from other iterators being that it receives its input via inter-process communication rather than iterator procedure call.[14] Use of the `exchange` operator allows other operators to be implemented without a view to parallelism: issues such as partitioning, flow control are encapsulated in and provided by the `exchange` operator. Thus, data manipulation and parallelism are orthogonal.

Volcano is extensible in a number of ways. New data types may be added and parallelised using the `exchange` operator. In order to add new functions on individual objects or aggregate functions, for example, geometric mean, just the appropriate support function is required. To include a new query processing operation, for example, the `nest` operation, a suitable algorithm needs to be coded as an iterator with `open, next` and `close` procedures and must use these procedures for its input streams. Portability is achieved by re-implementing the `exchange` operator for a new parallel machine platform.

The next section explores query modelling techniques.

2.4 Query Modelling Techniques

Transforming programs and proving theorems about them can be made easier by requiring that functions be expressed in stereotyped ways. One kind of stereotyping is the systematic use of higher-order functions that traverse recursive structures. Such traversal functions

[14] Processes create and execute on both sides of the process boundaries.

can capture common patterns of recursion that occur often during programming and therefore minimise the explicit use of recursion. Expressing operations in terms of a combinator allows easier reasoning which aids transformations, and *may* ease the task of parallelising queries.

Higher-order generic operators, such as *map* and *filter*, are an effective way for capturing traversals over collection structures, as discussed in § 2.2.5. Each operator characterises the control structure of an algorithm, while the instantiations of its functional parameters characterise the policy that elaborates the algorithm. By abstracting the policy details from an algorithm it is possible to express laws about all the algorithms with the same control structure. These laws may be apparent when these algorithms are examined in their fully instantiated form. By restricting the number of operators to a small but powerful set of higher-order operators, the number of transformation rules needed for optimising expressions can be reduced.

Since this thesis aims to develop a unified model of parallelism for data–intensive applications, query modelling techniques can aid in the process of developing a model of parallelism for queries based on a formal foundation. Furthermore, such a basis for modelling queries enables optimisations that are used to be proved correct.

2.4.1 Sheard

Sheard introduces an algebra, termed SRL, in which the main operation is a low-level `set_reduce` operator [SS89], as defined by:

```
set_reduce(set1,app,acc_fun,base,extra_params)
```

where `app` is the function that is applied to all elements of the set `set1`. `extra_params` is the additional parameter(s) that `app` requires, `acc_fun` is the accumulating function, and `base` is the initial accumulator value. For example, the set intersection of sets `r` and `s` is computed by:

```
set_reduce(r,([x,y]→if member(x,y) then {x}
                                   else emptyset),union,emptyset,s)
```

In the SRL algebra all parameters to `set_reduce` are SRL expressions, which can be `set_reduce` calls, set constructions (for example, `emptyset` or `insert`[15]), tuple constructions, tuple selections, equalities, or operations over primitive types, such as booleans and integers.

The broad class of traversal functions called *traversal combinators* captures a large family of type-safe primitive recursive functions. Most functions expressed as recursive programs where only one parameter becomes smaller at each recursive call are members of this family. This restriction excludes the computation of some valid functions, such as structural equalities and ordering, because they require that their two input structures be traversed simultaneously. The generic programs are *combinators* as each takes a function as inputs and returns a new function as output (the one that performs the actual reduction). One

[15]Given an element, `insert` forms the singleton set.

contribution of the approach is the treatment of the class of traversal combinators resulting from restricting the input functions to be themselves traversal combinators, termed *uniform traversal combinators*. The algebra can capture most polynomial time functions. The traversal combinator for sets, is given by [Feg93]:

```
tc_set(f1:[]→ β,
       f2:[α,set(α),β]→ β):[set(α)]→ β)

[x] → if x=emptyset
         then f1()
         else f2(choose(x),rest(x),tc_set(f1,f2)(rest(x)))
```

where `x = {choose(x)} ∪ rest(x)`.

2.4.2 SVP

The goal of SVP [Par92] is to model data parallel and pipelined parallel tree-based evaluation on sets and streams. Collections are represented as ordered binary trees over which divide-and-conquer mappings are defined. SVP supports nested collections[16] of arbitrary nesting and internal structure.

It is observed that specifying parallel computations over relations[17] often requires specifying how data partitioning (fan-out) will be done and how results will be collected (fan-in parallelism). Four main forms of parallelism are identified: independent parallelism; pipeline parallelism; fan-out parallelism; and fan-in parallelism. The goal is to support a formal semantics for divide-and-conquer computations on sets and streams to express independent (set) and pipeline (stream) parallelism. Parallelism for divide–and–conquer is specified using both the structure of the data, and the structure of the divide-and-conquer mapping: dividing-parallelism is specified by the data, and conquering parallelism is specified by the mapping. Partitioning can always be used to modify the data structure and thus affect dividing-parallelism.

SVP requires that all collection mappings be SVP–transducers, as defined by:

$$\begin{array}{ll} f(s) & = F(Q_0, \rho(s)) \\ & \\ F(Q, \langle\rangle) & = id_\theta \\ F(Q, \langle x\rangle) & = h(Q, x) \\ F(Q, S_1 \diamond S_2) & = F(Q, \rho(S_1)) \\ & \quad \theta \;\; F(\delta(Q, S_1), \rho(S_2)) \end{array}$$

$\langle\rangle$ denotes the empty collection, $\langle x\rangle$ denotes a unit collection and $S_1 \diamond S_2$ is a collection if S_1 and S_2 are collections.

Here, Q_0 is an arbitrary fixed value, ρ is either the identity mapping or an SVP-transducer, and h, θ and δ are arbitrary SVP mappings of two arguments (θ is a binary operator). An SVP mapping is either a constructor (tuple, collection formation ($\diamond$)), or a deconstructor, which consist of type-membership predicates for SVP values (a value is an atom, tuple or collection). The functions f, F and h are permitted to take additional arguments. In particular, f can be a function of other parameters besides the collection S (including other collection parameters). The mapping $\rho(S)$ typically performs data partitioning on

[16] That are treated as a set or stream.

[17] In the context of relational database queries.

the collection S. For example, if ρ partitions a tree into two subtrees by comparing with a median-estimate key value, then it is possible to divide sort data. In addition, the Q argument can be omitted if it is not used by h or δ. When θ is a complete operator with a left identity id_θ, θ is termed a collector (indeed, with no identity, it is not possible to carry out parallel evaluation in some cases).

The SVP-transducer is flexible, for example, it is possible to implement merge scans on streams. The merging of two streams is accomplished by making one of the streams the initial state while simultaneously consuming the other stream. It is noted that other properties of collectors such as commutativity and idempotency can be exploited to obtain greater parallelism. To limit the power of the SVP-mappings some restrictions are proposed.

2.4.3 Comprehensions

The desire to increase the expressive power of database query languages together with the need to model non-relational data structures - especially those that are provided by object oriented databases - has led research into using *comprehensions*. List comprehensions were first proposed by Trinder [Tri91] as a single query notation that can be defined over several bulk data types, since they can be defined over any type that is a *ringad*, examples of which are sets, relations, bags, lists and some trees. A comprehension over a ringad bulk type is translated into invocations of the appropriate ringad functions. Since Trinder's comprehensions [Tri91] are based on the typed λ–calculus, it is possible to use both computation and recursion in queries involving them: comprehensions can invoke pure recursive functions as they share the same semantic basis. Given this, optimisations are also possible because the λ–calculus semantics permit proof of equivalence and hence, transformation between comprehension expressions. By providing a translation that maps any relational calculus query into an equivalent list comprehension, the relational completeness of list comprehensions (augmented with three auxiliary functions), is proved in [TW88].

Functional database queries have also been modelled using comprehensions [Pou88, TW89]. Database implementations using comprehensions include FDL [Pou89, Pou92] - a sequential implementation based on a triple-store - and Heytens in AGNA [Hey92], a fine-grain dataflow multiprocessor evaluating a form of parallel list comprehensions. Object-oriented database queries have also been modelled using comprehensions [CTW94, BLS$^+$94].

2.4.4 Structural Recursion

Structural recursion was introduced as an effective way to capture database queries by Tannen and Buneman [TBW92, TS91, BNTW95]. Two forms of structural recursion over collection types are discussed. *Element–wise* structural recursion, also known as *structural recursion over the insert presentation*, *SRI*, in the form of the $g = sr\_add(i, e)$, is defined by:

fun g(empty) = e
| g(add(x,C))= i(x,g(C))

where $add(x, C)$ is a collection. In order for g to be well defined, i has to be commutative and idempotent.

The other form of structural recursion, divide–and–conquer structural recursion, also known as *structural recursion over the union presentation*, *SRU*, is defined by h such that $h = sr\_comb(u, f, e)$:

fun h(empty) = e
| h(sng(x)) = f(x)
| h(comb(C_1,C_2)) = u(h(C_1),h(C_2))

where $sng(e)$ and *comb(C_1,C_2)* represent the singleton collection consisting of element e and the collection which can be sub-divided into two smaller collections (the divide-and-conquer recursion template). Operations of the *SRU* require the validation of the associativity, commutativity and idempotency properties of the function u. It is recognised that the task of deducing properties such as associativity, idempotency and commutativity is undecidable. In view of this, the cases in which these conditions are automatically satisfied are considered. One particular case of this is when u is the same as *comb* - for sets, bags and lists, this case is termed as *ext(f)*. This maps the function f over each member of a collection and then "flattens" the result.

A first-order expressible translation from *SRU* to *SRI* is provided in [TS91]:

$$sru(u, f, e)(O) \stackrel{def}{=} sri(\lambda(x, y).u(f(x), y), e)(O)$$

where *sri* and *sru* denote the functions *sr_add* and *src_comb* respectively. The opposite translation from *SRI* to *SRU*, however, uses higher-order functions. Furthermore, Suciu and Wong [SW95] show that any uniform translation, where a uniform translation has to translate or compile $sri(i, e)$ by independently translating or compiling *sri*, i and e, must map some PTIME algorithms into exponential space ones (due to the creation of large intermediate data structures). The efficient translation of *sri* into *sru* using higher-order functions together with the proven inefficiency of any uniform translation of *sri* into *sru* suggests that higher-order functions could be useful in query languages. Any other translation will create large intermediate data-structures and is hence inefficient.

SRI is naturally associated with sequential processing of collections: the elements of some collection s are processed one by one. In contrast, *SRU* is naturally associated with parallel processing of collections, in a divide and conquer manner: to compute $f(s)$, one divides s into two components and computes $f(s_1)$ and $f(s_2)$ in parallel and independently, composing the two results.

CPL

The collection programming language (CPL), [BNTW95, DHP97] is a language mainly intended to allow integrated access to data in heterogeneous data sources in which pattern matching is used. CPL allows arbitrary nesting of the collection types set, bag and list together with record and variant types. The monad comprehension syntax is adopted in [BLS+94] as an alternative syntax to *SRU/SRI*. The approach uses monad comprehensions to capture operations that involve collections of the same type while structural recursion is used for expressing the rest of the operations (such as converting one collection type to another, and aggregation). CPL incorporates variant types, function declarations and pattern matching.

For example, the following CPL query over a students database, DB, nests the students by courses taken:

$$\{[course = c, students = \{x.id|\backslash x < -DB,\\ [number = c, ...] < - - - x.courses\}]|\backslash y < -DB, [number = \backslash c, ...] < - - - y.courses\}$$

where each student takes a number of courses *y.courses* and where $< - - -$ is a list generator for courses. When there is a generator inside a comprehension of a different type, the collection on which the generator iterates is converted to the type of the comprehension. In this case, *courses* is converted to a set before the iteration.

Suciu

Suciu investigates the implementation of collection languages in [Suc96, Suc95]. The potential of nested parallelism within object oriented queries is exploited using (sequence-based) flattening techniques based on segment descriptors [BS90]. Although the technique ensures load balancing, this is at the cost of flattening nested collections, distributing, re-distributing partial results, and performing co-ordination between processing nodes. An optimal algorithm is given for performing the necessary communication.[18] The model is based on parallel complexities (which is based on *parallel time complexity* and the *work complexity* (§ 2.2.5 and [Ble95, BG96])), and on the communication costs (based on the *logP* [CKP+92] model). The data model is based on the *nested collection calculus* (NCC) which operates on sets, bags and sequences. The central operation is the *parallel map*, which is of the form $map_{ct}(f)$ *collection*[19] (cf. *SRU*). When a function being mapped *(f)* is itself another *parallel map*, this gives the potential for nested parallelism. NCC also includes a number of operators, such as *union*, *flatten*, in addition to some other primitive arithmetic operations and conditionals. For sequences, in particular, the functions *enumerate*, *zip* and *split* (split is inverse to flatten) are included to allow efficient implementation of sets and bags in terms of sequences.

2.4.5 Fegaras

The inductive case in the definition of *SRU* in § 2.4.4 is:

```
h(comb(C1,C2) = u(h(C1),h(C2))
```

In order to have deterministic evaluations, Buneman *et al.* require that, for example, in the case where the base data type is set, that e and u form a monoid commutative-idempotent monoid, $\mathcal{M}(set, e, u)$. In order to guarantee that *SRU* is well-defined in all cases, a restriction is made. The restriction is that the function *comb* is the same as u, i.e. that the result monoid type is the same as the argument monoid type. Aggregate functions are evaluated by the application of the appropriate functions.

[18] In the sequence algebra.

[19] Where *ct* denotes *collection type.*

Instead of making this restriction, Fegaras [FM95a, FM95b] observes that the condition of *well-formedness* requires that if the argument monoid type is a commutative (-idempotent) monoid, then the result type must also be a commutative(-idempotent) monoid. In addition, Fegaras defines aggregate functions, such as *sum*, as monoids, e.g. $sum(int, 0, +)$.

Using the *monoid comprehensions* calculus, queries that simultaneously deal with multiple collection types can be expressed, subject to the same well-definedness conditions as in structural recursion (see § 2.4.4). For example, the cardinality of a bag xs, may be expressed as: $sum\{x|x \leftarrow xs\}$: both *bag* and *sum* are commutative monoids.

Monoid comprehensions are defined in terms of monoid homomorphisms, which have the same form as *SRU*. The monoid homomorphism is the only means of bulk manipulation of collection types supported in the underlying algebra. A simple normalisation algorithm is given for achieving canonical forms of monoid algebra expressions.

Monoid comprehensions are more expressive than monad comprehensions. This is a consequence of the fact that a monoid homomorphism is a generalisation of `flattenmap`.[20] In contrast to `flattenmap`, a monoid homomorphism can map one type into another type. If one generator in a monoid comprehension is over a commutative and/or idempotent monoid then the comprehensions must be over a commutative and/or idempotent monoid too. This condition can be checked by a type inference system and report errors to the user, since the commutativity and idempotency properties of a monoid are specified explicitly when the monoid is defined.

2.4.6 CROQUE

The CROQUE project (Cost and Rule-based Optimisation of object-based Queries), [RS96, GKG+97] aims to design and implement an object-based system which "realises a separation between logical and physical structure" (*the systems architecture is presented in [HK96]*). CROQUE attempts to define the formal semantics of the Object Definition Language (ODL), given in [Cat96]. The model is modified in two main ways: first, in CROQUE, mutable objects are atomic; second, objects can be instances of multiple types. The type of an object may be changed by means of gain and lose operations.

In CROQUE, queries are expressed in CROQUE-OQL/OML - a variant of ODMG-OQL/-OML (Object Manipulation Language). Attention is paid to exploiting physical designs which can differ significantly from the logical schema's view of data: partitioning, clustering and replication of database objects. The CROQUE data model is a development of that of COCOON [SLR+94]. It handles values (immutable objects) and orthogonal type constructors (tuples, sets, bags, lists and arrays). A general typing theory for mutable and immutable objects is presented as the basis for a closed query language.

Queries expressed in OQL [Cat96] are translated into a hybrid-algebra and calculus-based notation based on monoid comprehensions to allow a uniform optimisation framework for OQL queries. Within the model, an OQL query is translated into a representation which is a mix of the monoid calculus with additional algebra operators. Common algebraic rewrite rules are carried out on the calculus terms before the calculus terms are translated into applications of the *fold* function. The efficiency of mapping a language based on the monoid calculus to database query engines is considered in [GKG+97].

[20]`Flattenmap` is equivalent to the *sru* form.

Since the calculus expressions inherently describe nested loop processing strategies, a case is made for including other algebraic operators such as joins (especially the semi-join), selections, simple projects, grouping and nesting. Hence, a hybrid algebra/calculus approach is advocated in order to incorporate query execution plans beyond nested loop processing. The treatment is broadly similar to [Feg94], except that the CROQUE algebra incorporates additional operators such as *join*, and *nestjoin* (a join followed by nesting operation).

2.4.7 Comparison of Query Modelling Techniques

The most general of the stereotypical forms considered is the *SVP–transducer* [Par92] (§2.4.2), even though the only data types dealt with are set and stream. The *SVP–transducer* generalises map, parallel divide–and–conquer and parallel stream processing. The definition of the transducer allows the provision of a "seed", which is used at the singleton set/sequence processing stage. However, this seed may be changed throughout the evaluation, which in a parallel implementation will necessitate inter-processor communications.

An early work in this area is Beeri's [BK90], which features two operators `pump` and `apply-to-all`. The `pump` operator requires a unary and a binary function for element application and partial result combination, respectively. A number of optimisations are given for expressions involving these forms. However, no precise rules are presented for the application of these optimisations.

Beeri's *pump* operator is similar to its namesake in FAD [BBKV87, Ban92] (§2.3.3), a functional language intended for the Bubba parallel database machine. FAD features nested sets with object identifiers and two main parallel constructs: a parallel map (known as `filter`), and the parallel divide and conquer `pump` construct. A similar construct is Machiavelli's *hom* [OBT92]:

$$hom(u, f, e)(O_1 \cup O_2) = u(hom(u, f, e)(O_1), hom(u, f, e)(O_2))$$

where e is a zero element, f is the function which is applied to each element, u is the function that is used to merge results, and O_1 and O_2 are disjoint sets. The *hom* operator is based on *SRU* (*structural recursion over union presentation*), as discussed in §2.4.4. Proper applications of *hom*, which have the property of being computable in parallel, are defined to be when u is an associative, commutative operation and f has no side-effects.

The disjointness property of *hom* for the sets O_1, O_2 is in contrast to the *idempotency* requirement on function u of *SRU*. Due to the difficulty of deducing properties of the merging function u, the approach taken in *SRU* (§2.4.4) is to restrict the class of functions which can be handled. This restriction essentially turns *SRU* into a *map*-like operation, which precludes *reduction*-like functions. Built-in functions are incorporated for performing specific reductions, e.g. *sum*. More complex aggregate operations such as nesting can be coded in CPL §2.4.4, although the direct implementation of the comprehension expression does not yield efficient algorithms in many cases.

Fegaras §2.4.5 observes that the requirements on the merge function u in *SRU*, restrict the data types that are dealt with to *monoids*. Fegaras uses monoid comprehensions as a high-level language but also defines some of the reduction operations as specific monoids. This approach allows many data processing operations to be handled within the framework of monoid comprehensions. The underlying evaluation mechanism for the monoid comprehensions is the monoid homomorphism (cf. *SRU*). *CROQUE* §2.4.6 also uses monoid comprehensions except that additional operators are used for operations such as joins.

2.5 Summary

This chapter has presented a non–exhaustive survey of a number of key areas which are related to models for application level parallelism for data–intensive applications. The areas which were reviewed include models for parallelism, parallel database management systems and database query modelling techniques. Each of the abstract models of parallelism offers specific functionality which the application area requires. The low–level task parallel models, for example, provide more flexibility for building applications compared to the higher–level declarative approaches. However, the declarative approaches offer easier application development and greater flexibility especially when compared to parallel database management systems in the commercial domain. The current support for application–wide parallelism from parallel database management systems seems to be extremely weak, although Bubba's FAD language incorporates facilities for defining task parallelism within the query language. It is interesting to note that the theme of using a set of functional forms not only appears in the work on functional programming approaches, as one would expect, but also appears in a number of database query modelling techniques.

The next chapter shapes a set of requirements for achieving application–level parallelism in data–intensive applications out of the related work presented in this chapter. The related work is also evaluated with respect to these criteria, the promising approaches being highlighted.

Chapter 3

Requirements and Analysis

In chapter 2 a non–exhaustive survey of research in three areas was performed: general models for parallelism (§ 2.2), parallel database systems (§ 2.3.2 and § 2.3.3), and query modelling (§ 2.4). In this chapter, the applicability of work within each area is evaluated against the set of criteria given in this chapter (§ 3.1). Each section reviews the work and discusses the approach used in this thesis.

3.1 Requirements for a Model of Parallelism for Data-intensive Applications

An abstract model of parallel computation is an interface separating high-level problem-specific properties from low-level language and hardware issues. The model furnishes an abstract machine which provides certain operations to the programming level and requires implementations for each of these operations on all classes of parallel architectures. Some models are aimed at very specific application-areas or fit best with particular classes of parallel architecture making them unsuitable for investigation as general models.

The required properties for a suitable model are:

- *application-level parallelism* – the model must support parallelism across the query and non-query parts of a data–intensive application.
- *ease of programming* – the model should provide a high-level interface through which a programmer may construct applications with little concern for the problems of partitioning and mapping, communications and synchronisation.
- *architecture independence* – the model should be independent of any particular class of parallel architecture.
- *extensibility* – the model should allow the addition of new data types, such that operations over new data types are absorbed into the parallel evaluation automatically.
- *correctness* – the correctness of applications developed using abstract models that are based on theoretical principles is easier to establish compared to applications that have been developed using ad–hoc techniques. Further, a formal basis may allow the application of meaning–preserving rewrite rules.

- *efficiency & cost* – the model should be implementable over a range of parallel architectures. Further, it should be possible to determine cost measures from the application code.

These criteria are now applied to the work presented in chapter 2. Note that the approach that is advocated in this thesis with respect to each requirement is presented at the end of each section.

3.2 Application-level Parallelism

3.2.1 Motivation

Parallel database systems based on the relational model are now widespread in the commercial sector, with a product from each of the major database vendors. Such commercial parallel DBMSs have performed well in their primary roles of online transactions processing and decision support. However, current techniques of building database applications on parallel machines only exploit parallelism across the database component of an application.

In parallel database systems, the most readily utilisable source of parallelism, data parallelism, is realised through the concurrent execution of operations such as table scans and filter application on data distributed amongst a number of processors. Independent parallelism (a form of task parallelism), also known as *bushy tree* parallelism, is utilised when a query evaluation tree is partitioned such that evaluations on different partitions proceed on different processors. Pipeline parallelism (the other form of task parallelism), is also exploited between data-parallel and independent parallel parts.

Applications using an SQL-based parallel relational database system are usually written in an imperative language such as the C language, or more recently, Java, with database calls being handled through an API provided by the database system. The execution model is client-server based: the main application runs sequentially on a front-end uniprocessor machine whilst parallelism is exploited for the evaluation of the database queries on a (parallel) back-end database server.

It is not unreasonable to assume that today's high-end desktop workstations will be replaced by desktop multi-processor machines, such as the Intel-based SMPs which are now available. The inter-operability between a front-end application running on such a machine and a parallel database system running on a massively parallel architecture gives new opportunities for exploiting application level parallelism. The way in which this power can be utilised, however, is a somewhat unexplored area. Consider the following scenario:

> Languages such as Java which provide threads at the language level potentially facilitate more responsive programs by ameliorating the cost of expensive operations such as waiting for the completion of an I/O call, or returning database query results. Implementations of the Java virtual machine on operating systems which support threads (such as Solaris 2.6+) are now multi-threaded. Such multi-threading means that it is possible for true parallelism to be exploited across the whole of an application, provided the operating system hardware platform is a multi-processor machine. A number of threads can each be

assigned to different database queries, possibly against databases on physically separated machines. This gives true parallelism across the application as a whole.

Such a scenario potentially allows the exploitation of parallelism within both the database and non-database components of an application. However, this requires the programmer to identify and re–code program segments that may be evaluated independently, and possibly deal with synchronisation. Although parallelism is utilised within both the database and non-database parts of the application, there is still a distinction between them: there are still two programming models to contend with. Further, as noted by Mohan *et al* [MPTW94], there is a case for pushing more of the application functions into the lower layers of the DBMS, either in the from of operations on abstract data types, function libraries or method calls. The motivation for this is mainly to reduce the overall path length of transactions, thus allowing the front-end to issue a single call to the back-end to execute a sequence of SQL statements. Furthermore, some researchers [GJP97], are already investigating ways in which applications can interact with the database system *in parallel.* The proposed schemes attempt to eliminate the potential bottleneck that arises when applications can only interact with the database via a single thread. This is done by allowing the parallel subtasks of the database system *and* the application to communicate directly. A natural development of this is to more closely integrate the DBMS query language with the application language, as seen with the ODMG's Object Query Language Standard [Cat96]. However, the tighter coupling in the ODMG's standard does not readily facilitate greater parallelism in the non-database components of the application.

In today's parallel object-oriented databases, the main source of scalable parallelism is the set operations inherited from relational algebra. However, object-oriented databases also have the potential for nested parallelism which is associated with nested collections. A model of parallelism for data-intensive applications should be able to utilise nested parallelism in addition to data-parallelism and other forms of task parallelism.

Whilst it is important for a unified model to be expressive enough to allow the stipulation of queries in the query languages of common use, it is also important to concentrate on the model for *parallelism* rather than on a particular (database) data model. In addition, the goal of the unified model is not to squeeze the last ounce of performance, but to derive a flexible model at a high–level of abstraction, which can provide a framework for taking advantage of the potential of application-wide parallelism.

3.2.2 Treatment in Related Work

In discussing related work in the area of models for parallelism, two main categorisations were made (§2.2): implicit and explicit models. Implicit parallel models are easier to program since they provide an abstraction from the details associated with more explicit models: namely partitioning the problem into parallel parts, granularity definition, synchronisation and communication between parallel components.

Examples of implicit models which exploit task parallelism include models such as:

- graph-reduction (§2.2.5) and the automatic mapping of list processing functions to stereotypical forms;[1]

[1] Both in functional programs (§2.2.5).

- NESL(§2.2.5), which primarily makes use of data parallelism;
- the algorithmic skeletons (§2.2.5) approach in which both data and task parallelism are utilised.

Explicit parallel models require that the programmer defines at least one of the following tasks: problem partitioning, granularity selection, synchronisation and communication between parallel tasks. Explicit models of parallelism that comprise task parallel models (§2.2.2), data parallel models (§2.2.3), and hybrid task and data parallel models (§2.2.4) are now evaluated with respect to the criteria present in § 3.1.

Explicit Models

Task parallel models (§2.2.2) which use message passing usually provide abstractions of parallel hardware, such as standardised message passing primitives built using primitive ones and a virtual processor topology which is mapped over the real machine topology, and hence provide an imperative model of programming. Furthermore, using such an approach means that exploiting data parallelism, to which data processing applications are well suited, has to be mapped by hand. In order to reduce the need for duplicated code, the operations which all data processing applications perform in common could be abstracted in the form of library function calls.

Nevertheless, in explicitly task parallel models, the programmer is in charge of partitioning the problem, making decisions on the granularity of exploitation of parallelism, synchronisation of parallel tasks and communications. Although this offers flexibility and potentially optimal performance, especially for irregular and dynamic application areas, the realisation of these takes significant effort and requires the implementation to be *tuned* to a particular parallel machine. Given that today's relational parallel database systems offer parallelism through a declarative language (SQL), the option of condemning the application programmer to express application-level parallelism through the low–level means of message passing is not acceptable.

In task parallel models that use distributed objects (§2.2.2), the programmer makes partitioning and granularity selection decisions, whilst the compiler and runtime system manage communications and synchronisation. The main problem with such a model is the lack of support for mass data parallelism.

Purely data parallel models (§2.2.3) are well suited to applications in which the same action is performed on many different data items — such as in scientific and numeric applications which involve matrix manipulation — and parallel database systems. A data parallel language provides constructs for expressing that a statement sequence is to be executed in parallel on different data (or if data parallelism is assumed by default, in sequence). Implementations of data parallel models have been very much honed towards their intended application areas: for numeric/scientific applications, these have taken the form of extensions to procedural languages such as `C` and `Fortran`; for database applications, these have taken the form of declarative SQL embedded in a procedural language, such as C.

In approaches based on explicit data parallelism the programmer is required to stipulate data parallel (or non-data parallel) parts. In languages such as `HPF` (§2.2.3) data parallelism is exploited by loop-flattening. The programmer is required to make granularity decisions. Procedural data parallel languages are flexible and well suited to irregular applications. However, these approaches have support for a limited range of data types, vectors in the case of HPF, bags in the case of parallel SETL [HK93], and sets in the case of *parallel sets* [Kil92]. Support for multiple collection types would have to be implemented as required, although only on a one-off basis.

In database applications which use relational database servers, data parallelism is utilised in the query component of the application, this is even the case for some parallel object-oriented database systems, such as SHORE [Cea94] and OMEGA [GCKL93]. The application programmer is not required to make granularity decisions although the systems' administrator does usually stipulate data partitioning schemes.

In both approaches, the compiler performs scheduling and organises communications in the parallel components. The correctness of applications using the data parallel models is easier to ascertain than for task parallel models, although this is still not easy, especially for procedural languages. The natural mapping of data parallel operations to architectures, at least for simple types, makes efficient implementation possible.

However, task parallelism is difficult, if not impossible, to express in implementations of the the data parallel models based on procedural languages. Whilst in parallel implementations of relational database systems data and task parallelism are exploited in the query component, no task or data parallelism is exploited within the *non–query* parts of data processing applications. Whichever approach is taken, be it that of using an explicitly data parallel model or embedded SQL, the goal of application level parallelism (§3.2) cannot be realised using any of the solely data parallel models.

Hybrid data parallel and task parallel models offer the opportunity to exploit both data parallelism, for performing the same operations over different data in parallel, and task parallelism in performing concurrent evaluation of simple and complex data parallel tasks. However, implementations of hybrid task and data parallel models (§ 2.2.4) generally require the insertion of explicit constructs in the application. These constructs are used by the computer to determine the parts of the application from which it is possible to extract task or data parallelism.

Hybrid approaches that integrate both task and data parallelism lack a formal model for dealing with operations over multiple collection types. In addition, only one of the data parallel implementations reviewed incorporates direct support for nested data parallelism (Braid [WG95]), although nested parallelism may be implemented as required by the application programmer with the others. Perhaps most important of all, the question is whether more declarative approaches can provide the required flexibility for data processing applications, whilst reducing the burden on the application programmer of partitioning the problem, mapping the problem and handling synchronisation and communication issues.

Implicit Models

Implicit models considered in this section are those based on functional languages and homomorphic and algorithmic skeletons.

In NESL [Ble95, BG96] data parallelism is exploited through the use of the data parallel *map-like* `apply-to-all` function and built-in function, such as `sum`. Pseudo task parallelism is also exploited in the form of nested-parallelism (also known as *subset level* *parallelism*), which naturally occurs in object-oriented database systems due to the presence of nested structures. NESL incorporates a PRAM-based [FW78] cost model, which uses the *work* and *depth* complexities of a program to make granularity decisions. Related work is that of Suciu [Suc96] who derives and implements a number of algorithms that ensure efficient nested–parallelism within the context of collection-oriented languages.

Another implicit data parallel model is that of homomorphic skeletons (§2.2.5) which is a derivation of the Bird-Meerterns Formalism (BMF). Operations on lists are structured as the composition of a `map` function followed by a `reduce` function. Hence, all homomorphisms on a data type can be expressed as an instance of a single recursive highly-parallel computation pattern, so that the arrangement of computational steps in an implementation needs only to be done once for each data type. The template for scheduling the individual computations and communications can be reused to compute many different homomorphisms by replacing the operations that are done as part of the template, where the operational semantics of the template, and the amount of parallelism that can be exploited depend on the properties of the data type. Although the homomorphic skeletons approach models data parallel operations over lists and other collection types, task parallelism is not incorporated. However, a similar approach, namely the algorithmic skeletons models (§2.2.5), in addition to the *reduce* and *map* forms for data parallelism, also incorporates task parallel skeletons, in the form of process pipelines and process farms. Skeletons raise the level of abstraction since they are fundamental units with which the programmer must work. Although skeletons are internally parallel, they are composable sequentially so that programmers do not need to be aware that they are programming in parallel.

In the parallel database systems, the embedded SQL approach to constructing data processing applications allows task parallelism to be exploited only in the concurrent evaluation of independent data parallel parts. If the host operating system supports threading, then the application may also experience parallelism in the independent execution of different database queries using operating system threads.

3.2.3 Thesis Approach

The approach to achieve integrated parallelism across the whole application is based on utilising *algorithmic skeletons*. The skeletons used are: a *pipe* skeleton for modelling pipeline parallelism, a *farm* skeleton for independent parallelism, *reduce* skeleton for data parallelism, and a *split* skeleton for dividing tree parallelism. The motivation for this choice follows.

Some approaches separate data parallelism from task parallelism whilst still supporting pipeline parallelism within and across data parallel components and also between data parallel and task parallel components (where task parallelism is supported). The approaches that use data parallelism make extensive use of a single function to iterate over elements of a collection. This approach is used for evaluating both filter and reduction operations. Proponents of this approach include FAD's higher-order (§2.3.3) `pump` operator and SHORE's `set-apply` operator in database work; NESL's `apply-to-each` and `map` and `reduce` skeletons (§2.2.5) in implicit approaches loosely based on functional programming; and *SVP* (§2.4.2) and monoid homomorphism (§2.4.4 and §2.4.5) in query modelling work.

Another approach is to envelope all forms of parallelism into a single operator. The `exchange` operator in Volcano (§2.3.3), which is explicitly inserted into query execution plans, supports data parallelism (on partitioned sets) and task parallelism (independent parallelism) between portions of the query execution plan where an `exchange` operator appears. Pipeline parallelism is also exploited in the merging of partial results from data parallel operations. Since Volcano implements dynamic query execution plans, the precise form of parallelism that is exploited is decided at runtime. In order to deriver a parallel implementation for a new parallel architecture, only the exchange operator requires re–implementation. The use of a single structure to model all forms of parallelism means that integrating primitives for non–query parallelism into the structure will involve a certain amount of redundancy (duplicated control structures). In addition, the amount of parallelism that is extracted from programs does not completely depend on runtime factors. For example, it is possible to estimate the size of partial results in data–intensive applications, provided that statistics on the selectivities of predicates are maintained. This enables decisions on the runtime forms of parallelism to be made at compile–time. Hence, enclosing all three forms of parallelism within a single operator is not considered to ease the task of deriving a model for application-wide parallelism.

Hence approaches in which each form of parallelism has been explicitly identified and treated in a different way are now considered. In the explicit approaches the programmer is required to partition a problem into parallel components, decide which components are actually run in parallel at runtime, and organise synchronisation and communications between the parallel components. On the other hand, declarative approaches, such as graph reduction machines for functional programs, which have attempted to distance the programmer from all of these issues, have met with limited success on non shared–memory machines. This is mainly due to problems associated with ascertaining the optimal grain of parallel execution.

The automatic exploitation of machine dependent features is far more complex in the parallel world that in the sequential case. Most of the problems to be solved are intractable and the development of effective heuristics is bounded by the interwoven issues that influence the overall performance. Suitable algorithms can be found only if the control and interaction pattern of the parallel program and the topology of the architecture are constrained. Most of the models of parallelism and that languages that are derived from them are too powerful in this sense. They allow the expression of parallel computations with an unrestricted control and interaction pattern. This universality makes the problems too complex to be solved effectively by the tools. Further, the programmer cannot predict the performance of the program unless resource allocation is dealt with explicitly.

From this, it became increasingly recognised that the only way to address high–level massive parallel programming is to limit the structure of the parallel computations that can be expressed within the chosen programming model [ST98]. In this vein, the approach of using *skeletons* requires the programmer to design and structure applications using a *restricted* set standard forms of computation. Some of these forms, such as ones that apply a `map` and or `reduce` operation over the elements of collection, appear as iterators in many (sequential) collection–oriented languages. However, other forms such as the independent farm or pipe skeletons do not. Nonetheless, designing and coding applications in terms of standard constructs reduces the task of the compiler and runtime system to that of determining the right granularity of parallelism. This leaves granularity selection, a purely dynamic factor, to the runtime implementation. After all, since some runtime factors contribute to the difficulty of this task, it is a job which the programmer would find difficult. This is reinforced by the fact that in the explicit approaches to parallelism the granularity of parallelism is fixed by the programmer at compile time.

The skeletons approach incorporates two additional features: *skeleton composition*, which allows application to be built by hierarchical composition of a fixed set of skeletons; and *rewrite rules* which allow optimisations to be carried out on groups of composed skeleton expressions. The possibility of applying skeleton rewrite rules at compile–time means that efficient implementations may be realised even from badly stipulated forms. Furthermore, portability is gained through the use of machine–dependent compilation rules.

The skeletons approach is adopted in this thesis and this decision is explained in this section.

Data Parallelism

For the requirements of data processing applications which make extensive use of data parallelism in the form of iterations over partitioned data (collections), a combined *reduce* and *map* skeleton, such as the `DC` skeleton (§ 2.2.5), is required. The `DC` skeleton more closely matches the virtual process topology of data parallel operations, since it has a process graph which is determined by the function which is used to combine partial results. Furthermore, it has also appeared as the *SVP-transducer* (§2.4.2) in the work on query modelling, reinforcing its suitability for parallelism within data-intensive tasks. Consider the definition of the `DC` skeleton (§2.2.5) given below:

```
DC t s d c x = if t(x) then s(x)
               else c(map (DC t s d c) (d(x))
```

The two tasks of partitioning the data set `x` and that of evaluating a *map* followed by a *reduce* function have been integrated into a single skeleton (this is also the case with the *SVP-transducer* (§2.4.2)). Since for some data processing tasks, data is available ready-partitioned at a group of nodes within a parallel machine, the data distribution and function evaluation components of this skeleton should be separated. This approach has been taken in P^{3}L (§2.2.5) in which both the `reduce` and `map` skeletons operate on ready-partitioned streams of data. This means that a separate skeleton that partitions data, namely a dividing tree, should also be incorporated.

Considering the function evaluation component only, implementations which include the `DC` skeleton do not make use of any properties pertaining to the functions which are used as arguments to this higher-order function.

In fact, properties on the function `c`, such as associativity, commutativity, idempotency and existence of a unit element for `c` may expose opportunities for increased parallelism and optimisations. This includes the use of runtime resource information to adapt the form of runtime virtual process(or) topology. The branching factor, which is determined by the function `d` in the above definition, gives a fixed virtual process(or) topology. Varying this branching factor, where algebraic properties allow, facilitates a more flexible virtual process(or) topology.

The use of a combined *map* and *reduce* has long been advocated, first in the *Bird-Meerterns Formalism* (BMF) (§2.2.5), and more recently in some query modelling techniques (§2.4), albeit in a restricted form as a monoid homomorphism (§2.4.4, §2.4.5) and §2.4.6). Hence, a combined `map` and `reduce` skeleton, and a separate partitioning skeleton are adopted.

Task Parallelism

Task parallelism appears in two forms: pipeline parallelism and independent parallelism. For both forms of parallelism appropriate explicit operators are used: `pipe` for pipeline parallelism, and `farm` for independent parallelism. A core feature of the model is skeleton composition. This is the feature that enables the parallelism within applications to be structured in a hierarchical way. In addition, it allows the compiler to deduce the parallel behaviour of a program whilst at the same time providing the programmer with a way of re–structuring *existing* application code instead of rewriting code. The use of a procedural language does not make the task of the compiler any more difficult since the compiler uses a cost measure based on the *parallel* behaviour of programs when performing rewrites of a skeleton program. The parallel behaviour of a program is defined by its skeleton structure.

The skeletons model prescribes that skeletons communicate data in a stream–based fashion. Since the skeletons "interchange" data using a pre–defined interface, skeletons may be added to the system as required, as long as each addition conforms to this interface. This requires an implementation template, performance models, and (some) rewrite rules for the new skeleton. Furthermore, the model allows optimisations based on the parallel behaviour of programs to be performed across all of the application.

The skeleton approach has applicability even in application domains of numerical applications. For example, the behaviour of a number of parallel matrix multiplication algorithms may be easily expressed using the model. Currently, such tasks are typically carried out using a data parallel language such as HPF which utilises *loop flattening* to parallelise independent iterations of a loop, and thereby achieve a parallel implementation for matrix multiplication.

3.3 Ease of Programming

3.3.1 Motivation

A model for parallel programming of data–intensive applications must hide much of the detail concerning parallel evaluation from the programmer if it is to compete with the current embedded SQL approach. The complex issues relating to the execution of the parallel computation should be implicit and capable of being inferred from the program. This implies that models for parallel programming require high levels of abstraction. The detailed execution of a program on one architecture is likely to be different from that on another. Hence, abstractions are required that conceal the differences between different classes of architectures. In particular, the model should conceal: the division of programs into parallel chunks; the mapping of parallel tasks to processors; the setting up and management of communication channels between processors; and synchronisation steps. It seems impossible to expect the programmer to handle the task of scheduling, communication, and synchronisation of thousands of parallel threads. Burdening the programmer with these complex tasks is not only error-prone, but results in embedding the application semantics within implementation, leading to costly maintenance. However, ease of programming should not be facilitated at the expense of major inefficiencies.

For data–intensive applications in particular, a uniform interface to both query and non–query components should be provided to the programmer. In addition, the model should provide a high-level declarative interface such as SQL. The query notation should also have built-in redundancy to allow optimising rewrite rules to be carried out. Furthermore, the complexity of the application programmer's world is reduced if a single query notation can be defined over several or all bulk types.

3.3.2 Treatment in Related Work

General models of parallelism are considered first. Models that shield the programmer from the chores associated with partitioning a problem into parallel components and managing synchronisation and communication include those based on functional paradigms (e.g. NESL §2.2.5 and skeletons approach §2.2.5), and dataflow languages (e.g. §2.2.6). NESL (§2.2.5) for example, is a functional language[2] in which parallelism is exploited through built-in functions and the single *apply-to-all* function. Although implicit data parallel languages such as NESL have the potential for expressing data parallel operations on collections in a declarative way, extracting task parallelism from them in a general way is a more difficult task. In approaches which use algorithmic skeletons (§2.2.5), the programmer is provided with a set of parallel forms each of which is designed to capture a particular parallel form of computation. Typical skeletons include *map* and *reduce* for data parallelism, and *pipe* and *farm* for pipeline and task parallelism respectively. Research (§2.2.5) has shown that building parallel programs using the hierarchical composition of skeletons is an abstract and powerful technique, whilst at the same time facilitating advantages such as compile-time optimisations, tuning for different parallel machines, and portability. Although the programmer *is* aware of parallelism when using skeletons, concerns such as the explicit partitioning of problems into parallel components of a suitable granularity, synchronisation and communication are all handled by the system.

Considering the query component of applications, current database applications which use parallel relational databases interact with the database using the declarative language SQL. This declarative interface (and the machine independence which it results in), has been one of the contributory factors to the proliferation of parallel relational database systems. Current object-oriented database systems are based on the declarative language OQL (*Object Query Language*) [Cat96], and support nested collections and operations over multiple collection types amongst other object-oriented features such as object identity, inheritance and encapsulation.

In an attempt to model queries over non-nested and nested collections, comprehensions have been advocated as a declarative way of expressing queries in functional, relational and object-oriented database systems [Tri91, Pou88, Hey92, BLS$^+$94, BNTW95]. The approach taken by Buneman *et al* [BNTW95] is to incorporate monad comprehensions into a core language as an alternative to explicit structural recursion (§2.4.4). Since the monad comprehensions syntax does not express operations over multiple collection types, these are realised using explicit functions. Furthermore, aggregate functions are realised through built-in functions.

A syntax based on monoid comprehensions that facilitates operations across multiple collection types and the evaluation of functions such as summing and aggregation, is Fegaras' [FM95a, FM95b] approach.

---

[2] That does not allow higher-order functions.

3.3.3 Thesis Approach

The approach to achieve integrated parallelism across the whole application is based on utilising *algorithmic skeletons*. The skeletons used are: a *pipe* skeleton for modelling pipeline parallelism, a *farm* skeleton for independent parallelism, *reduce* skeleton for data parallelism, and a *split* skeleton for dividing tree parallelism. This choice is based on the requirement to model parallelism in data–intensive applications that are currently served by commercial database technology. Note that some applications may require additional skeletons to model, in the most natural way, all available parallelism.

Although using skeletons to restructure procedural application code is suitable for the non–query part of data–intensive applications, programming using a data parallel skeleton for the query component is not suitable for applications that currently code query components in SQL. This problem can be overcome by providing the *monoid homomorphism* as the basis of data parallel computation. The *monoid comprehension*, which has a natural mapping to the monoid homomorphism, may be used as a notation for stipulating queries over multiple collection types. This delivers the advantages of a declarative notation for queries which also has a data parallel implementation.

The use of comprehensions gives a declarative notation, similar to SQL and OQL, whilst at the same time providing a high-level interface for monoid homomorphism expression, through a simple translation from comprehension expressions to monoid homomorphism expressions. In addition, task parallelism may be incorporated into queries through the use of the the task–parallel *farm* skeleton.

3.4 Architecture Independence

3.4.1 Motivation

Computer architectures have comparatively short life spans, due to the speed with which processor and interconnection technology are developing. Users of parallel computers must be prepared to see their computers replaced, perhaps every five years. In addition to providing more features, each new computer will provides existing features in a different way compared to its predecessors. Redeveloping software from scratch each time this happens is not cost-effective, although this is what happens today. Hence, many users are forced to stay with one manufacturer of parallel computers since the low–level message passing libraries used by the implementations of parallel models are backwardly compatible across the machines within the same range. This still does not cater for changes in the class of parallel architecture. Nonetheless, if parallel computing is to achieve greater popularity, it should attempt to insulate the software from changes in the underlying parallel hardware.

3.4.2 Treatment in Related Work

Shared memory parallel computers typically provide communication through standard concurrency control tools such as shared variable semaphores. This model of computation alleviates the programmer of the need to decompose and map the problem at the expense of potential scalability and portability problems. Non-shared memory machines communicate through message passing. The use of standard message passing libraries such as MPI [mpi97] provides portability for application code. Code portability between shared memory and shared-nothing systems is difficult due to the fundamentally different models of parallelism. Systems based on both explicit and implicit models of parallelism aim to provide portability as a fundamental prerequisite. However, implicit models of parallelism are more abstract since they are honed for problem stipulation, rather than mapping to particular parallel machine classes.

Both the approaches of homomorphic skeletons (§2.2.5) and algorithmic skeletons (§2.2.5) lead to a declarative specification which provides a machine class independent programming interface. The efficient implementation of each skeleton needs to be done only once for each architecture by system implementors. For each skeleton in P^3L there is a set of implementation templates, each of which deals with load balancing issues. for each target architecture. The compiling tools in P^3L generate object code using performance models associated with each skeleton together with parameters relating to the particular virtual machine for which code is to be produced. Implementation templates which comprise a process graph model and associated performance models are incorporated into the compilation process. Portability just requires implementation templates for new architecture.

As previously stated, data–intensive applications which have relied on parallel (relational) database systems to deliver parallelism to applications, have been insulated from low-level architectural details through the provision of a high-level machine-independent query language interface, namely the SQL query language. This parallelism does not extend into the whole application. A model that exploits both application level and database level parallelism must abstract specific architectural details from the programmer.

3.4.3 Thesis Approach

The use of algorithmic forms of computation based on skeletons allows applications to be written as a hierarchic composition of parallel forms. These forms are independent of any particular architecture. Each time that an implementation is required on a new parallel machine, all that is required is to incorporate the cost parameters for the parallel machine into the compilation model, and to rework each of the skeletons for the new class of architecture, if applicable.

3.5 Extensibility

3.5.1 Motivation

Some applications may require skeletons other than the *pipe, farm, hom* and *split* skeletons to model, in the most natural way, all available parallelism.

The collection types which are available in SQL are limited to sets and bags. OQL [Cat96] provides additional data types in the form of lists and arrays, and is extensible at the method level. However, parallel evaluation or even optimisation of methods requires a well–defined semantic specification. No language or system based on SQL or OQL is data type (and collection type) extensible. A model for parallelism for data–intensive applications should allow the addition of application–specific data types. The new data types should be seamlessly integrated into the language and operations involving them should be parallelised. Further, the way in which operations over new data types are parallelised should depend on properties of the data types only. The addition of new data types has not been incorporated into any commercial implementation. Furthermore, facilities for the addition of new operations or re–implementations of existing ones (such as nesting), are not provided in the majority of implementations.

3.5.2 Treatment in Related Work

Explicit data parallel and task parallel models offer extensibility, both in the provision of new (parallel) operations, and the incorporation of new data types. Although these models provide the necessary abstractions, realising data type extensibility requires a large amount of work. Implicit models are less flexible and there is little extensibility with respect to addition of new parallelisable operations. Although the skeletons-related approaches (§2.2.5) do not incorporate explicit facilities for adding new skeletons, the model may be easily augmented to allow this. In parallel database research, only Volcano (§2.3.3) allows the definition of new operators, which can be parallelised using the `exchange` operator. However, the incorporation of new collection types into a parallel database environment has not, to our knowledge, been treated.

3.5.3 Thesis Approach

Two forms of extensibility are considered: adding of new skeletons that implement a stereotypical form of parallelism; and, data type extensibility. The incorporation of a new skeleton only requires the addition of a new implementation template for the skeleton, which is easily performed. Hence, applications that require parallel constructs that closely model the parallelism within them have the potential to incorporate these.

The use of monoids to model simple and collection types offers opportunities for the extension of data and collection types. Monoid definitions and their respective properties (such as associativity, commutativity and idempotency) may be declared in a high-level abstraction thereby facilitating extensions of the built-in collection types. The proposed framework in which data (collection) types and operations on them are modelled as monoids facilitates the seamless addition of new operations over monoids. In the proposed framework, the only operation which handles iterations over elements of a collection type is the monoid homomorphism. The functionality of the monoid homomorphism is independent of any particular collection type.

3.6 Correctness

3.6.1 Motivation

The correctness of large scale software systems developed for sequential machines is usually not established formally: systems are subject to tests which increase the programmer's confidence. The many potential sources of error with software for parallel machines, together with the proliferation of parallel architectures, makes it difficult to both develop software and deduce correctness. This means that a model for parallelism should promote a development methodology which nurtures correctness, or at least, that makes it possible for this property to be established by the system. This necessarily requires, at the minimum, the separation of the semantics of an application from the implementation.

With respect to the data processing component in particular, the model should incorporate type checking of expressions involving operations across multiple data types (collections). This should be based on the properties of each collection type which influence the maximum amount of exploitable parallelism.

An advantage of the relational data model is the existence of a formal calculus and algebra to model database queries. However, the relation model is set based whereas SQL is bag based.[3] Further, in practice, the formalism has failed to model many of the features present (for example, aggregation, grouping and duplicate elimination), in commercial query languages, such as SQL. In query processing, these features are implemented in an ad-hoc manner. The gap widens as one moves to object-oriented languages, such as OQL [Cat96], which deals with multiple collection types and nesting of type constructors amongst other features. Hence, an approach which aims to model application-wide parallelism should model both the query and non-query parts of the application based on a firm semantic foundation. Thus, only a process which incorporates software building which is correct by construction can work. Ad-hoc implementation techniques cannot be expected to work.

A model for parallelism for data intensive applications should be sufficiently expressive to model tasks carried out in the query and non-query components of an application. In addition, it should have well-defined control–structures that model parallel behaviour. Furthermore, the model should enable the verification of the applicability of optimisations across a range of collection types based on formal rules and properties of the particular data types and the model for parallelism itself.

3.6.2 Treatment in Related Work

The coupling between the specification of a task and its implementation governs the ease with which the correctness of the application can be deduced. Implementation of implicit models for parallelism make this task easier than more explicit models, in which the application semantics are integrated into the implementation.

In explicitly task parallel models it is difficult to reason about the behaviour of programs, especially in the presence of dynamic behaviour. This largely stems from the fact that many task parallel models have no formal grounding in the way that they model concurrent activities. Exceptions include *communicating sequential processes* (CSP) [Hoa78] and CCS [Mil80]. Data parallel models are easier to reason about but cannot be considered in isolation since they have no support for modelling task parallelism. The approaches based on using pre–defined forms allow reasoning about the behaviour of programs and enable the deduction of some notion of correctness. For example, the approach of using algorithmic skeletons (§2.2.5) encourages the programmer to think of the problem in terms of stereotypical forms which can be composed to build complete applications. Each of these stereotypical forms has well-defined behaviour and operational semantics. Further, the hierarchic composition of these forms also follows well-defined rules. Although in some approaches, expressions involving skeletons are subject to rewrite rules, these equivalences

[3] For a detailed analysis of the repercussions of this see [Mur93].

are based on algebraic properties pertaining to each of the forms and their composition. The changes to the program are of a structural nature and are meaning–preserving.

The approach of Buneman *et al.* (§ 2.4.4 [BNTW95]) and Fegaras (§ 2.4.5 and [FM95a]) is based on the *universal mapping property* which states that every monoid homomorphism $f : \mathcal{F} \rightarrow \mathcal{M}$ from a free monoid $(\mathcal{F}, \star, e)$ to a monoid $(\mathcal{M}, \circ, u)$, is uniquely determined by a set function from the underlying set of the free monoid to the result monoid $\mathcal{M}$, and that every set function from this underlying set to the monoid $\mathcal{M}$ is *uniquely* determined by a homomorphism. The repercussion of the universal mapping property is that the generator of the free monoid, (S, i), together with a set function, uniquely determine a homomorphism. There are no equational properties on set functions whereas there are equational requirements on functions between monoids.[4] In general, it is not possible for a compiler to check for these equational conditions. The approach taken by Buneman *et al*, is to limit manipulations to the set monoid and incorporate required operations, such as *sum*, as built–in functions.

3.6.3 Thesis Approach

A combined approach of using a small number of algorithmic skeletons together with monoid homomorphisms is adopted for the modelling of application–level parallelism. When considering the correctness of parallel implementations, the most difficult task is deriving a model of the parallel behaviour exhibited by the system. By basing the skeleton model on the principle of abstracting the parallel behaviour of an application from its sequential behaviour, the problem of elucidating the parallel correctness of an application becomes easier to solve since this behaviour is evident from the skeletal process graph of the application.

Considering the query component of applications, the monoid homomorphism is used as the sole means of achieving data parallelism. The monoid homomorphism is used as the sole operator in the underlying query modelling technique, although a high–level language based on monoid comprehensions is also incorporated. A formal basis for query modelling enables provably correct optimisations to be carried out on queries. The approach of incorporating operations across multiple collection types based on monoids and monoid homomorphisms allows the modelling of a greater range of queries within a single framework, when compared to the approach of using built-in functions to model simple operations over multiple collection types. Furthermore, by defining a parallel implementation, the monoid homomorphism may be integrated into an overall skeleton model that models parallelism in both the application and the query.

3.7 Efficiency & Cost

3.7.1 Motivation

A model should be efficiently implementable over a range of parallel architectures. For most problems, a level of performance as high as possible on a given architecture is unnecessary, especially if it is obtained at the expense of development and maintenance costs.

[4] A full treatment of this is performed in chapter 6.

Skillicorn [ST98] notes that to achieve efficient implementation across many architectures it is necessary to: reduce the amount of communication allowed in programs by a factor proportional to the diameter of realistic parallel computers; and, make computations more regular, so that processors do fewer different operations at each moment for SIMD architectures. The amount of communications that a program carries out can be reduced in two ways: either by reducing the number of simultaneous communication actions, or by reducing the distance that each travels. Only models which limit the frequency of communication or are restricted enough to make local placement easy to compute can be efficiently implemented across a full range of target parallel machines.

A model must allow the cost of its operations to be calculated, and enable the programmer to determine which choice of algorithm is suited to a particular architecture. This will feed into the software development process. Programmers make decisions on one algorithm rather than another based on some notion of complexity. It is thus important for cost to be accurately reflected in the model so that such decisions can be made. Skillicorn states that a model has cost measures if it is possible to determine the cost of a program from its text, minimal target computer properties (at least the number of processors that it has), and information about the size, but not the values, of its inputs. This requires that models provide predictable costs.

3.7.2 Treatment in Related Work

In models which have a dynamic thread structure, such as SISAL (§2.2.6), CC++ (§2.2.4), MPI-2 [mpi97], there is no way to limit communications volume. Furthermore, with such models the communications capacity of some classes of parallel computers will be overstretched. Hence, these models do not include cost measures since program costs depend on runtime decisions and hence cannot be predicted. Algorithmic skeleton models however (§2.2.5) have a static thread structure, but do not limit communications, which may again result in network flooding. Models such as homomorphic skeletons (§2.2.5) and NESL (§2.2.5) have a static thread structure, and also limit communications such that the execution time of programs is implicit in each program's structure. The NESL compiler uses cost indicators based on the PRAM model [FW78]. The homomorphic skeletons model is based on operations over each collection type. The communication required for each form is deducible from the structure of the data type, so each implementation needs to construct an embedding of this communication pattern in the interconnection topology of each target computer. List homomorphisms, for example, require the existence of logarithmic depth binary trees in the target architecture interconnection network. Then all communication can take place with nearest neighbours (and hence in constant time). Efficient implementations can be built for any target computers into whose interconnection topologies the standard topology can be embedded. Because the complete schedule of computation and communication is determined in advance by the implementer, cost measures can be provided [SC95].

3.7.3 Thesis Approach

A model which allows fully dynamic process behaviour will not be efficiently implementable or possess cost measures. This is due to the potential to generate excessive communication, and the fact that the cost of communication depends on the interactions of processes whose existence and placement is not known until runtime.

A model that does not allow the dynamic creation of processes is more likely to to permit predictable performance (even then, it may still allow too much communication). Only when the structure of the program is static and the amount of communication is bounded can a model satisfy both of these requirements. The approach of using algorithmic skeletons delivers a static threading structure, but may subject the interconnection network to excessive communication demands. The approach of homomorphic skeletons reduces the effects of this problem by incorporating local data placement thereby reducing the distance that each communication travels.

The approach which is adopted incorporates a small number of skeletons. This offers potential for making use of the natural local data placement of the homomorphic skeletons model, thereby limiting communications by reducing the average distance that messages travel.

3.8 Summary

This chapter has been devoted to elucidating a set of requirements for supporting data–processing applications, both for *decision–support* and *complex applications* in a unified manner. Relevant work in the areas of parallel programming models, parallel database systems and processing, and database query modelling were highlighted in forming the approach necessary for addressing each of the issues within its own sphere. The promising approaches which have applicability across more than one of these requirements were pinpointed. Chapter 4 presents a unified framework that address the requirements presented in this chapter.

Chapter 4

Framework

Chapter 3 defined a core set of requirements for a unified model of parallelism for data–intensive applications and evaluated the related work in chapter 2 with respect to these criteria. In this chapter, an overview of the complete framework for dealing with the requirements presented in chapter 3 is presented. Details are provided in subsequent chapters.

4.1 Approach

The general approach is based on the use of algorithmic skeletons, although provision is also made for expressing queries in a declarative form. A unified approach is developed to the two application areas under consideration in this chapter: *decision–support database queries* (§ 4.2), which are defined to be applications that comprise of just queries. and *complex applications* (§ 4.3), which consist of queries and procedural language statements. The former usually consist of ad–hoc database queries. The latter consists of whole applications which comprise both query and non-query components. An architecture is presented for each application area.

The framework presented draws upon the advances made in two areas: programming parallel computers using *structured parallel programming* (§2.2.5); and query modelling techniques (§2.4). This chapter provides an overview of the model presented in this thesis. Details follow in later chapters.

4.2 Decision Support Queries

This section outlines the architectural model for decision support queries.

The skeleton most appropriate for modelling database queries is the *reduce* skeleton. Although this skeleton is appropriate for use as part of a parallel data–processing language for *complex applications*, it is not suitable for use in a query language due to its operational nature. A query language for decision support applications should be of a similar level of sophistication to query languages in use today, such as SQL. In order to address this issue, the decision–support framework utilises *monoid comprehensions* as a high level query notation. In addition to offering a uniform notation for queries involving multiple collection types, monoid comprehensions are also suitable as a target language for translating SQL queries. Further, a monoid comprehension is merely a high–level formulation of an operational form, the monoid homomorphism.

Monoid homomorphism expressions are amenable to source–to–source optimising transformations. This includes the application of many of the algebraic optimisations that are applied to relational and object–oriented queries. Although the monoid homomorphism is a restricted form of the reduce skeleton, it has a similar data parallel implementation. This makes it a suitable candidate for inclusion into an architecture for decision–support queries and complex applications. Support is provided in the architecture to enable ad–hoc decision–support queries to be programmed in the high–level declarative notation using monoid comprehensions, or a current database query language such as SQL.

Queries may either be expressed in the monoid comprehension language (termed *MQL*) or in a database query language such as SQL, in which case an SQL to MQL translator is applied (§ 7.6). Subsequently, MQL source code is processed to produce homomorphism language code (termed MHL). The following example shows a query in SQL, MQL and MHL:[1]

<table>
<tr><th>SQL</th><th>Monoid Comprehensions</th><th>Monoid Homomorphisms</th></tr>
<tr><td>SELECT name</td><td></td><td>hom[bag,bag] λ cd.</td></tr>
<tr><td>FROM cds</td><td>bag{name(cd)|cd←cds,</td><td>(if (distributor(cd) = "EMI")</td></tr>
<tr><td>WHERE</td><td>distributor(cd)="EMI"}</td><td>then unit[bag](cd)</td></tr>
<tr><td>distributor="EMI"</td><td></td><td>else zero[bag]) cds</td></tr>
</table>

MHL is then processed by the *MHL Rewriter* which applies the source–to–source rewrite rules that are presented in Appendix C to the MHL source code. The output of the MHL Rewriter is processed by the Parallel Type Checker (PTC) which determines the skeleton program necessary to obtain correct parallel evaluation for the input MHL query (§ 7.9). The output of the PTC module is a skeleton program which is processed by the *Code Generator* (§ 7.10). The object code generator produces a *Annotated Skeleton Program* (ASP) for evaluation on the runtime software system on the parallel machine (chapter 8).

In order to realise a correct parallel evaluation for an MHL program, the PTC requires access to the following information sources:

- a *monoid instance dictionary*, which holds information on the monoid instances in the database (cf. tables in a relational database), and a *monoid element dictionary* that holds information on the structure of the individual elements of each collection (cf. a row in a relational database) (§ 8.1.2.5)

- a *data partition dictionary* which stores details on the way in which each resident monoid instance has been partitioned on the parallel machine (§ 7.3).

The PTC constructs a *symbol table* in which it stores the predicates that are associated with each query variable (§ 7.9.2). This Symbol Table is also used by the Object Code Generator.

Figure 4.1 presents the framework for decision-support queries.

[1] The MHL used is in an abbreviated form.

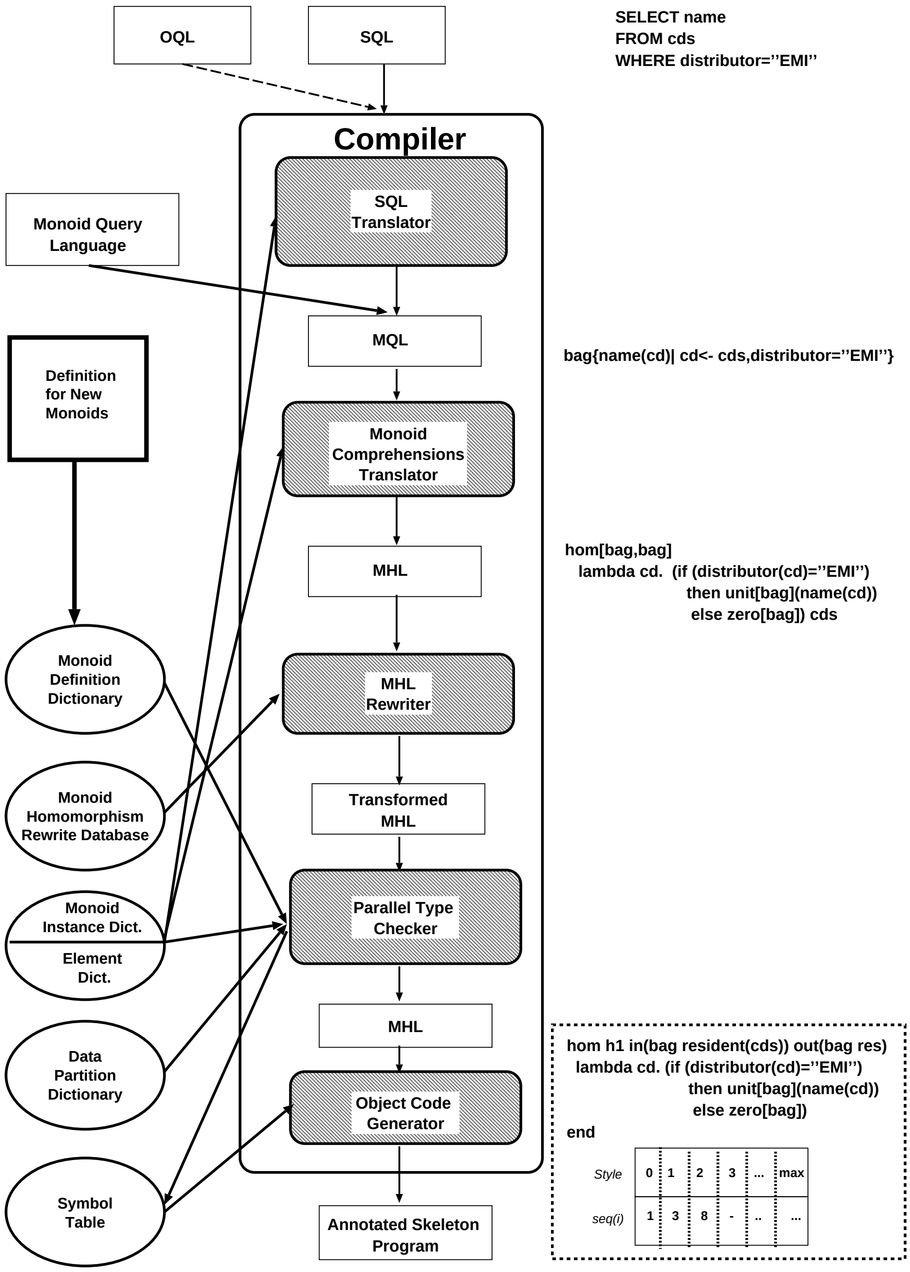


Figure 4.1: Decision Support Interaction Model

4.3 Complex Applications

The goals of the approach advocated in this thesis for supporting parallelism in the query and non–query components of complex applications are programmability, machine abstraction and efficiency. The compiler designer has to implement the functional templates that are required for each parallel form on a once–and–for–all basis.

The approach that is advocated is based on structured parallelism in which the useful forms of parallelism are incorporated as primitives at programming language level. No low–level means of stipulating communications or parallelism is provided. These primitives are used to structure applications that are currently developed in a procedural language. In order to provide continuity from the current language paradigms used in application programming and enable software reuse, the skeletons are provided as a means of structuring sequential code (such as C/C++/Java).

The basic collection of skeletons that have been identified for application–level parallelism for data–intensive applications are the `pipe` and `farm` skeletons for task parallelism, and two data parallel skeletons, namely a `hom` skeleton modelling *combining tree parallelism*, and a `split` skeleton modelling *dividing tree parallelism.* The parallelism exploitation patterns are used to compose both sequential and parallel parts of the application code in a structured way. Thus, a complete application is constructed through the *composition* of these primitives, each of which has a well–defined input and output interface. An application may thus be viewed as a construct tree in which the nodes are algorithmic skeletons and the leaves are sequential modules. Nesting within the construct tree represents *layered* parallelism.

The body of the `hom` skeleton however, must be in a form that is amenable to optimisations. Since a sequential language is unsuitable for this purpose, an appropriate language, *MHL*, has been designed and implemented. Complex applications that are developed using the paradigm may contain embedded monoid comprehensions, This application development paradigm is similar to the existing approaches which embed database queries in procedural language statements.

Figure 4.2 shows the model for complex applications which consist of query and non-query components. The interaction model is based on the model presented for decision-support queries (§ 4.2).

A complex application may consist of a complete skeleton program with MQL or MHL expressions. However, since the query parts of the application may be written in the MQL language, this part undergoes a process of translation. During the first phase, *each* piece of MQL code in the application is translated into the MHL equivalent which is substituted in place of the original MQL expression. Any other skeleton's code in the skeleton program is left unchanged. During the next phase, this process is repeated except that the MHL expressions are subject to processing by the PTC (§ 7.9): the result of the application of the PTC module to each MHL expression is substituted in place of the MHL expression. The output of this stage is an application that consists of skeleton code only, namely, there is no MQL or MHL. This combined skeleton program is subsequently processed by the *Skeleton Program Compilation* module to produce the native object code for evaluation on the parallel machine. The internal structure of this Skeleton Program Compilation module is given below (and figure 4.2):

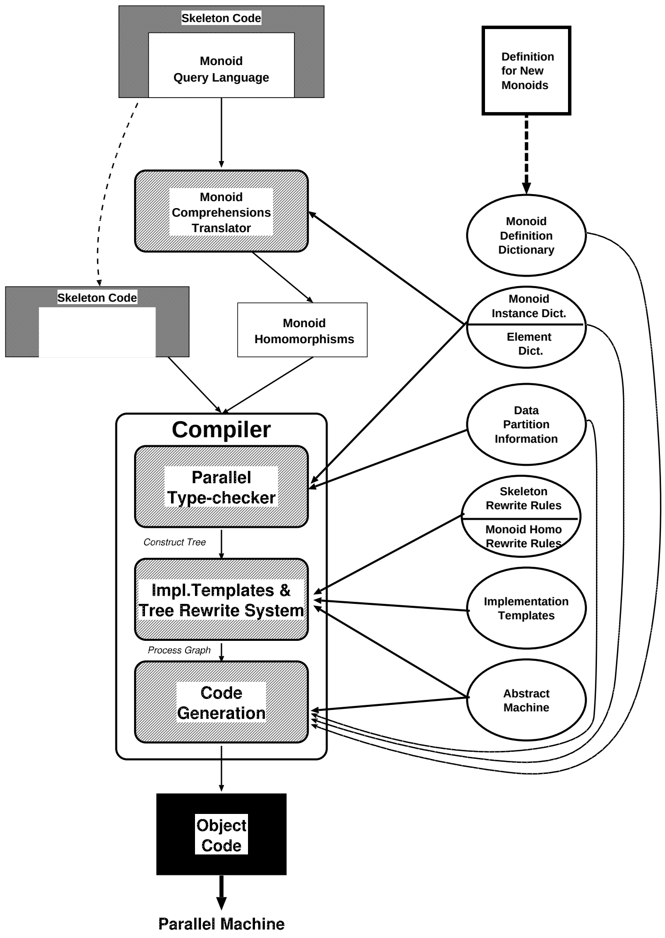


Figure 4.2: Complex Applications Interaction Model

- First phase:
  - Skeleton tree generation
- Second phase:
  1. Program transformation — Skeleton programs are subject to a series of rewrite rules:
     (a) MHL rewrite rules[2]
     (b) Skeleton rewrite rules
  2. Substitution of implementation templates — The compiler incorporates *implementation templates* for each of the skeletons and its compositions. An implementation template is a particular static network of communicating processes plus a performance model for this static network. There may be a number of implementation templates, one for each class of different distributed–memory MIMD machines for which an implementation of the model is required. In order

[2]The MHL Rewriter in the decision–support architecture also performs this function.

to effectively exploit the features of each target architecture, the compiler tunes each skeleton implementation using the performance and complexity models of the specific target parallel platform. The resultant process graph is used in the code generation phase.

- Final phase:
  - Code generation — The code generation phase make use of the features of the target architecture which are exported through an abstract machine. The output of this stage may be object code that is suitable for compilation and execution on the parallel machine, or some interim target code. Taking the compiled route means that any necessary runtime components would need to be linked into the users' application. The alternative interpreted approach of producing an interim format, such as byte code, means an interpreter on the nodes of the parallel machine would have to map the interim code to native machine code. The recent interest in interpreted solutions, namely the Java Virtual Machine, and just in time compiler technology (JIT), mean that the choice between interpreted and compiled solutions is becoming blurred.

The compiler architecture presented for complex applications is motivated by the compilation tools in P^3L [Pel93].

4.4 Summary

This chapter has described the approach used to support both decision–support and complex applications. The distinguishing feature is the unified way in which support is provided for both application areas based on same set of low–level parallel primitives.

Chapter 5 presents the individual skeletons that constitute the skeleton language which is used as the basis for obtaining parallelism from the query and non–query components of complex applications. Chapter 6 concentrates on the `hom` skeleton. The formal foundation of this form is described. The MQL and MHL languages are also defined in this chapter. A prototype parallel machine implementation for the decision–support model is outlined in Chapter 8.

Chapter 5

Application Level Parallelism

Chapter 4 presented an architecture for decision support queries and an architecture for complex applications that are based on the *algorithmic skeletons* model of parallelism. This chapter considers application–level parallelism and defines the form and functionality of each of the algorithmic forms that comprise the *skeleton language* component in figure 4.2. These constructs are used in the construction of applications. The theory underlying the database language and the data parallel form, `hom`, is discussed in Chapter 6.

5.1 Algorithmic Skeletons

In the algorithmic skeletons model of parallelism, the parallel behaviour of an application is modelled using the hierarchical composition of a fixed set of constructs each of manageable size and well–defined parallel behaviour. Since the input and output interfaces of each construct are declared statically, compile time type checking may be performed. Furthermore, programming with algorithmic skeletons moves the burden of process graph mapping, deciding upon the appropriate granularity of parallel computation, scheduling, load balancing and managing communications from the programmer onto the compiler. In the general case, some of these problems are known to be NP–hard, but owing to the use of a restricted set of parallel forms whose performance properties are well–known, it is possible to model their impact analytically. Taking into account the result obtained by using analytical models along with the specific features of the target architecture, the compiler can choose one of the possible implementations for each algorithmic form with a view to maximising performance or efficiency.

Each construct can be thought of as a dataflow construct that computes, in parallel or sequentially, a function on a given stream of input data and produces an output stream of data whilst preserving input and output ordering. A construct may either be a *sequential* construct, a *main* construct or a *parallel* construct, and is denoted by its input and output parameter lists and its name. The input parameter list denotes the parameters passed, by value, to the construct and the output parameter list those passed back. The body of a construct comprises one or more body construct calls. Each construct call can be a call to another construct or an in–line construct (a construct declared by means of an in-line declaration). The stream produced by one construct may be used as an input into another parallel or a sequential construct. Applications can thus be built by the hierarchical composition of constructs.

The form, usage and behaviour of each of the constructs is outlined in the following sections.

5.1.1 pipe

The `pipe` constructor models pipeline parallelism. A set of stages execute in turn over the elements from an input stream producing an output stream of results.

Formally, suppose that a function $f : \alpha \to \zeta$ can be expressed as the composition of n functions $f_1, \ldots, f_n$ such that $f_1 : \alpha \to \beta$ and $f_2 : \beta \to \gamma, \ldots, f_n : \varepsilon \to \zeta$, then pipeline evaluation can be exploited by executing each f_i using a different processing resource. Pipeline evaluation is realised using the `pipe` skeleton which normally serves as a "glue" binding other skeletons together (see figure 5.1). In order to perform this task efficiently, it implements flow control and load balancing between its prospective stages.

```
pipe P in(int x,y) out(int res)
     P1 in(int x) out(int z)
        ${ ... z= f(x); return z; }$
     end
     P2 in(int y,z) out(int res)
        ${... res=g(y,z); return res; }$
     end
end
```

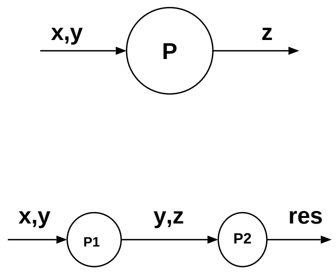


Figure 5.1: `pipe` Process Pipeline

Note that the input parameter of each stage of the pipe need not match the output parameters of the previous stage. In the above example, the variable y is not required for P_1 but nonetheless is piped through P_1. This feature facilitates software reuse since the interfaces between modules need not match identically, but may instead overlap.

5.1.2 farm

The `farm` skeleton models process pipelines. Three forms of *farms* are discussed: a *pure farm*, which deals with the concurrent evaluation of a single function by different workers on different elements in an input stream; an *independent farm*, which implements independent parallelism between a number of worker processes evaluating different functions on elements on the same input stream; and a *guarded farm*, in which the execution of independent worker processes is dependent upon particular values of elements from the input stream.

Pure Farm

In pipeline parallelism each construct deals with a specific part f_i of the overall computation f. If f cannot be considered as a composition of n distinct parts, then parallelism may be exploited between whole computations of f on different input elements. This behaviour is captured by the **farm** construct which models processor farm parallelism between distinct activations (figure 5.2). A set of identical workers execute, in parallel, the tasks which come from an input stream, and produce an output stream of results. In practice, a number of worker processes are created to which elements from the input stream are assigned. Note that no reference is made to the number of workers at the source code level. The appropriate number of workers would be decided by compiling tools based on characteristics of the target architecture using techniques such as Pelagatti's [Pel93].

```
farm P in(bag distributors) out(int cardinality)
     W in(bag distributors) out(int element_sum)
       ${...
         cardinality = calculate_card(distributors);
         return cardinality;
         }$
     end
end
```

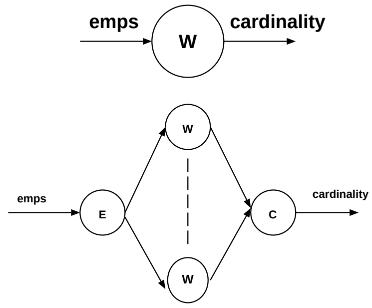


Figure 5.2: Pure **farm**

The process graph for the **farm** skeleton consists of an *emitter* process that assigns work to individual farm workers within the farm, and a *collector* process that gathers the results from the workers and outputs. Each worker receives a partition of the input data of the farm. The output of each worker will be a data partition of the whole farm. The collating function which is built into the collector process handles orderings between separate invocations of the farm either as determined by the type of the input, or FIFO ordering, as appropriate. The merging of individual results which are produced in response to a single input stream element is also handled by the collating function. Note that the data parallel processing of elements of a monoid type should be programmed using the **hom** skeleton and not a **farm**.

Independent Parallelism

The farm construct may also be used to exploit task parallelism between independent tasks. Each data element on the input stream is passed to all the process workers which process the element. The collector process of the farm collates the ouput from each worker and outputs a single result from this. Note that this is not the usual implementation for the independent farm in which the collector process simply outputs individual results from the workers. For the example given in figure 5.3, for each bag in the input stream, the results *res*1 and *res*2 are evaluated and put onto the output stream.

Guarded Farm

In order to incorporate the flexibility to perform parallel actions based on results of sequential or other parallel actions, a *guarded farm* constructor may be used (figure 5.4).

```
farm P in(bag emps) out(min res1, max res2)¹
     W1 in(bag emps) out(int res1)
        ${
          ...
          res1= find_min_salary(emps);
          return res1;
        }$
     end
     W2 in(bag emps) out(int res2)
        ${
          ...
          res2= find_max_salary(emps);
          return res2;
        }$
     end
end
```

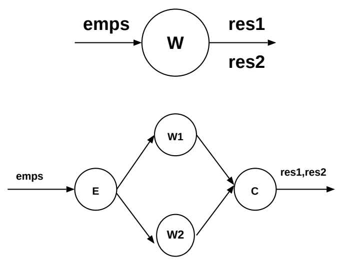


Figure 5.3: Independent `farm`

The execution of each worker on a portion of the data set is conditioned by some guards. The input of the constructor is processed by all workers whose guards are satisfied.

```
farm P in(set emps,int firstone) out(int res)
 firstone == 1:  W1 in(set emps) out(int res1)
          ${...
            res1 = find_min_salary(emps);
            return res1;
          }$
 end

 firstone == 0:  W2 in(set emps) out(int res2)
          ${ ...
             res2 = find_max_salary(emps);
             return res2;
          }$
 end
end
```

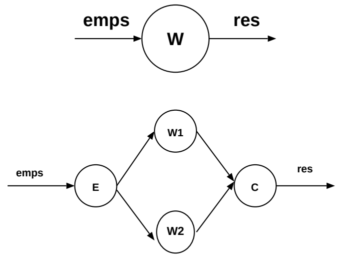


Figure 5.4: Guarded `farm`

5.1.3 `hom`

The **hom** skeleton is similar to the *fold* function over lists in functional languages and models combining tree parallelism on a collection of data. There are two major difference between **hom** and its counterpart in functional languages.

First, in some parallel implementations, the *fold* function manages both the data–partitioning required to distribute data between parallel processing resources, and the *folding* process itself (AGNA [Hey92] uses this approach). In the approach advocated in this thesis, there is a clear separation of concerns between the tasks of data partition and data combination. The **hom** skeleton only models combining tree data parallel evaluation on pre–partitioned data. This data may come in the form of a partitioned stream from an earlier skeleton or from parallel secondary storage operation. The data partitioning itself is handled by the **split** skeleton 5.1.4. Second, the *fold* function takes a single function

[1] **max**, **min** and **sum** are *Ground Monoids* as discussed in chapter 6.

as input and *folds* this over the argument list. The hom construct however, is similar to FAD's pump [Ban92] which takes both a unary operator and a binary operator. The body of the hom skeleton is equivalent to the unary function in FAD and is defined as a λ expression (details chapter 6). The binary function is defined to be the *merge* function of the result monoid (§ 6.3).

The hom skeleton models the behaviour of the monoid homomorphism (§ 6.3) which operates over a *Monoid* argument and has a *Monoid* result (§ 6.3). Further, the hom skeleton may require scalar variables in order to carry out its computation. Hence, additional scalar values may be provided as input to the construct. An individual scalar element may be of type integer, float, boolean, array, structure, Monoid, free Monoid, or α, where α signifies an individual element that is only accessible through a set of methods specific to the type of the element.[2] Hence, the interface to the hom skeleton allows both Monoid and scalar values. This typing scheme also applies to the other skeletons and should be incorporated into the type system of the sequential language.

The skeleton code outlined in in figure 5.5 uses the hom skeleton to calculate the total number of cds that have been released on the EMI label. The keyword **resident** denotes that the collection is resident on secondary storage on a number of processing nodes.

```
hom h in(bag resident(cds)) out(sum res)
    λx.  if recorded_label(x) = ''EMI''
         then 1
         else 0
end
```

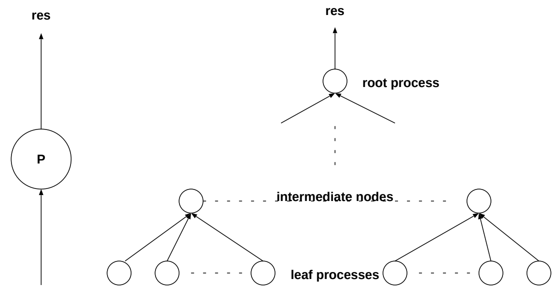


Figure 5.5: hom

Note that generating data from resident collections places restrictions on the nature of the hom process graph since the bottom–level nodes in the process–graph have to be mapped to precisely the same nodes across which the resident collection is partitioned. The depth and breadth of the process graph is determined by the compiling tools using information about the size of the input data, and the expected amount of data that is merged at intermediate level nodes.

Although the syntax of the body of hom differs from that of other constructs, hom may be nested within instances of other constructs, including itself. For example, the code fragment figure 5.6 may be used to define a query that returns the total number of elements in $xs \bowtie ys$. The given parallel evaluation of the relational join denote the case in which the two argument collections are partitioned on the filter $P(x, y)$. In this case, the overall result is obtained through the set union of all partial results from independent parallel evaluations. If the data partitioning on the two relations does not fulfill this requirement, the evaluation of the inner hom may be farmed out. However, the evaluation paradigm used in parallel relational database management systems is based on re–partitioning both of the argument collections such that $\forall x \in xs, \forall y \in ys, P(x, y) \Rightarrow node(x) = node(y)$, where

[2]This disallows pointers.

```
hom h1 in(set resident(xs)) out(sum res)
       λx.  if P(x)
                then hom h2in(α x, bag resident(ys)) out(sum res)
                            λy.  if (P(y) ∧ P(x,y))
                                   then 1
                                   else 0
                       end
                else 0
end
```

Figure 5.6: Implementation of Join Using the `hom` Skeleton

$node(i)$ returns the physical node on which element i resides. If such re–partitioning is carried out, the complete evaluation may proceed efficiently using a single parallel `hom` logical process tree. Note that iterations of the inner collection are performed *locally* at each node by an inner `hom`.

5.1.4 `split`

The `hom` skeleton discussed in § 5.1.3 models combining tree parallelism and requires ready–partitioned data. In order to realise data partitioning, the `split` construct performs dividing tree partitioning of a data collection.

The `split` skeleton has three components. The first is the number of partitions at the leaf level (n). The second determines if the data chunks should be distributed to the same physical nodes as another collection that has already been partitioned. This is applicable to situations in which the partitioning is being performed in order to facilitate combination with another data collection. In such cases the partitioning degree given by n must be the same for both collections, and the physical nodes across which the new collection is partitioned must identically match with those for the former collection. The third component may either be the keyword `DUPLICATE`, or a `division_function`, which is the partitioning function that is used to partition elements of the data collection. The `division_function` is defined as a sequential construct and may take the form of a hash function which returns the logical partition for each element. The outline of an instance of the `split` skeleton is shown in figure 5.7. Although both the input and output types of the `split` skeleton are collections, the skeleton outputs a partitioned collection, which can be statically or dynamically piped into a `hom` skeleton or written to parallel disks. In order to write data to disk at each node the keyword *resident* is prefixed to the collection's name in the input parameter list, as shown in the code segment in figure 5.7. The `split` skeleton is used to realise the data–partitioning requirements for nested `hom` skeletons in both decision–support queries and complex applications and is applicable to MIMD, SIMD and shared–disk architectures.

5.1.5 `seq`

The `seq` construct affords flexibility by enabling the programmer to use existing code fragments. It constitutes the leaves of the hierarchical composition and contains the sequential code blocks of the application. A sequential code segment is written in a procedural language such as C/C++/Java and is encapsulated by braces of the form

```
split split1 in(bag data_in)
      out(bag resident(data_out))

      int n;

      monoid_name x;

      division_function in(α tup) out(int n)
            {
             ...
            }

      end

end
```

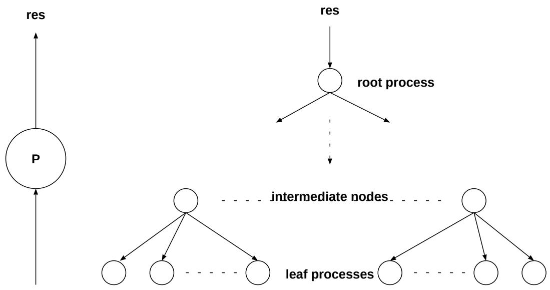


Figure 5.7: `split`

${ *code* }$ with clearly marked *in* and *out* interface information. The `seq` construct works in a purely functional way, applying its sequential body to each of the elements within the input data stream — there is no notion of distributed state. Hence, sequential code may not contain any reference to a parallel construct.

In order to produce an efficient parallel implementation it is necessary to obtain information about the complexity of sequential constructs. This can be obtained through profiling methods and is used by the compilation tools during the process graph construction phase.

5.1.6 Application Interface

Data–intensive applications may be constructed using only the parallel constructs, or by embedding MQL queries in place of (possibly nested) `hom` constructs in the main skeleton program. In the latter case, *each* MQL query that is embedded in the skeleton program undergoes the following translation processes during which an appropriate skeleton expression replaces a corresponding MQL term.

$$MQL \Longrightarrow MHL \overset{PTC}{\Longrightarrow} skeleton\ \ program$$

The complete resultant application is processed as a complex application as determined by the complex applications architecture.

5.2 Examples

This section introduces an example that shows the way in which the algorithmic forms may be used in the construction of complete applications. The application calculates the median and standard deviations of the selling prices of cds. This information is displayed graphically in the form of a histogram that plots, for the top ten selling cds, the total number of cds that have been sold in each of a number of price intervals. Table 5.2 shows a code segment for this application.

The main part of the application returns the total number of sales that are within each of eight price bands, where each price band is determined by the mean μ and standard deviation σ of the selling price for a particular cd. This main result is evaluated by the *calculate-nums-each-sd* function. The form of each result element is given in table 5.1. The

cd	$(< \mu - 3\sigma)$	$(\geq \mu - 3\sigma) \wedge (< \mu - 2\sigma)$	$(\geq \mu - 2\sigma) \wedge (< \mu - \sigma)$	$(\geq \mu - \sigma) \wedge (< \mu)$
	$(\geq \mu) \wedge (< \mu \wedge \sigma)$	$(\geq \mu + \sigma) \wedge (< \mu + 2\sigma)$	$(\geq \mu + 2\sigma) \wedge (< \mu + 3\sigma)$	$(\geq \mu + 3\sigma)$
cd_i	3240	15643	134202	198543
	205574	115108	32071	8000

Table 5.1: Format of the Final Result

data for calculating the variance and the mean value is gathered by the *calculate-variance-and-sum-and-count* function which collects, by $cd(sale)$, the totals: $\Sigma\ amount(sale)^2$, $\Sigma\ amount(sale)$, and $\Sigma\ 1$. The mean and the variance as defined:

$$\sigma^2 = \sum_{i=1}^{n} \frac{e_i^2 - \mu^2}{n}$$

are calculated from this result by the *calculate-sd-and-mean-from-res* function, where n is the total number of elements for which the sum Σ is evaluated. The standard deviation and mean price for each cd are fed through to *calculate-nums-each-sd* which performs eight concurrent iterations over the sales collecting the number of elements that lie within each of the intervals, as defined in table 5.1. The result of the *calculate-nums-each-sd* function is piped through to the `plot` function which sorts the results according to highest total sales value[3] and provides the elements that achieve the top ten total sales to the *plot_histogram* function. This function plots a two dimensional histogram of the results. Note that any data re–partitioning that is required at some stage of the evaluation has to be realised through the incorporation of the `split` skeleton by the compiler.

In order to realise executable code for this application, it would be necessary to substitute the appropriate $grouped(V \times \mathcal{N})$ Monoid in place of each *grouped* monoid in the skeleton code. It is also necessary to define implementations for the `hom` expression that have not been defined, and a sequential or parallel implementation for each of the constituent functions of the *plot* function. At present, this application could not be evaluated using the current prototype implementation since it does not support the $grouped(V \times \mathcal{N})$ monoid, which is extensively used in the example application.

5.3 Summary

This chapter has outlined the constructs that constitute the skeleton language that is used to develop applications for the *complex applications architecture*. Applications are structured through the hierarchical composition of this fixed set of parallel forms, and one sequential form. Task parallelism at the application level can be exploited through the use of `pipe` and `farm` parallel forms. Data parallelism may be realised in the form of dividing tree parallelism that is modelled by the `split` skeleton, and combining tree

[3] Which is obtained by the addition of the totals across each of the eight categories.

```
farm filter-collections in() out(bag filtered-cds,filtered-sales)
  hom filter-cds in(bag resident(cds)) out(bag filtered cds)
    λ cd.  (if (year-distributed(cd) = 1998)
              then unit[bag](cd)
              else zero[bag])
  end

  hom filtered-sales in(bag resident(sales)) out(bag filtered-sales)
    λ sale.  (if year(sale) = 1998
                then unit[bag](sale)
                else zero[bag])
  end
end

pipe plot in(grouped(ℰ × ℱ) categorised-sales) out(boolean success)
  parallel_sort in(grouped(ℰ × ℱ) categorised-sales) out(list sorted-categorised-sales)
  chose in(i 10, list sorted-categorised-sales) out(list top-cds)
  plot_histogram in(list top-cds) out(boolean terminate)
  finish in(terminate) out(boolean success)
end

main in() out(boolean success)
  filter-collections in() out(bag filtered-cds, filtered-sales)
  calculate-variance-and-sum-and-count in(bag filtered-sales)
            out(grouped(ℕ × ⟨sum × sum × sum⟩) grouped-variances)
  calculate-sd-and-mean-from-res in(grouped(ℕ × ⟨sum × sum × sum⟩)
            grouped-variances) out(bag sd-and-mean-by-cd)
  calculate-nums-each-sd in(bag sd-and-mean-by-cd, bag filtered-cds)
            out(grouped(ℰ × ℱ) totals-in-each-interval)
  plot in(grouped(ℰ × ℱ) totals-in-each-interval) out(boolean success)
end
```

Table 5.2: Complex Application

parallelism that is realised by the `hom` skeleton. The `seq` construct allows the incorporation of procedural code. In addition to being used in application programs, these skeletons form the operational language which is the target of the compilation process for decision support queries. Note that the degree of parallelism that is exploited at runtime for each construct is decided by compiling tools. Even though a parallel form may be defined for a particular task, the compiler may decide on a sequential execution. Chapter 6 outlines the theory underlying the query model.

Chapter 6

Theory and Query Model

6.1 Introduction

This chapter outlines the theory underlying the query model for decision–support applications. The query model uses *monoids* to model collection and aggregate types, and the *monoid homomorphism* as the sole data manipulation primitive in the *Monoid Homomorphism Language* (MHL). A language based on comprehensions, the *Monoid Query Language* (MQL), is introduced as the programming interface to MHL. Earlier presentations of this work appear in [AK94, ABBM97a, ABBM97b].

6.2 Approach

The use of comprehensions as a concise, expressive and clear notation for modelling queries across many suitably defined collection types has been shown [TCH91, Feg94, BNTW95]. Comprehensions provide a natural way of modelling queries over nested and non–nested collections, and have been advocated as a declarative way of defining, translating and evaluating queries in functional, relational and object-oriented database systems [Tri91, Pou88, Hey92, BLS$^+$94, BNTW95]. They 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 query nesting in OQL [Cat96]. Comprehensions can be defined in terms of monoid homomorphisms. Monoid homomorphisms are very expressive: the monoid homomorphism from set to set has been shown to capture the nested relational algebra [TBW92, TW88]. Further, the monoid homomorphism resembles the `reduce` skeleton which is effective for data parallel evaluations on decomposed forms of data and for divide–and–conquer computations. Hence, it is ideal as a basis for both query modelling and as part of a parallel application development language for data–intensive applications. The next section (§ 6.3) introduces the theoretical basis for MQL and MHL by defining a *monoid*, *monoid comprehension* and *monoid homomorphism*. § 6.6 outlines data type extensibility issues and leads on to § 6.7 which discusses the theoretical and practical aspects of a parallel evaluation paradigm for MQL and MHL queries. § 6.8 discusses query optimisation issues and outlines proofs for some existing optimisations.

6.3 Theoretical Basis

Context The use of monoids as a basis for data manipulation raises the question of what properties the monoid operation must satisfy. By definition a monoid is associative, but should attention be focussed on the commutative case? Is it possible to treat the (Boolean) case in which all elements are idempotent in a uniform way? Without being too formal, the theory might be developed in any one of three categories:

(a) the (unrestricted) category of monoids, M;

(b) the category of commutative monoids, CM;

(c) the category of commutative and idempotent monoids, *CIM*.

The route that is followed is to work wherever possible in the general category of Monoids, which may be specialised to any of the three cases. Where differences arise they are noted.

Def 6.1 ***(Monoid)***

- *A data type $\mathcal{T}$ is expressed as a Monoid $\mathcal{M}$, such that $\mathcal{M} = (\mathcal{T}, \star, e)$, where the function $\star$, of type, $\mathcal{T} \times \mathcal{T} \to \mathcal{T}$, is* associative *with left and right identity e; if for all $a, b \in \mathcal{T}$, $a \star b = b \star a$, then $\mathcal{M}$ is commutative, and lies in the category CM; if in addition for all $a \in \mathcal{T}$, $a \star a = a$, then $\mathcal{M}$ is also idempotent, and lies in the category CIM.*

Def 6.2 ***(Monoid Homomorphism)***

A monoid homomorphism between a Monoid $(\mathcal{T}, \star, e)$ and a Monoid $(\mathcal{M}, *, I)$ is a function $f : \mathcal{T} \to \mathcal{M}$ which satisfies the following:

$$f\ e = I$$
$$f(a \star b) = f(a) * f(b)$$

Note that if $\star$ is commutative (/idempotent) on $\mathcal{T}$, then $*$ is commutative (/idempotent) on $f(\mathcal{T})$.

For example, the function $f : (list(\mathbb{N}), @, [\,]) \to (\mathbb{N}, +, 0)$, as defined below, is a monoid homomorphism.

$$f\ [\,] = 0$$
$$f\ [x]\ @\ xs = x + (f\ xs)$$

Def 6.3 ***(Free Monoid)*** *A free Monoid, $(\mathcal{F}, \star, e)$ generated by (S, i)*[1]*, is such that:*

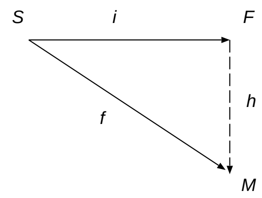
$\forall f : S \to \mathcal{M}$ — where f is a set function and $\mathcal{M}$ is a Monoid

$\exists^1\ h : \mathcal{F} \to \mathcal{M}$ — where h is a monoid homomorphism

$s.t.\ h \circ i = f$

[1] Where S is the underlying set of the Monoid $\mathcal{F}$.

This is shown graphically in the figure below (where the dotted line represents uniqueness of the map):



The free Monoid on a set S is best defined by the universal mapping property (Lemmas 6.1 and 6.2). A free Monoid is referred to as a collection. The free Monoids over S may be identified as follows:

(a) in M, all finite *lists* of elements of S;

(b) in CM, all finite *bags* of elements of S;

(c) in CIM, all finite *subsets* of S.

Given such a collection, it is possible to identify it with a finite sequence of (insertions of) elements of S. A particular collection determines the associated sequence as follows:

(a) M, uniquely;

(b) CM, uniquely up to permutation;

(c) CIM, uniquely up to permutation and repetition.

The Monoid operation on collections (Append for lists, Union for bags/sets) then becomes the concatenation of the representative finite sequences (evidently associative), which is denoted by $\ominus$. It is possible to identify a collection instance with the equivalence class defined by a representative finite sequence. The empty collection over any set is denoted by $\emptyset$.

Universal Mapping Property

The definition of a *free Monoid* leads to the *universal mapping property.* First, consider two lemmas;

Lemma 6.1 *Every monoid homomorphism $j : \mathcal{F} \to \mathcal{M}$ is determined uniquely by a set function $g : \mathcal{S} \to \mathcal{M}$, where $\mathcal{F}$ is the free Monoid $(\mathcal{F}, \star, e)$, and $\mathcal{M}$ is the Monoid $(\mathcal{M}, *, I)$.*

Proof Define $g\ :\ (\mathcal{S} \to \mathcal{M})$ such that $g \stackrel{def}{=} j \circ i$: then by the definition of freeness:

$$\exists^1\ h.\ st.\ h \circ i = j \circ i$$

thus $h = j$, and it is indeed the case that g determines j.

Assume there is a function $g'\ :\ \mathcal{S} \to \mathcal{M}$

$$\begin{tikzcd} S \arrow[r, "i"] \arrow[dr, "g"'] & F \arrow[d, dashed, "j"] \\ & M \end{tikzcd}$$

$$j \circ i = g'$$

but $j \circ i = g \quad \Rightarrow \quad g = g'$, so g is unique.

Lemma 6.2 *Every set function $g : S \rightarrow \mathcal{M}$ is uniquely determined by a function $j : \mathcal{F} \rightarrow \mathcal{M}$, where $\mathcal{F}$ is a free Monoid given by (S, i) and where j is a homomorphism.*

Proof By definition of a free Monoid.

The conjunction of Lemma 6.1, that states that every monoid homomorphism $j : \mathcal{F} \rightarrow \mathcal{M}$ is uniquely determined by a set function, and Lemma 6.2 that states every set function from the underlying set of the free Monoid $\mathcal{F}$ to the Monoid $\mathcal{M}$ is uniquely determined by a homomorphism, constitute a *universal mapping property.* The universal mapping property signifies that there is a *unique* homomorphism $\mathcal{F} \rightarrow \mathcal{M}$ for *every* set function $\mathcal{S} \rightarrow \mathcal{M}$. Alternatively, the generator of the free Monoid $(\mathcal{F}, \star, e)$, (S, i), together with a set function uniquely determines a homomorphism. Whilst there are no equational properties on set functions, in order to be a monoid homomorphism a function must satisfy the conditions of Def 6.2.

In order to implement a scheme for *monoid programming* based on homomorphisms, there are two alternatives: requiring the programmer to write a homomorphism for each Monoid to Monoid function; or, requiring the programmer to provide a set function from the underlying set of the collection $\mathcal{F}$ to the target Monoid $\mathcal{M}$. This set function together with the generator of the collection then determines the *unique* homomorphism from $\mathcal{F}$ to $\mathcal{M}$. Taking the first option would require a compiler to check that the candidate function is a homomorphism. This is not an acceptable solution since the checks that are required are undecidable in general. The alternative of requiring the programmer to just provide a set function from the underlying set of the collection $\mathcal{F}$ to the target Monoid $\mathcal{M}$ is a better solution. This set function, together with the injection from $\mathcal{S}$ into $\mathcal{F}$ *uniquely* determines a homomorphism from $\mathcal{F}$ to $\mathcal{M}$. Since there are no equational conditions to check for set functions, the process of generating the unique homomorphism from the set function may be carried out automatically by a compiler. This forms the basis of the approach used in this thesis.

Lemma 6.3 *The function hom : $\mathcal{F} \rightarrow \mathcal{M}$ is a Monoid homomorphism, where $\mathcal{F}$ and $\mathcal{M}$ denote the collection $(\mathcal{F}, \star, e)$ with generator (S, i), and the Monoid $(\mathcal{M}, *, I)$ respectively, and where hom is uniquely determined by $hom(i(a)) = f(a)$, where $a \in S$.*

6.4 Data Modelling

The computational universe under consideration in this thesis, MT, is a sub–category of the category of all monoids and is specified below.

Def 6.4

MT	$::=$ *PRIM*
	$\mid$ *AGG(MT)*
	$\mid$ *AGG(ground)*
	$\mid$ *MT_list*
MT_list	$::=$ *MT* $\mid$ *MT_list* , *MT*
F	$::=$ *list* (for M)
	$::=$ *bag* (for *CM*)
	$::=$ *set* (for *CIM*)
PRIM	$::=$ *BASE* $\mid$ *AGG*
AGG	$::=$ *list* $\mid$ *bag* $\mid$ *set* (for M)
	$::=$ *bag* $\mid$ *set* (for *CM*)
	$::=$ *set* (for *CIM*)
BASE	$::=$ *sum* $\mid$ *prod* ($\mid$ *max* $\mid$ *min* $\mid$ *all* $\mid$ *some* for *CIM*)
ground	$::=$ *string* $\mid$ $\mathbb{N}$ $\mid$ $\mathbb{B}$ $\mid$ *ground_list*
ground_list	$::=$ *ground* $\mid$ *ground_list* , *ground*

When there are no ambiguities, the notation F is used in place of $F(type)$. The zero element[2] of a Monoid of type $F(t)$ is denoted by $zero[F(t)]$, the merge function by $merge[F(t)]$, and the injection into the free Monoid from the underlying set, t, by $unit[F(t)]$.

F defines the free Monoids for each of the categories M, CM, CIM. In the rest of this thesis the word *collection* is used; the interpretation depends on the category under consideration. The system implemented is accessed through SQL, the corresponding category is CM, and the collection type is the *bag*. The term ground Monoid captures the notion of primitive types, such as natural numbers and booleans which are built in. Some ground and free Monoids are presented in table 6.1.

The core data within the *Swiss Radio's CD Collection* (SR-CD) is modelled as two collections: $bag(cds) = (bag(cds), \uplus, \{\{\ \}\})$, and $bag(tracks) = (bag(tracks), \uplus, \{\{\ \}\})$ for the cds and tracks collections respectively.[4] A number of other collections also exist. Details of these and the precise nature of the underlying sets *cds* and *tracks* appear in Appendix E. Note that since the experimental collections are accessed via SQL, the category of Monoids under consideration is that of commutative monoids, *CM*.

6.5 Data Manipulation

This section discusses the core language, *Monoid Homomorphism Language* (MHL) that is based on the monoid homomorphism. A high–level form for MHL, termed *Monoid Query*

[2]Also known as identity.

[3]Where $\mathbb{N}$ denotes the set of natural numbers and $\mathbb{B}$ denotes the set of boolean values.

[4]Where *cds* and *tracks* denote the underlying set of the *cds* and *tracks* collections respectively.

Free Monoids				
$\mathcal{M}$	***zero***	***Generator (i(x))***	***merge***	*Category*
set	$\{\}$	$\{x\}$	$\cup$	*CIM*
bag	$\{\{\ \}\}$	$\{\{x\}\}$	$\uplus$	*CM*
list	$[\,]$	$[x]$	@	*M*

Ground Monoids				
$\mathcal{M}$	Underlying set (S)[3]	***zero***	***merge***	*Category*
sum	$\mathbb{N}$	0	$+$	*CM*
prod	$\mathbb{N}$	1	*	*CM*
max	$\mathbb{N}$	0	*max*	*CIM*
min	$\mathbb{N}$	$+\infty$	*min*	*CIM*
all	$\mathbb{B}$	*true*	$\wedge$	*CIM*
some	$\mathbb{B}$	*false*	$\vee$	*CIM*

Table 6.1: Examples of Free and Ground Monoids

Language (MQL) is also presented. Discussion of the translation scheme from MQL to MHL is deferred until chapter 7.

6.5.1 Monoid Homomorphism Language (MHL)

An operational language based on the monoid homomorphism, the *Monoid Homomorphism Language* (MHL) is introduced. The sole iterator in this language is the function *hom*, as defined in Lemma 6.3. The use of this iterator for filtering a bag cds with a predicate p is given in the MHL formulation below:

$$hom[bag, bag]\lambda a.\ (\text{if } p(a) \text{ then } unit[bag](a) \text{ else } zero[bag]) \text{ cds}$$

MHL_term	::= *monoid_term* | *monoid_hom*
monoid_term	::= *zero* [*MT*] | *unit*[*MT*] (*filter*) | *merge* [*MT*] (*monoid_term* , *monoid_term*) | *collection_name*
monoid_hom	::= *hom* [*F* , *MT*] λ *variable_exp* . (*hom_body*) *monoid_term*
hom_body	::= *monoid_term* | *conditional_exp*
conditional_exp	::= *if filter then monoid_term else zero* [*MT*]
filter	::= *head_exp* | *head_exp comp_op filter* | *head_exp* / *head_exp*
head_exp	::= integer | *bool_val* | *variable_exp* | *att_sel* | *head_exp_list* | *monoid_term* | *filter*
head_exp_list	::= *head_exp* | *head_exp_list* , *head_exp*
att_sel	::= *attribute_name*(variable_exp)
variable_exp	::= *variable* | variable.*integer*
bool_val	::= *true* | *false*
comp_op	::= < | > | = | != | <= | >=

Table 6.2: MHL BNF

The first monoid in the square bracket indicates that the argument of the *hom* operator is a *bag* Monoid, whereas the second argument denotes the type of the result Monoid, which is the same as the argument in this case. Note that the term *unit*[*bag*(cds)] injects an element of cds as a singleton into the *bag*.

The BNF for MHL is given in table 6.2. Notice that the <conditional_exp> form defines an asymmetric predicate construct. The motivation for this asymmetry lies in the need to realise a dataflow (parallel) evaluation model paradigm for efficiency purposes.

As briefly indicated in Def 6.2, the maps considered are homomorphisms from a free Monoid to a target Monoid *within* the same category as the free Monoid.[5] In order to ensure that all the homomorphisms used at the language level are within the category of the free Monoid over which the homomorphism is defined, checks must be performed on the category of the target Monoid. The check is defined over the free Monoid $(\mathcal{F}, \star, e)$ generated by $(\mathcal{S}, i)$ and the Monoid $(\mathcal{M}, \times, I)$, and requires that $\pi(\mathcal{F}) \preceq \pi(\mathcal{M})$, where π is defined such that M $\preceq$ CM $\preceq$ CIM. The enforcement of this requirement ensures the correctness of the MHL evaluation model. For example, a Monoid homomorphism from the free Monoid $set(\mathbb{N})$ to the *sum* Monoid that captures the cardinality of a set is *not* valid since $set(\mathbb{N})$ is in the CIM category whereas *sum* is in the *CM* category.

The next section discusses the *Monoid Query Language.*

6.5.2 Monoid Query Language (MQL)

The *Monoid Query Language* (MQL) is the high–level declarative form for expressing queries in the system and is based on *Monoid comprehensions*, which may be compiled down to MHL in a way analogous to the compilation of Standard ML may be compiled down to the lambda calculus. MQL and MHL forms of the homomorphism expression which filters a set with a predicate, P, are given below:

MQL	**MHL**
$set\{\ a \mid a \leftarrow xs,\ P(a)\ \}$	$hom\ [set, set]\lambda a.\ (if\ P(a)\ then\ unit[set](a)\ else\ zero[set])\ xs$

The monoid name prefixing the curly brackets in the MQL formulation refers to the result Monoid type, *set* in this case. The MQL comprehension consists of two components which lie either side of the pipe (|). The left and right of the pipe are termed the *head* and *tail* of the comprehension respectively. The term $a \leftarrow xs$ denotes a homomorphism over the collection xs producing a *stream* of singletons, each of which is instantiated to the variable a.[6] The last term in the tail of the comprehension is a predicate term that is applied to each element from the incoming stream, xs. The expression in the head of the comprehension is applied to all instantiated elements in the tail of the comprehension for which the predicate is satisfied. The result of the application of the tail to an element is returned in the form of a singleton of the appropriate collection. The singletons are then combined using the *merge* operator of the result Monoid, $\cup$ in this case.

The BNF for MQL is given in table 6.3. Constructs that are common to both MQL and MHL have been omitted.

[5]This is a direct result of the *Universal Mapping Property.*

[6]The term $a \leftarrow xs$ is also referred to as a *generation.*

MQL_term	::=	*monoid_comp* | *monoid_hom*
monoid_comp	::=	*MT*{ *comp_head_term* | *tail_term_list* }
comp_head_term	::=	*filter* | *MQL_term*
tail_term_list	::=	*tail_term* | *tail_term_list* , *tail_term*
tail_term	::=	*generation* | *filter*
generation	::=	*variable* ← *monoid_term* | *variable* ← *monoid_comp*

Table 6.3: MQL BNF

Some examples of valid MQL expressions follow.

$set\{x \mid x \leftarrow xs, y \leftarrow ys, P(x,y)\}$	(join of xs and ys on filter P)
$bag\{y \mid x \leftarrow xs, y \leftarrow x\}$	(unnest a collection xs)
$sum\{1 \mid x \leftarrow xs\}$	(the sum of natural numbers collection)
$some\{x = c \mid x \leftarrow xs\}$	(c $\in xs$)
$sum\{1 \mid x \leftarrow merge[bag](xs, ys)\}$	(total number of elements in $xs \uplus ys$)
$some\{x = true \mid x \leftarrow bag\{P(y) \mid y \leftarrow ys\}\}$	(there exists y for which $P(y)$ is true)

The SQL query which finds all tracks in the tracks collection for which the parent cd has not yet been entered into the cds collection follows.

```
SELECT t.cd_id
FROM tracks, t
WHERE NOT EXISTS(
                SELECT (*)
                FROM cds, cd
                WHERE t.cd_id = cd.cd_id)
```

The MQL equivalent of this query is:

$$bag\{\ cd\_id(t) \mid t \leftarrow tracks,\ all\{cd\_id(t) \mathrel{!=} cd\_id(cd) \mid cd \leftarrow cds\ \}\}$$

The MQL formulation of a query which records the total first year sales for each artist appearing on a track in the cd database is given by:

$$bag\{\langle name(artist), sum\{sales(cd) \mid cd \leftarrow cds, a \leftarrow artists(cd), a = artist\}\rangle \mid cd \leftarrow cds, artist \leftarrow artists(cd)\}$$

MQL is not restricted to a particular data model. Data model specific aspects are determined by the implementation of the *equality* operator. For example, the query which filters a collection xs may be over a relational or an object–oriented database. Hence, MQL may be used to model both relational and object-oriented queries. However, and more importantly, MQL models queries which cannot be efficiently represented in a language such as SQL. For example, MQL and MHL have forms expressing the evaluation of any number of results from a single stream of elements, as shown by the following MQL query:

$$bag\{z \mid z \leftarrow unit[bag](\langle max, min\rangle\{\langle f(a), g(a)\rangle \mid a \leftarrow as\}), z.1 > max\{d \mid d \leftarrow ds\}\ \}$$ [7]

The Monoid $\langle max, min\rangle$ in the query stipulates that two partial results namely, $max(f(a))$ and $min(g(a))$, can be calculated concurrently for the inner comprehension, thus enabling the evaluation of two distinct results from a single element stream. This MQL query does have an OQL equivalent, although SQL is not capable of expressing this in one query. A

[7] This query form is used in the evaluation chapter, chapter 9.

less efficient way of performing this query is given below:

$$bag\{\ \langle z', min\{g(a)|a \leftarrow as, Ps(a)\ \}\rangle\ |z' \leftarrow unit[bag](max\{f(a)|a \leftarrow as, Ps(a)\ \}),\ z'.1 < max\{d|d \leftarrow ds\}\}$$

Note the way in which the selection of an element of a record is performed in MQL using the infix "." notation. In the original query, the first element of the record which comprises the result element z is denoted by $z.1$. Without this facility, the two partial results have to be evaluated independently, which results in the production of *two* singleton streams produced by two homomorphisms on the collection as.

6.5.3 An Example — average calculation

Consider the following SQL query which calculates the average of a number of integer values (representing the price of a cd):

```
SELECT avg(price)
FROM cds cd
WHERE cd.date_received < 19920000
```

The calculation of the average may be performed in MQL by first calculating the cardinality of the collection and the partial sum, as shown in the MQL form below:

$$\langle sum, sum\rangle\{\ \langle price(cd), 1\rangle\ |\ cd \leftarrow cds, date\_received(cd) > 19920000\}$$

An alternative way of performing this task is to define a Monoid, avg.

Def 6.5 $avg = (\mathbb{N} \times \mathbb{N}, \langle 0,0\rangle, m)$

where $m \stackrel{def}{=} \langle a,b\rangle\ m\ \langle c,d\rangle \rightarrow \langle a+c, b+d\rangle$

Given this Monoid, which is in the *CM* category, the SQL query that calculates the partial sum of the price of cds and the partial count may be rewritten as:

$$avg\{\langle price(cd), 1\rangle | cd \leftarrow cds,\ date\_received(cd) > 19920000\}$$

Buneman *et al.* [BF79] in the earlier language FQL define arithmetic operators which take a pair and evaluate to a singleton, wrapped in an applicative notation. In MQL, the evaluation of the mean from the partial sum and count may be achieved using the following form:

$$set\{\ \langle x.1/x.2\rangle\ |x \leftarrow unit[bag](avg\{\ \langle price(cd), 1\rangle\ |\ cd \leftarrow cds, date\_received(cd) > 19920000\})\}$$

6.6 Extensibility

The range of monoids that may be used in MHL and MQL queries is not restricted to the ones defined in Def 6.4. One of the innovative aspects of the approach advocated in this thesis is the model's data type extensibility properties. Incorporating facilities to extend built–in data types enables more natural modelling of the data types in an application. We shall consider some sample queries to illustrate this point. In these cases, the definition of a new Monoid will clarify the query in hand.

6.6.1 Grouping

Consider the task of nesting the results of a query, as performed by the `GROUP BY` construct in SQL. The following query groups all tracks according to the cd on which they appear.

```
SELECT track.name
FROM tracks track
GROUP BY track.cd_id
```

An entirely new Monoid that is not in MT that captures the behaviour of the grouping operation is now defined.[8]

The grouped Monoid

Consider the free Monoid $\mathcal{F} = collection(V \times \mathcal{N})$ that is defined over the Cartesian product of a set of values V and the Monoid $\mathcal{N}$, such that $\mathcal{F} = (C, \ominus, \emptyset)$, where C is the set of collections. Elements $c \in C$ are identified with (equivalence classes of) finite sequences of pairs $[(m_i, n_i) | 1 \leq i \leq k]$, where $k \geq 0$, and $m_i \in V,\ n_i \in \mathcal{N},\ 1 \leq i \leq k$.

The grouping map on C is the map $g : C \to C$ which, given for some $k \geq 0$

$$c \sim [(m_i, n_i) | 1 \leq i \leq k]^{\;9}$$

groups pairs by V–value, forming the aggregate of the associated $\mathcal{N}$–values on the Monoid $(\mathcal{N}, \star, e)$ for each V–value. The range of g is also the set of collections over $(V \times \mathcal{N})$. g is defined in such a way as to preserve the order of the occurrence of pairs in c; informally, $g(c)$ contains exactly one pair for each distinct V–value that occurs.

Def 6.6 The grouping map $g : C \to C$ is defined inductively on the length of representative sequences as follows:

(i) $c = \emptyset$, define $g(\emptyset) = \emptyset$;

(ii) $c \sim [(m_1, n_1)]$, $len(c) = 1$, define $g(c) = c$;

(iii) suppose $len(c) = (k+1) > 1$, say $c \sim [(m_i, n_i) | 1 \leq i \leq (k+1)]$, and set $c' \sim [(m_i, n_i) | 1 \leq i \leq k]$, $len(c') = k$.

Assume that $g(c') \sim [(\bar{m}_j, \bar{n}_j) | 1 \leq j \leq l]$ has been defined, where $1 \leq l \leq k$ and $[\bar{m}_j | 1 \leq j \leq l]$ are distinct. Then

either $m_{k+1} = \bar{m}_J$ for a unique $1 \leq J \leq l$

define $n'_j = \bar{n}_j,\ 1 \leq j \leq l,\ j \neq J$
$n'_J = \bar{n}_J \star n_{k+1}$
and $g(c) \sim [(\bar{m}_j, n'_j) | 1 \leq j \leq l]$

or $\forall\, 1 \leq j \leq l,\ m_{k+1} \neq \bar{m}_j$

[8] The mathematical specification which follows is in conjunction with Ken Moody and builds upon the algorithmic formulation in Appendix B by the author.

[9] This treatment uses general sequences, which are denoted by square brackets. The appropriate merge operation is $\ominus$.

define $\bar{m}_{l+1} = m_{k+1}$, $\bar{n}_{l+1} = n_{k+1}$
and $g(c) \sim [(\bar{m}_j, \bar{n}_j)|1 \leq j \leq (l+1)]$

Note that separating out the case in which $len(c') = 1$ is not strictly necessary, but it is included to clarify the exposition.

Theorem 6.1 $g : C \to C$ is well–defined for all collection types in the following sense:

(a) let $c \sim [(m_i, n_i)|1 \leq i \leq k]$, and $c' \sim [(m_{\rho(i)}, n_{\rho(i)})|1 \leq i \leq k]$ for some permutation $\rho \in \Sigma_n$, and $\mathcal{N}$ be commutative;

(b) let $c \sim [(m_i, n_i)|1 \leq i \leq k]$, where c' is represented by a sequence obtained by extending the sequence for c by repeating certain elements, and $\mathcal{N}$ be commutative and idempotent.

In either case, the representative sequence for $g(c')$ is a permutation of representative sequence for $g(c)$.

Let D be the subset of C consisting of those collections c which may be represented by a sequence of pairs $[(m_i, n_i)|1 \leq i \leq k]$ in which the V–values are distinct.

Theorem 6.2 The grouping map $g : C \to C$ has the following properties:

(a) $\forall\ c \in C;\ \ g(c) \in D$;

(b) $\forall\ c \in C,\ \ len(g(c)) \leq len(c)$;

(c) $c \in D \Rightarrow g(c) = c$.

This theorem leads to two corollaries:

Corollary 6.4 $g(C) = D$

Corollary 6.5 $\forall c \in C,\ g(g(c)) = g(c)$

The grouped Monoid The grouping map has now been defined on collections and it has been shown that $g(C) = D$. The grouping map, g, is now exhibited as a monoid homomorphism. In order to do this, it is necessary to identify the binary operation $\triangle$ on D with respect to which

$$g(c \ominus c') = g(c) \triangle g(c')$$

Suppose $d, d' \in D$. By Theorem 6.2 $g(d) = d$, $g(d') = d'$. Hence, it is necessary to define $d \triangle d' = g(d \ominus d')$, where d, d' are regarded as elements of C on the right hand side. This leads to the definition of the $grouped(V \times \mathcal{N})$ Monoid.

Def 6.7 The Monoid *grouped*$(V \times \mathcal{N})$ defined over V and $\mathcal{N}$ by grouping on V−values is $(D, \triangle, \emptyset)$, where Δ is as defined.

Theorem 6.3 $g : (C, \ominus, \emptyset) \to (D, \triangle, \emptyset)$ is a monoid homomorphism, called the grouping *homomorphism.*

Proof Suppose given $c, c' \in C$, required to prove that:

$$g(c \ominus c') = g(c) \triangle g(c') = g(g(c) \ominus g(c'))$$

First note that, by definition,

$$g(c \ominus c') = g(g(c) \ominus c')$$

It is sufficient to consider the proposition $P(k)$:

for all finite sequences c, c' , such that $len(c') < k$,
$g(g(c) \ominus c') = g(g(c) \ominus g(c'))$.

Certainly $P(1)$, since $len(c') = 0 \Rightarrow g(c') = [\,]$.
Now suppose that $P(k)$ is true for some $k \geq 1$, and given finite sequences c, c', such that $len(c') = k$. Write
$c' = c'' \ominus [(m_k, n_{k'})]$ with $len(c'') < k$.

$$\begin{array}{llll}
\textbf{LHS} = & g(g(c) \ominus c') & = g(g(c) \ominus (c'' \ominus [(m_k, n_{k'})])) & \\
 & & = g((g(c) \ominus c'') \ominus [(m_k, n_{k'})]) & \text{by associativity of } \ominus \\
 & & = g(g(g(c) \ominus c'') \ominus [(m_k, n_{k'})]) & \text{by definition of } g\text{, Def 6.6} \\
 & & = g(g(g(c) \ominus g(c'')) \ominus [(m_k, n_{k'})]) & \text{by inductive hypothesis} \\
 & & = g(g(c) \ominus g(c'') \ominus [(m_k, n_{k'})]) & \text{by definition of } g\text{, Def 6.6}
\end{array}$$

The key m_k occurs at most once in each of the $g(c)$ and $g(c'')$. A number of cases are hence considered.

Case 1. m_k does not occur in c''.

Hence certainly $g(c') = g(c'') \ominus [(m_k, n_{k'})]$, and the required result follows immediately.

Otherwise the key m_k occurs exactly once in c'', say in the pair $(m_k, n_{k''})$. Then $g(c')$ is derived from $g(c'')$ by replacing the pair $(m_k, n_{k''})$ with the pair $(m_k, n_{k''} \star n_{k'})$. The two sequences are identical in every other respect.

Case 2. m_k occurs once in c'', but does *not* occur in c.

In this situation $g(g(c) \ominus g(c'))$ is derived from $g(g(c) \ominus g(c''))$ by replacing the pair $(m_k, n_{k''})$ with pair $(m_k, n_{k''} \star n_{k'})$.

This is by definition $g(g(c) \ominus g(c'') \ominus [(m_k, n_{k'})])$.

Case 3. m_k occurs once in c'', and also once in c.

Suppose that the key m_k occurs in c as the pair (m_k, n_k). In this case $g(g(c) \ominus g(c'))$ is derived from $g(g(c) \ominus g(c''))$ by simply replacing the pair $(m_k, n_k \star n_{k''})$ with pair $(m_k, n_k \star (n_{k''} \star n_{k'}))$.

Further, $g(g(c) \ominus g(c'') \ominus [(m_k, n_{k'})])$ is derived from sequence $g(g(c) \ominus g(c''))$ by replacing the pair $(m_k, n_k \star n_{k''})$ with pair $(m_k, (n_k \star n_{k''}) \star n_{k'})$.

But $n_k \star (n_{k''} \star n_{k'}) = (n_k \star n_{k''}) \star n_{k'}$ by associativity of $\star$.

Hence

$$\begin{aligned} g(c \ominus c') &= g(g(c) \ominus g(c'') \ominus [(m_k, n_{k'})]) \text{ by inductive hypothesis} \\ &= g(g(c) \ominus g(c')) \text{ as required.} \end{aligned}$$

Corollary 6.6 $\triangle$ *is associative on D*

Proof

$(d_1 \triangle d_2) \triangle d_3$	
$= (g(d_1) \triangle g(d_2)) \triangle g(d_3)$	g is the identity on D
$= g((d_1 \ominus d_2) \ominus d_3)$	by theorem 6.3
$= g(d_1 \ominus (d_2 \ominus d_3))$	by associativity of $\ominus$
$= g(d_1) \triangle (g(d_2) \triangle g(d_3))$	by theorem 6.3
$= d_1 \triangle (d_2 \triangle d_3)$	g is the identity on D

The definition of *grouped*$(V \times \mathcal{N})$ involves the Monoid structure of $\mathcal{N}$ only; V serves merely as a set of values with equality testing. If $\mathcal{N}$ is a free Monoid, provided that a restriction is made to the category of commutative (/idempotent) monoids, then the *grouped*$(V \times \mathcal{N})$ Monoid is also free. In this case, grouping is essentially the same operation as *nest* in the nested relational model (hence, invertible by *unnest* — therefore, the universal mapping property is preserved).

On the other hand, in the non–commutative case the *grouped* monoid is *not* free even if $\mathcal{N}$ is free, as may be seen by considering an example. The lists represented by the finite sequences

$$[(a, [x]), (b, [x]), (a, [y]), (b, [y])]$$
$$[(a, [x]), (b, [x]), (b, [y]), (a, [y])]$$

are both grouped to form the list that is represented by

$$[(a,[x,y]),(b,[x,y])]$$

Note that the two initial sequences are distinct as lists, of course, but *not* as bags. Further, the notation of this example points the (inessential) difference between grouping and the *nest* operation of the nested relational model.

Having defined the $grouped(V \times \mathcal{N})$ monoid, it is now possible to formulate a query using the $grouped(\mathbb{N} \times bag(tracks))$[10] monoid to express a range of queries that involve nesting. The original grouping OQL query under consideration which nests the elements in the tracks database according to cd may now be expressed in terms of this monoid:

$$grouped(\mathbb{N} \times bag(tracks))\{\langle cd\_id(track), track\rangle | track \leftarrow tracks\}$$

Grouping with an additional aggregate

Consider the following OQL query which calculates the average cd price for each singer, for cds that the singer has been the main artist:

```
SELECT artist, average_price_per_cd:  avg(SELECT cd.price FROM partition)
FROM cds cd, cd.artists a
GROUP BY artist :  a.name
```

At first sight, the equivalent MQL formulation of this query using the corresponding $grouped(string \times avg)$ monoid would be:

$$set\{\ \langle y.1, y.2.2/y.2.1\rangle\ | y \leftarrow grouped(string \times avg)\{\langle artist, \langle price(cd), 1\rangle\rangle| \\ cd \leftarrow cds, artist \leftarrow artists(cd)\}\}$$

However, this MQL query is *ill-defined* as the partial result Monoid, $grouped(string \times avg)$ is not free. Further, there exists no suitable homomorphic function, h, such that $h\ :\ grouped(string \times avg) \rightarrow set(string \times \mathbb{R})$[11] since it is not possible to define a monoid homomorphism that is consistent with the formation of the quotient.[12] The lack of a suitable homomorphism means that the average calculation cannot be performed on the fly, which is intuitively the way things are!

Grouping with additional conditionals

In some cases, the results of a nesting operation are to be filtered according to properties of each equivalence set. For example, the following query in OQL selects the artist and total number of cds released by an artist where the total number of cds released is less than 10.

[10]Given a monoid $(\mathbb{N}, \cup, \{\{\ \}\})$, and the monoid $bag(tracks)$ as defined by $(bag(tracks), \uplus, \{\{\ \}\})$. Define $grouped(\mathbb{N} \times bag(tracks)) = (bag(\mathbb{N} \times bag(tracks)), \Delta, \{\{\ \}\})$.

[11]Where $\mathbb{R}$ denotes the set of real numbers.

[12]After all, if h is a homomorphism, then this would require that $\frac{a}{b} + \frac{a'}{b'} = \frac{a+a'}{b+b'}$!

```
SELECT artist, price_per_cd:  (SELECT COUNT(*) FROM partition)
FROM cds cd, cd.artists a
GROUP BY artist :  a.name
HAVING (SELECT COUNT(*) FROM partition) > 10
```

Again, this query cannot be formulated using MQL since the partial result which is produced before the application of the `HAVING` construct is the Monoid $grouped(string \times sum)$, which is not free.

6.6.2 Ordering

In some cases, it is necessary to order the results returned by a query. In order to achieve this, a new Monoid, *sort* may be defined that orders data elements in ascending order. Note that in this case it is necessary to work with lists rather than bags (where order does not matter, cf. SQL).

The sort Monoid

Consider the Monoid of lists of $(key, value)$ pairs, $L = list(K \times V)$. The aim is then to show a monoid homomorphism (in the non-commutative category) which corresponds to the sorting function $s : L \to L$.

As it is necessary to handle repeated keys, so that a *stable* sorting algorithm is required. Let S be the subset of L which consists of those lists that are non-descending by key.

Theorem 6.4 The sorting map $s : L \to L$ has the following properties:

a) $s(l)$ belongs to S for all lists l.

b) if l is a member of S, then $s(l) = l$.

This leads to two corollaries:

Corollary 6.7 $s(L) = S$.

Corollary 6.8 $\forall\, l \in L \;\; s(s(l)) = s(l)$.

The sort monoid The sorting map has now been defined on lists and it has been shown that $s(L) = S$. The sorting map s is now exhibited as a monoid homomorphism. In order to do this, it is necessary to identify the binary operation $\otimes$ on S with respect to which

$$s(s_1 \mathbin{@} s_2) = s(s_1) \otimes s(s_2)$$

Suppose that $s_1, s_2 \in S$. By Theorem 6.4 $s(s_1) = s_1$ and $s(s_2) = s_2$. Hence, it is necessary to define $s_1 \otimes s_2 = s(s_1 \mathbin{@} s_2)$, where s_1, s_2 are regarded as elements of L. This leads to the definition of the $sort(K \times V)$ Monoid.

Def 6.8 The Monoid $sort(K \times V)$ defined over K and V by sorting on K–values is the monoid $(S,\otimes,\emptyset)$ where:

$s_1 \otimes s_2 \ =$
    if $s_1 = [\,]$ then s_2
    else if $s_2 = [\,]$ then s_1

    else if $head(s1).key \leq head(s_2).key$

        then $[head(s_1)]$ @ $(tail(s1) \otimes s_2)$
        else $[head(s_2)]$ @ $(s_1 \otimes tail(s_2))$

Theorem 6.5 $(L, @, [\,]) \rightarrow (S, \otimes, \emptyset)$ is a monoid homomorphism called the sorting *homomorphism.*

Notice that the function $\otimes$ merges two already sorted lists. It is possible to define a sorting function in terms of this function:

Def 6.9
$i\_sort(l)$=
    if $l = [\,]$ then $[\,]$
    else $[head(l)] \otimes i\_sort(tail(l))$

Although this definition establishes an inefficient algorithm, it has the advantage of being easy to establish as correct by induction on the length of l; $\otimes$ can also be proved correct by induction on $len(s_1) + len(s_2)$, for example.

An improvement can be made to the algorithm providing that the length of a list is cheap to determine:

Def 6.10
Let

$left(l) =$
    if $len(l) \leq 2$ then $[\,]$
    else $sublist(l, 1, (len(l)/2))$

$right(l) =$
    if $l = [\,]$ then $[\,]$
    else $sublist(l, (len(l) + 2)/2, len(l))$

where the function $sublist(alist, start, finish)$ returns the sublist which is the result of the selection of elements from position $start$ to position $finish$ in the list $alist$. Using Def 6.10 together with the definition of $\otimes$, a new sorting function $merge\_sort$ is now given:

Def 6.11
$merge\_sort(l)$=
    if $len(l) \leq min\_size$ then $isort(l)$
    else $merge\_sort(left(l)) \otimes merge\_sort(right(l))$

This more efficient algorithm gives an efficient formulation for a parallel sort which can be easily implemented. It also admits a simple correctness proof, given that now both $\otimes$ and *i_sort* are correct.

An example The *sort(K × V)* Monoid may used in the following MQL query that orders the cds in the cds database that are distributed by EMI according to price:

$$sort(\mathbb{N} \times \mathbb{N})\{\ \langle price(cd), cd\_id(cd)\rangle \mid cd \leftarrow cds,\ distributor(cd) = \text{“}EMI\text{”}\ \}$$

This section has outlined the MQL and MHL languages and the way in which the queries in SQL and OQL may be modelled in these languages. The next section outlines a data parallel evaluation paradigm for the *hom* function.

6.7 Parallel Evaluation Model

This section is devoted to the parallel evaluation model for MHL. The MHL is based on a single operation form, the *Monoid homomorphism* (§ 6.3), which subsumes the so–called *sru* form of structural recursion (§ 2.4.4 and [TBW92, TS91, BNTW95]). The evaluation of a single Monoid homomorphism over a free Monoid (collection) may be naturally divided into two distinct stages: a binary tree partitioning phase during which the concrete collection is partitioned into single element collections; and a combining phase during which the result of the homomorphism is evaluated by pairwise *merge* operations. This is termed a unified dividing tree and a combining tree, as depicted in figure 6.7. However, for greater flexibility these two phases have been separated into the distinct tasks of division of the argument collection, and combination of partial results produced by independent parallel evaluations. These are modelled by the `split` (§ 5.1.4) and `hom` operators respectively. This chapter focuses on the `hom` function, which performs the combining phase.

Given that in realistic situations the depth of an evaluation tree is capped due to resource limitations, the granularity of the independent parallel evaluations which constitute the leaves of the evaluation tree may be at the level of sub–collections, and not individual elements. The definition of a monoid homomorphism (§ 6.3) exhibits a natural *divide–and–conquer* behaviour, which although suitable for task division, is not ideal for sequential evaluation. However, there exists an efficient translation from the *sru* formulation to the element–at–a–time loop–based iterative implementation, *sri* (§ 2.4.4):

$$sru(u, f, e)(O) \stackrel{def}{=} sri(\lambda(x, y).u(f(x), y), e)(O)$$

This translation is also applicable to the monoid homomorphism form used in MHL and may be used at the sub–collection level. To see why this form offers greater efficiency for sequential evaluation compared to the divide–and–conquer form prescribed by the definition of the monoid homomorphism, consider an application of $\mathrm{sru}(\cup,\ \lambda\ x.insert((x+1,\{\}),\{\})$. to a set of integers, $\{3, 5, 1, 2\}$:

$$\cup(\cup(\{4\}, \{6\}), \cup(\{2\}, \{3\}))$$

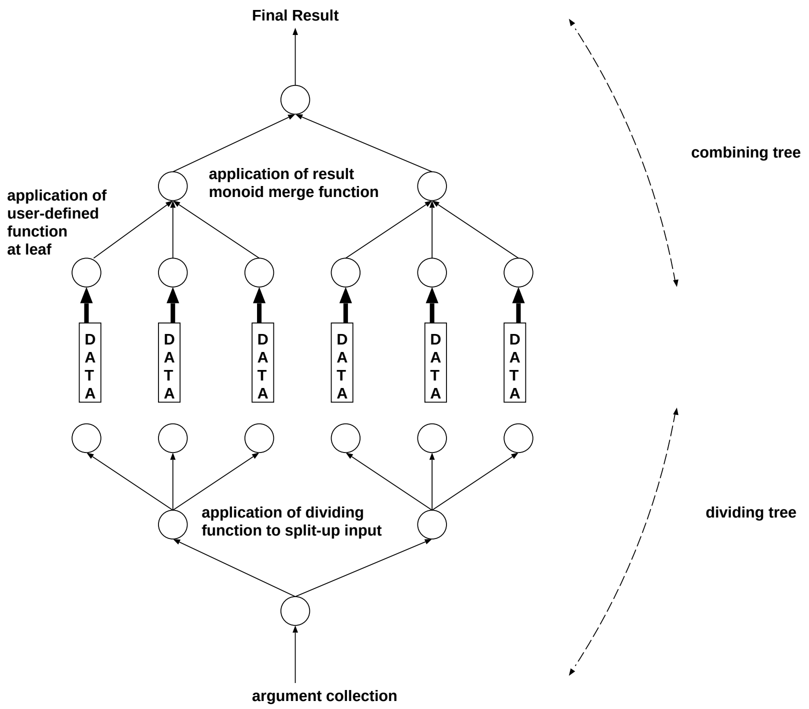


Figure 6.1: Homomorphism Evaluation

where the function *insert* takes an element, adds 1 to it, and returns the result as a singleton set. This formulation is equivalent to an application of $sri(\lambda(x,y).(\cup)\ (insert(x+1,\{\}),y),\{\})$ to the argument set, giving:

$$\lambda(x,y).\cup(insert(x+1,\{\}),y)(3,\lambda(x,y).\cup(insert(x+1,\{\}),y)(5,\lambda(x,y).\cup(insert(x+1,\{\}),y)$$
$$(1,\lambda(x,y).\cup(insert(x+1,\{\}),y)(2,\{\}))))$$

which through a process of β–reduction on the definition of *insert* gives:

$$\cup(\{4\},\cup(\{6\},\cup(\{2\})\cup(\{3\},\{\})))$$

Note that this form, in a similar way to the *foldr* (fold right) function in functional programs, leads to intermediate forms before the final result is evaluated. Since all monoids have an *associative* merge function, the evaluation can proceed from right to left starting with an initial accumulator (the *zero* of the Monoid) in a manner similar to *foldl* (left fold). Such an evaluation plan is more space–efficient since the production of intermediate forms may be dispensed with. Furthermore, a *rolling* partial result is produced from the start of the evaluation, which facilitates pipelining. Hence, the insertion based representation is adopted for the sequential evaluation phase[13] on sub–collections in preference to the divide–and–conquer union representation.

Now consider the task of combining the partial results of independent parallel evaluations. Note that the same iterative, element-at-a-time philosophy is not efficient, whereas the

[13] Which run in parallel!

use of a collective *merge* function is efficient. Hence, for the phase which involves the combination of partial results, the *merge* function of the result Monoid is used in preference to its *element–at–a–time* form.[14] Note that if the result monoid is in the CM category, then partial results from different sources can be combined as soon as they become available, and in any order, thereby delivering a higher degree of potential parallelism.

6.8 Query Optimisation

Many optimisation systems for relational and object-oriented query languages transform a query into an expression of a logical algebra in which equivalences are defined. The optimiser uses the equivalences between different algebraic forms to transform expressions. In order to guide the rewrite process, rule based optimisers heuristically prescribe the direction and application of each transformation. The implementation of a logical algebra operator may be realised in a number of different ways. Hence, in addition to algebraic equivalences, a number of implementation–based optimisations are also normally applied during the query optimisation process.

Many optimisations which are currently carried out in database systems are of a somewhat ad–hoc nature and we know of none of which have been formally proven correct. A major aspect of this model presented in this thesis is the possibility of formal verification of common optimisations used in many DBMSs. The homomorphism rewrite database includes a set of rewrite rules many of which have been proven correct (Appendix C). The rewrite rules presented can be categorised into two: algebraic equivalences which do not directly lead to greater operational efficiency but are intermediate forms for other optimisations, and those that reduce the overall complexity of the query (and hence afford a more efficient implementation). The techniques used to achieve more efficient evaluations are based upon the dual reduction of both collection scan operations and partial result production within a pipelined evaluation paradigm.

The approach taken in this thesis differs from other related work which considers equivalences on a number of stereotyped higher–order functions, such as `filter`, `map` and `reduce`, on a single collection type (list for example, in the case of [Fok91]). In contrast, the query framework in this thesis considers the *homomorphism*. Work within the area of unnesting relational queries (Kim *et al.* [Kim82], Dayal *et al.* [DMB+87], and work from Murray [Mur93]) are also utilised in the optimisation process. However, in contrast to these approaches, the overall approach here is based upon monoids and equivalences between Monoid comprehension and homomorphism expressions. Furthermore, data placement information is not directly used as part of the algebraic rewrite process — data partitioning/repartitioning requirements are realised after the query rewrite phase.

The following sections show the way in which proofs of these equivalences may be carried out.

[14] Contrast this with the *insert* function in *sri*.

6.8.1 Example Proofs

Ax 7

Consider the identity given below:

$$\mathcal{M}_1\{\ r_1 \mid v_1 \leftarrow \mathcal{M}_2\{r_2|v_2 \leftarrow M_i(\alpha), P_1^*(v_2)\ \},\ P_2(v_1)\ \} \equiv$$
$$\mathcal{M}_1\{\ r_1^{r_2(v_2)/v_1} \mid v_2 \leftarrow M_i(\alpha), P_1^*(v_2),\ P_2^{r_2(v_2)/v_1}(v_1)\}$$

---

where terms of the form $t^{a/b}$ means replace all free occurrences of b in t with a. The predicates P_1, P_2 are optional, and if the filter $P_1(v_1)$ is a homomorphism defined over $\mathbb{B}$ then the range of the homomorphism is in the *CIM* category. This rule is restricted to cases in which r_2 does not contain comprehension terms since if the term r_2 is a comprehension term, then the cost of calculating this value a number of times for both r_1 and P_2 will reduce the efficiency of this transformation. A guide to the notation used in the presentation of identities, such as the one given above, can be found in Appendix C.

This rule may be applied to the following MQL expression which finds all cds recorded by Sting that cost less than £15:

$$bag\{cd|cd \leftarrow bag\{y|y \leftarrow cds, artist(y) = \text{“}Sting\text{”}\},\ price(cd) < 15\}$$

The proof for this identity is given.

Proof Ax 7 Required to prove that:

$$\forall\mathcal{M}_i\quad hom[\mathcal{M}_2,\mathcal{M}_1]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_1](r_1)\ else\ zero[\mathcal{M}_1])\ A\ \equiv$$
$$hom[\mathcal{M}_i,\mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_2)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ zero[\mathcal{M}_1])\ \ \mathcal{M}_i(\alpha)$$

where

$$A = hom[\mathcal{M}_i,\mathcal{M}_2]\ \lambda\, v_2(if\ P_1(v_2)\ then\ unit[\mathcal{M}_2](r_2)\ else\ zero[\mathcal{M}_2]\ )\mathcal{M}_i(\alpha)$$

By Induction on $\mathcal{M}_i$.

- $\mathcal{M}_i = zero[\mathcal{M}_i]$

LHS

$hom[\mathcal{M}_2,\mathcal{M}_1]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_1](r_1)\ else\ zero[\mathcal{M}_1])\ hom[\mathcal{M}_i,\mathcal{M}_2]\ \lambda\, v_2(if\ P_1(v_2)$ $then\ unit[\mathcal{M}_2](r_2)\ else\ zero[\mathcal{M}_2]\ )\ zero[\mathcal{M}_i]$

β reduction

$zero[\mathcal{M}_1]$

RHS

$hom[\mathcal{M}_i,\mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_2)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ zero[\mathcal{M}_1])\ zero[\mathcal{M}_i]$

β reduction

$zero[\mathcal{M}_1]$

$\diamond$

- $\mathcal{M}_i = unit[\mathcal{M}_i](b)$

LHS

$hom[\mathcal{M}_2,\mathcal{M}_1]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_1](r_1)\ else\ zero[\mathcal{M}_1])\ \ hom[\mathcal{M}_i,\mathcal{M}_2]\ \lambda\, v_2\ k\ unit[\mathcal{M}_i](b)$

where $k = if\ P_1(v_2)\ then\ unit[\mathcal{M}_2](r_2(v_2))\ else\ zero[\mathcal{M}_2]$

β reduction

$hom[\mathcal{M}_2,\mathcal{M}_1]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_2](r_1)\ else\ zero[\mathcal{M}_2])$
$hom[\mathcal{M}_i,\mathcal{M}_2]\ \lambda\, v_2(if\ P_1(b)\ then\ unit[\mathcal{M}_2](b)\ else\ zero[\mathcal{M}_2]\ )$

case $P_1(b)$:

- *false*
  $hom[\mathcal{M}_1,\mathcal{M}_2]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_2](r_1)\ else\ zero[\mathcal{M}_2])\ zero[\mathcal{M}_i]$
  β reduction
  $zero[\mathcal{M}_1]$
- *true*
  $hom[\mathcal{M}_1,\mathcal{M}_2]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_2](r_1)\ else\ zero[\mathcal{M}_2])unit[\mathcal{M}_2](b)$
  $hom[\mathcal{M}_1,\mathcal{M}_2]\lambda\, v_1\ if\ P_2^{r_2(b)/v_1}(v_1)\ then\ unit[\mathcal{M}_j](r_1^{r_2(b)/v_1})\ else\ zero[\mathcal{M}_1]$
  case $P_2^{r_2(b)/v_1}(v_1)$
  * *false*
    $zero[\mathcal{M}_1]$
  * *true*
    $unit[\mathcal{M}_1](r_1^{r_2(b)/v_1})$

  $\diamond$

RHS

$hom[\mathcal{M}_i,\mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_1)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ \ zero[\mathcal{M}_1]$
$else\ zero[\mathcal{M}_1])\ unit[\mathcal{M}_i](b)$

β reduction

$if\ P_1(b)\ then\ if P_2^{r_2(b)/v_1}(v_1)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(b)/v_1})\ else\ zero[\mathcal{M}_1]$

case $P_2^{r_2(b)/v_1}(v_1)$:

- *false*
  $zero[\mathcal{M}_1]$
- *true*
  $unit[\mathcal{M}_1](r_1^{r_2(b)/v_1})$

$\diamond$

- $\mathcal{M}_i\ =\ merge[\mathcal{M}_i](M_1, M_2)$

LHS

$hom[\mathcal{M}_2,\mathcal{M}_1]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_j](r_1)\ else\ zero[\mathcal{M}_i])\ hom[\mathcal{M}_i,\mathcal{M}_2]\ \lambda\, v_2(if\ P_1(v_2)$
$then\ unit[\mathcal{M}_2](v_2)\ else\ zero[\mathcal{M}_2]\ )\ \ merge[\mathcal{M}_i](M_1, M_2)$

β reduction

$merge[\mathcal{M}_i](hom[\mathcal{M}_2,\mathcal{M}_1]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_j](r_1)\ else\ zero[\mathcal{M}_1])\ hom[\mathcal{M}_i,\mathcal{M}_2]\ \lambda\, v_2(if\ P_1(v_2)$
$then\ unit[\mathcal{M}_2](v_2)\ else\ zero[\mathcal{M}_2])M_1, hom[\mathcal{M}_2,\mathcal{M}_1]\lambda\, v_1.(if\ P_2(v_1)\ then\ unit[\mathcal{M}_j](r_1)\ else\ zero[\mathcal{M}_1])$
$hom[\mathcal{M}_i,\mathcal{M}_2]\ \lambda\, v_2(if\ P_1(v_2)\ then\ unit[\mathcal{M}_2](v_2)\ else\ zero[\mathcal{M}_2])M_2)$

Inductive Hypothesis

$merge[\mathcal{M}_i](\ hom[\mathcal{M}_i,\mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_1)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ zero[\mathcal{M}_1])\ \mathcal{M}_1$
$hom[\mathcal{M}_i,\mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_1)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ zero[\mathcal{M}_1])\ \mathcal{M}_1\ )$

RHS

$hom[\mathcal{M}_i, \mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_1)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ zero[\mathcal{M}_1])$
$merge[\mathcal{M}_1](M_1, M_2)$
β reduction
$merge[\mathcal{M}_i](\ hom[\mathcal{M}_i, \mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_1)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ zero[\mathcal{M}_1])\ \mathcal{M}_1$
$hom[\mathcal{M}_i, \mathcal{M}_1]\lambda\, v_2.(if\ P_1(v_2)\ then\ if P_2^{r_2(v_2)/v_1}(v_1)\ \ then\ unit[\mathcal{M}_1](r_1^{r_2(v_2)/v_1})\ else\ zero[\mathcal{M}_1])\ \mathcal{M}_1\ )$
$\diamond$

$\square$

Ax 9

Ax 9 produces a partial result in a homomorphism expression in which there are two or more homomorphisms. The MQL formulation for this optimisation is given below:

$$\mathcal{M}_1\{\ r \mid CT_1,\ \ v_i \leftarrow \mathcal{M}_i(\alpha),\ P_1^+(v_i),\ CT_2\ \} \equiv$$

$$\mathcal{M}_1\{\ r \mid CT_1,\ v_i \leftarrow \mathcal{M}_i\{\ v_{ii} \mid v_{ii} \leftarrow \mathcal{M}_i(\alpha),\ P_1^+(v_i)^{v_{ii}/v_i}\ \},\ CT_2\ \}$$

where CT_1 contains at least one generation (i.e. is not null), and CT_2 contains no predicates solely on variable v_i. This optimisation may be applied to the query form given below:

$$bag\{\langle distributor(cd), length(track)\rangle | cd \leftarrow cds, disk\_no = 2,\ track \leftarrow tracks,$$
$$date\_recorded < 19940000, cd\_id(track) = cd\_id(cd)\}$$

The proof for this axiom follows.

Proof Ax 9 Required to prove that:
$\forall M_i\ \ hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])\mathcal{M}_i(\alpha)\ \ \equiv$
$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ \ (CT_2\ )\ hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})\ \ else\ zero[\mathcal{M}_i])\ M_i(\alpha)$
where CT_2 contains no variables which are solely on the variable v_i, i.e. v_i is free in CT_2.
By Induction on $M_i(\alpha)$

- $\mathcal{M}_i(\alpha) = zero[\mathcal{M}_i]$

LHS

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])(zero[\mathcal{M}_i])$
β reduction
$zero[\mathcal{M}_i]$

RHS

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ \ (CT_2\ )\ hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})\ \ else\ zero[\mathcal{M}_i])$
$zero[M_i]$

β reduction

$zero[\mathcal{M}_i]$

$\diamond$

- $\mathcal{M}_i(\alpha) = unit[\mathcal{M}_i](b)$

LHS

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])\ \ unit[\mathcal{M}_i](b)$

β reduction

$if\ P_1(b)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i]$

case $P_1(b)$:

- *false*
  $zero[\mathcal{M}_i]$
- *true*
  CT_2

RHS

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ \ (CT_2\ )\ hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})\ \ else\ zero[\mathcal{M}_i])$
$unit[M_i](b)$

β reduction

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ \ (CT_2\ )\ (if\ \ P_1^{b/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})\ \ else\ zero[\mathcal{M}_i])$

case $P_1^{b/v_i}(v_i)$:

- *false*
  $hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ \ (CT_2\ )zero[\mathcal{M}_i]$
  β reduction
  $zero[\mathcal{M}_i]$
- *true*
  $hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ \ (CT_2\ )\ unit[\mathcal{M}_i](b)$
  β reduction
  $hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ \ (CT_2\ )\ unit[\mathcal{M}_i](b)$
  β reduction
  CT_2

$\diamond$

- $\mathcal{M}_i(\alpha)\ =\ merge[\mathcal{M}_i](M_1, M_2)$

LHS

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])\ merge[\mathcal{M}_i](M_1, M_2)$

β reduction

$merge[\mathcal{M}_i](hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])\ M_1,$
$\quad hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\ v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])\ M_2)$

RHS

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ \ (CT_2\ )\ hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})\ \ else\ zero[\mathcal{M}_i])$
$\quad merge[\mathcal{M}_i](M_1, M_2)$

β reduction

$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ \ (CT_2\ )\ merge[\mathcal{M}_i](hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})$
$\quad else\ zero[\mathcal{M}_i])\ M_1,$
$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ \ (CT_2\ )\ hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})\ \ else\ zero[\mathcal{M}_i])\ M_2)$

β reduction (on outer *hom*)

$merge[\mathcal{M}_1](hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ \ (CT_2\ )\ hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})$
$\quad else\ zero[\mathcal{M}_i])\ M_1,$
$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ \ (CT_2\ )\ hom[\mathcal{M}_i, \mathcal{M}_i]\ \lambda v_{ii}\ (if\ \ P_1^{v_{ii}/v_i}(v_i)\ \ then\ unit[\mathcal{M}_i](v_{ii})\ \ else\ zero[\mathcal{M}_i])\ M_2)$

Inductive Hypothesis:

$merge[\mathcal{M}_i](hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])\ M_1,$
$hom[\mathcal{M}_i, \mathcal{M}_1]\ \lambda\, v_i\ (if\ P_1(v_1)\ \ then\ CT_2\ \ else\ zero[\mathcal{M}_i])\ M_2)$

$\diamond$

$\square$

This section has shown proofs for some optimisations that are part of the homomorphism rewrite database. These proofs have been based on the algebraic properties of the queries as they appear in MHL.

6.9 Summary

This chapter has introduced the theoretical basis of the query model. Collection and primitive types are modelled as monoids with a single iterator, the *monoid homomorphism*, as the sole means by which processing is carried out on these data types. A high–level notation based on monoid comprehensions is introduced in the form of the *Monoid Query Language* (MQL), which is a declarative form of the lower-level language the *Monoid Homomorphism Language* (MHL). The (language) model is data model independent and may hence be used to model both relational and object–oriented queries.

The paradigm of *monoid programming* has been advocated in this chapter as a basis for defining and modelling queries in data–intensive applications. In the same way that functional programming is based on functions between structures, monoid programming is based on structure-preserving functions (homomorphisms). The checks that are required in order to validate whether a Monoid to Monoid function is a homomorphism are, in general, undecidable. Instead of requiring the programmer to define homomorphisms, it is possible to restrict the domain of Monoid to Monoid functions to *free* monoids instead. By the universal mapping property, the programmer is then merely charged with writing a set function from the underlying set of the free Monoid to the result Monoid. This set function uniquely determines a homomorphism which can be generated in an automated fashion. This approach eliminates the need for equational checks and hence eases the task of the programmer. This forms the basis of the query model for MQL and MHL.

Although programming using homomorphisms from free Monoids to Monoids is adequate for most purposes, in some cases it is necessary to define monoids in addition to those defined by MT. Data type extensibility can be realised through facilities for the definition of suitable monoids. Suitable monoids at the language level have been defined to facilitate evaluations performing both *grouping* and *sorting* operations. A newly defined monoid may be used as the range of a monoid homomorphism without the imposition of any requirements. However, the domain of a homomorphism is restricted to a free Monoid. Hence, a newly defined Monoid has to be proven to be free before it can be used as the domain of a monoid homomorphism. An alternative to proving the freeness of the domain Monoid is to define a candidate homomorphism from the domain Monoid to the result Monoid. This homomorphism may be used provided that it can be proved that the function is a monoid homomorphism. Once a function has been proved to be a homomorphism, then the function definition may be stored in a dictionary of user–defined homomorphisms for future use. Alternatively, a compiler directive may be given by the programmer to guarantee that a particular function *is* a homomorphism. Even though the process of defining new Monoid to Monoid functions may seem cumbersome, the limited expressiveness of query languages such as SQL and OQL means that the definition of only a small number of Monoid to Monoid functions is required. Since all the potential argument and result monoids are known in advance, only a limited number of proofs are necessary in order to facilitate the modelling, in MQL/MHL, of *all* queries expressed in a database query language such as SQL. Further, the process of defining new functions and showing that they are homomorphisms may be carried out once–and–for–all for each new query language that is modelled in MQL/MHL.

The next chapter introduces the compiler architecture for the parallel machine.

Chapter 7

Compilation Process

An implementation of the complete model for decision–support queries has been carried out on a parallel machine. The implementation consists of two distinct components: the front-end compiler tools which carry out code translation, type checking and optimisation; and, the back-end parallel machine runtime environment that executes the object code produced by the compilation tools. This chapter presents the front-end compiler architecture. The back-end architecture is presented in chapter 8.

7.1 Interaction Model — Outline

The individual components of the interaction model for decision support queries (figure 7.1), and the information sources are outlined below:

- **Functional Components**
  - *SQL to MQL translator*
  - *MQL to MHL translator*
  - *MHL Rewrite System*
  - *Parallel Type Checker* (*PTC*)
  - *Object Code Generator*
- **Information Sources**
  - *Monoid Definition Dictionary*
  - *Data Partition Dictionary*
  - *Monoid Instance Dictionary*
  - *Element Dictionary*
  - *Monoid Homomorphism Rewrite Database*
  - *Symbol Table*

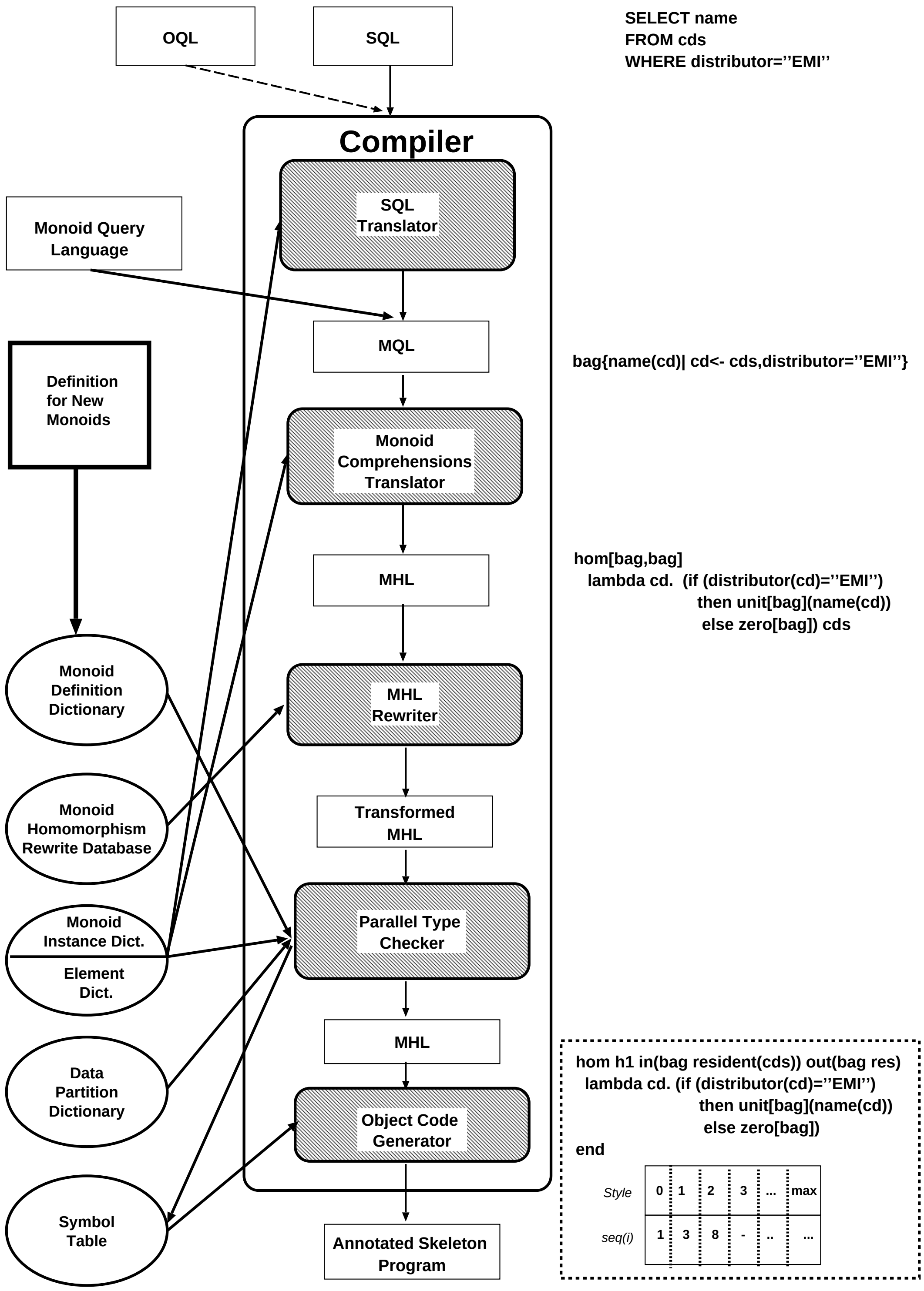


Figure 7.1: Decision–Support Interaction Model

7.1.1 Summary

This chapter presents the machine independent compilation tools that process MQL queries to produce annotated skeleton code suitable for execution on the runtime software on the parallel machine. The input to the compilation process is a query in MQL; an SQL to MQL translator has also been implemented. Source MQL is translated into MHL that is input into the MHL rewriter and then into the PTC. The PTC is the main component of the front–end compilation tools. It takes an MHL form and generates a symbol table in which predicates are grouped according to the variables that they use. Each occurrence of a variable in a predicate is substituted by the VSL formulation for the predicate which is then normalised. The subsequent resolution phase of the PTC labels each symbol table record with an annotation that determines the data partitioning requirements of the query. From this, the object code generator realises the data partitioning requirements for each record in the symbol table taking into account the partitioning on resident collections in realising an implementation for the query. The resultant skeleton code then undergoes a resource allocation process in which processing resources are allocated to the evaluation of each skeleton within the overall program. The output of the object code generator is an annotated skeleton program (ASP) that is suitable for evaluation on the parallel machine environment.

The next section discusses the contents and purpose of each of the first five information sources given at the start of § 7.1. Discussion of the nature and role of the symbol table is delayed until § 7.9.2. The rest of this chapter elaborates upon the functionality of each of the functional components and outlines the role of the information sources in the overall compilation process.

7.2 Monoid Definition Dictionary

The *monoid definition dictionary* is a repository which contains definitions for monoids. The definition for each Monoid determines the base type of the Monoid, and the implementation of the zero and merge functions, and the generator[1] in the case of a free Monoid. The zero and *merge* functions for a ground Monoid are based on built-in systems types *natural number*, *string* and *boolean* and hence have concrete implementations. However, the implementation of these functions for a collection is independent of the type of the elements of the collection. In the current implementation the Monoid functions are implemented in a procedural language and compiled as libraries. When a new Monoid needs to be added, the definition for the new Monoid may be added to the monoid definition dictionary. The front–end compilation tools set the necessary flags to inform the back–end evaluator of the update to the monoid definition dictionary. At evaluation time, the parallel machine master process (§ 8.1.1) loads the new library functions and broadcasts them to each of the workers. These functions may then be dynamically linked into the runtime system of each worker on the parallel machine and used during the evaluation phase.

[1]Only the injection into the free Monoid from the underlying set is recorded.

7.3 Data Partition Dictionary

The *data partition dictionary* holds information pertaining to the *physical* data distribution of persistent collections across the nodes of the parallel machine. The data partition dictionary holds, for each collection, one or more records that contain the following information, for each different logical partition:

de–clustering An array with equal cardinality to the number of physical nodes on the parallel system where the contents of each location of the array, i, determine whether a partition of the collection is resident at the corresponding node, i. This representation allows cheap checks to be carried out to ascertain whether two collections are de–clustered, partitioned and assigned in precisely the same way. For any two collections, an *exclusive OR* operation over the two arrays gives an array of zeros if the partitioning of the two collections is fully–overlapped. The number of nodes across which data has been distributed is given by the total number of 1s in the array.

partitioning Data elements may be distributed using round–robin, hash or range partitioning where a set of attributes is defined as the partitioning key and their values in each element are used for hash or range partitioning. *Duplicated* partitioning, which is defined as the case in which the collection is replicated across *all* nodes, is also available.

assignment Assignment information is extracted from the array. For hash and range based partitioning, assignment information appears as a list of predicates each of which is in normalised VSL notation (§ 7.9.2).

The two main persistent collections[2] *cds* and *tracks* are fully de–clustered and fully overlapped on the cd_id attribute for the former, and both the cd_id, track_id attributes for the latter, using range–based partitioning. This decision was motivated by the desire to support equijoin operations on the two collections using the common cd_id field, and nesting of tracks according to cd_id.

7.4 Monoid Dictionaries

The cell runtime system requires information on the persistent collections in the database. The required information is available in two dictionaries:

- *Monoid Instance Dictionary* — The monoid instance dictionary holds information on the types of each collection. No reference is made to the internal structure of the elements;
- *Element Dictionary* — The element dictionary holds information on the type and structure of the individual elements that comprise each collection. The type of each attribute is also recorded.

[2]Recall that cds and tracks are modelled as bag collections, $bag(cds)$, $(bag(cds), \uplus, \{\{\}\})$ and $bag(tracks)$, $(bag(tracks), \uplus, \{\{\}\})$ respectively.

This information is used to update the information kept by the worker processes on the parallel machine.

7.5 Monoid Homomorphism Rewrite Database

The *monoid homomorphism database* is a repository of source MHL to source MHL transformations, as outlined in Appendix C. These transformations are in the form of rules in *Standard ML* (SML) and are used by the *MHL Rewriter* during the optimisation process.

The next section discusses the functional components of the interaction model.

7.6 SQL to MQL

An SQL [fSIPS89] front–end which interfaces to the MQL to MHL translator has been implemented[3] as part of this prototype. This enables existing SQL queries from the cd database to be used in the evaluation of the system.

The SQL standards [fSIPS89, fSIPS92] do not formally define the semantics of SQL queries. Hence it is difficult to prove the correctness of SQL queries, or equivalence between SQL queries and those expressed in another language such as MQL. This makes it hard to translate SQL into optimised MQL. Although IBM originally based SQL on the Relational Theory defined by Codd [Cod70], SQL is more than relationally complete with features such as grouping, aggregate functions and NULL values, which do not have an equivalent in the relational model. Attempts have been made with limited success to specify SQL formally through a number of extensions to the relational algebra and calculus [CG85, vB89, NPS91, Mur92].

SQL queries are modelled as having the general form.

```
SELECT [DISTINCT] {expression [,expression]...} {*}
FROM table-name [alias] [,table-name [alias]]...
[WHERE search-condition]
[GROUP BY column-name [,column-name]... ]
[HAVING search-condition]
[ORDER BY column-name [DESC] [,column-name [DESC]]...]
```

which is represented using the following abstract syntax, using regular expression notation:

```
SELECT      value exp {,value exp}*          <selection list>
FROM        table ref {,table ref}*          <from clause>
{WHERE      search condition}?               <where clause>
{GROUP BY   column specification}            <group by clause>
            {, column specification}* }?
{HAVING     search condition}?               <having clause>
```

[3]Note that this thesis is concerned with the Data Manipulation Side (DML) of SQL and within this, only the *query specification*, sometimes referred to as the SQL *select statement* or *query block*.

Table 7.1 gives some common SQL forms and their MQL equivalents.

An example of an SQL query and its MQL equivalent are presented below.

```
SELECT      MIN(cd.price)
FROM        cds cd
WHERE       cd.total_duration > 60
```

$min\{price(v2)|v2 \leftarrow cds, total\_duration(v2) > 60\}$

SQL	***MQL***	***Comment***
<*select list*>		
SELECT V $v_i.e_i, \ldots, v_n.e_n$	$bag\{\langle e_i(v_i), \ldots, e_n(v_n)\rangle \| \ldots\}$	
SELECT DISTINCT $v_i.e_i, \ldots, v_n.e_n$	$set\{\langle e_i(v_i), \ldots, e_n(v_n)\rangle \| \ldots\}$	
SELECT *agg_op* $v_i.e_i$	$agg\_op\{e_i(v_i) \| \ldots\}$	where *agg_op* is MAX/ MIN/ SUM/ COUNT
SELECT *agg_op*(DISTINCT $v_i.e_i$)	$agg\_op\{v_k \| v_k \leftarrow set\{e_i(v_i) \| \ldots \| \ldots\}$	where *agg_op* is MAX/ MIN/ SUM/ COUNT
SELECT COUNT(*)	$sum\{1 \| v_1 \leftarrow \ldots\}$	
<*from clause*>		
FROM $V_i\ v_i, \ldots, V_n\ v_n$	$v_i \leftarrow V_i, \ldots, v_n \leftarrow V_n$	
<*where clause*>		
WHERE $\langle v_i.att_i\ relop\ lit_j, \ldots$	$att_i(v_i)\ relop\ lit_j, \ldots$	where lit_j is a literal
WHERE $\langle v_i.att_i\ relop\ v_j.att_j, \ldots$	$att_i(v_i)\ relop\ att_i(v_j), \ldots$	
WHERE (... OR ...)	$merge[some](\ldots, \ldots)$	
WHERE $v_i.e_i$ IN (*sql_query*)	$\mathcal{M}\{\ldots \| \ldots, some\{e_i(v_i) = \ldots \| \ldots\}\}$	
WHERE EXISTS (*sql_query*)	$\mathcal{M}\{\ldots \| \ldots, some\{single\_predicate \| \ldots\}\}$	
WHERE $v_i.e_i$ *relop b_term* (*sql_query*)	$\mathcal{M}\{\ldots \| \ldots, b\_term\{e_i(v_i)\ relop\ v_j \| \ldots\}\}$	where *b_term* is ALL/ SOME
<*grouping clause*>		
GROUP BY $v_i.e_i$	$grouped(\mathcal{M}_1 \times \mathcal{M}_2)\{\langle e_i(v_i), \ldots\rangle \| ..\}$	
<*having clause*>		
GROUP BY $v_i.e_i$ HAVING <*search condition*>[4]	if free(M_2) then (hom $[grouped(\mathcal{M}_1 \times \mathcal{M}_2), \mathcal{M}]$ $\lambda v_j\ (\ldots))\ grouped(\mathcal{M}_1 \times \mathcal{M}_2)$ $\{\langle e_i(v_i), \ldots\rangle \| \ldots\}$	
ordering clause		
ORDER BY $v_i.e_i$ *ord*	$sort\{\langle e_i(v_i), \ldots\rangle \| \ldots\}$	where *ord* is ASC. For the DESC case, a suitable Monoid would have to be defined.

Table 7.1: SQL and equivalent MQL forms

[4]A restricted form of the search condition.

7.7 MQL to MHL

MQL is the high–level declarative language interface for decision–support queries. MQL queries are translated into MHL using the translation algorithm that is based upon the definition for a Monoid comprehension given in table 7.2. The translation process is relatively simple since there is no mismatch between the semantics and the operational models of the two languages owing to the fact that MQL is just a high–level notation for MHL.

In addition to semantic type checking[5] there are two additional issues that need to be resolved in order to ensure correct parallel evaluation of MHL expressions:

- *transition checking* which has to be performed on each homomorphism in relation to the category of the argument and result Monoid.
- *data placement checking* that determines the required data placement for the evaluation of an expression.

Both of these issues are handled by PTC which is discussed in § 7.9.

7.8 MHL Rewrite System

The MHL Rewrite System has been implemented as a source to source rule–based optimiser which uses the rules in the homomorphism rewrite database. Many of the rewrite rules are MHL formulations of existing optimisations on relational and object–oriented databases. This shows the applicability of existing optimisation schemes to the query model outlined in this thesis. The rules are based on MHL expressions and may be categorised into:

Identities A number of identities are presented that do not reduce the cost of the evaluation of the query with respect to the unoptimised version. These identities serve as intermediate forms that facilitate the application of the algebraic and implementation–based optimisations.

Algebraic Optimisations Algebraic optimisations constitute the major part of the rule–base. The rules are obtained by orienting equational axioms of MHL in a manner that reduces the amount of intermediate data generated within queries. Standard optimisation techniques that have been captured include *filter promotion*, *loop fusion* and *qualification of invariant code* as have a number of optimisations that enable a more pipelined implementation. The correctness of a number of these rules has been proven formally, as outlined in § 6.8.1 and § 6.8.1. Algebraic optimisations do not operate on the usual algebra of relational operators, such as the filter selection, projection and joins, but on composed homomorphism expressions that have defined input and output monoids. All optimisations may thus be considered as parameterised loops, and not as individual operations.

[5] For example, checking the names of collections and attributes.

[6] Notation may only be used to refer to variables, not other MQL expressions.

[7] Method invocation in the context of object–oriented database systems.

MQL Term	Type	MHL Translation
$\lVert$collection$\rVert$	a collection	*resident(collection_name)*
$\lVert x \rVert$	variable	x
$\lVert c \rVert$	constant (int,bool,string)	c
$\lVert A(e) \rVert$	attribute projection [6]	$A(e)$
$\lVert e_1/e_2 \rVert$	division of two integers	$\lVert e_1 \rVert / \lVert e_2 \rVert$
$\lVert e.i \rVert$	i integer, gives ith element of a record	$e.i$ [7]
$\lVert \langle e_1, \ldots, e_n \rangle \rVert$	record	$\langle \lVert e_1 \rVert, \ldots, \lVert e_n \rVert \rangle$
$\lVert e_1 \; op \; e_2 \rVert$	where op is one of $\{>, \geq, <, \leq, =, != \}$	$\lVert e_1 \rVert \; op \; \lVert e_2 \rVert$
$\lVert \text{zero}[\mathcal{M}] \rVert$		$\text{zero}[\mathcal{M}]$
$\lVert \text{unit}[\mathcal{M}](e) \rVert$		$\text{unit}[\mathcal{M}](\lVert e \rVert)$
$\lVert \text{merge}[\mathcal{M}](e_1, e_2) \rVert$		$\text{merge}[\mathcal{M}](\lVert e_1 \rVert, \lVert e_2 \rVert)$
$\lVert \mathcal{M}\{\text{e} \mid\} \rVert$		if free($\mathcal{M}$) then unit[$\mathcal{M}$]($\lVert$e$\rVert$) else $\lVert$e$\rVert$
$\lVert \mathcal{M}\{\text{e} \mid \text{x} \leftarrow \text{u}, CT_i\} \rVert$		$\text{hom}[\mathcal{M}_j, \mathcal{M}]$ $(\lambda \text{x} \; \lVert \mathcal{M}\{\text{e} \mid CT_i\} \rVert)$ $\lVert u \rVert$ where *type_of(u)* $= M_j$
$\text{hom}[\mathcal{M}_i, \mathcal{M}]$ $(\lambda \text{x} \; \mathcal{M}\{\text{e} \mid CT_i\})$ u where *type_of(u)* $= M_j$		$\text{hom}[\mathcal{M}_i, \mathcal{M}]$ $(\lambda \text{x} \; \lVert \mathcal{M}\{\text{e} \mid CT_i\} \rVert)$ $\lVert u \rVert$ where *type_of(u)* $= M_j$
$\lVert \mathcal{M}\{\text{e} \mid \text{pred}, CT_i\} \rVert$		if $\lVert$pred$\rVert$ then $\lVert \mathcal{M}\{e \mid CT_i\} \rVert$ else zero[$\mathcal{M}$]
$\lVert \mathcal{M}\{c_1, \ldots, c_n\} \rVert$	c_i are constants	$\textit{incoming}(\mathcal{M})\{c_1, \ldots, c_n\}$

Table 7.2: MQL to MHL translation

Implementation–based Improvements A number of implementation–based improvements, such as the use of indexes, have also been incorporated into the optimiser.

The rewrite system is implemented in *Standard ML* (SML). The rewrite rules are expressed as SML functions and are pattern matched on MHL abstract syntax objects. Specifically, a rewrite rule is a SML function that takes an MHL abstract syntax object and produces a list of equivalent abstract syntax objects where each constituent part denotes a substitution for the original expression in MHL. The rules that lead to a reduction in the size of intermediate results, such as *filter promotion*, are applied taking into account the restrictions that are imposed by the cardinalities of each collection and the selectivity of a number of the predicates, a small set of which are currently stored in the rewrite database. Since the *hom* skeleton is the only looping construct in MHL, the sequential efficiency of optimisations may be roughly estimated by comparing the number of loops before and after an optimisation is carried out. A simple cost evaluation function based on this is incorporated to "drive" the application of the optimisations. However, the cost function takes into account cases in which the introduction of *additional* loops increases the amount of potential parallelism in the query without increasing the overall sequential evaluation time. One example is exemplified by rule Ax 9 which introduces an additional homomorphism, and hence loop, but is nevertheless more efficient.

Note that rewrite rules in the MHL rewrite database are independent of the *skeleton rewrite database* which is part of the complex applications architecture. The skeleton rewrite database would include not only equivalences between skeletal programs, but also skeleton performance models that are parameterised by the features of the specific parallel architecture.

7.9 Parallel Type Checker

This section describes the functionality of the *Parallel Type Checker (PTC)*, which ensures that the evaluation of source MHL expressions proceeds correctly. This entails checking the argument and result Monoid categories for each homomorphism (§ 7.9.1); and, evaluating the data partitioning requirements of the query using information from the data dictionary, the monoid instance dictionary and the monoid element dictionary. In general, the evaluation of an MHL expression may require resident data to be re–organised across nodes both prior to and during the parallel evaluation phase. In order to determine the required data distribution, the PTC reduces the predicates in an MHL expression to the same normalised notation as that which is used to define data partitioning in the data partition dictionary. In order to perform this task, the PTC constructs a *Symbol Table*, during the *symbol table construction* phase, which records the predicates and the final result associated with each instantiated query variable in an MHL expression. During the symbol table construction process variables appearing in the predicates are substituted by their equivalent *VSL* (Variable Substitution Language) representations, where the VSL representation of a variable denotes the variable's generation path. The *resolution phase* produces data partitioning requirements that must be satisfied in order to carry out a parallel evaluation of the query. These requirements are used by the *object code generator* which outputs a complete skeleton program that realises the evaluation of the query, including any data re–organisations necessary. The internal structure of the PTC and object code generator is illustrated in figure 7.2 and discussed in the next two sections.

7.9.1 Monoid Transition Type Checker

The monoid transition type checker tests the category of the argument and result Monoid of each homomorphism in an MHL expression. For a function $h : \mathcal{M}_1 \mapsto \mathcal{M}_2$ to be a homomorphism, two properties have to be preserved:

$$h(zero[\mathcal{M}_1]) = zero[\mathcal{M}_2]$$
$$h(merge[\mathcal{M}_1](a, b) = merge[\mathcal{M}_2](h(a), h(b))$$

The definition of h over the $merge$ case requires that $\pi(\mathcal{M}_1) \preceq \pi(\mathcal{M}_2)$, where π is defined such that $C \preceq CM \preceq CIM$ (as defined in chapter 6). Checks are performed in order to ensure that this is the case.

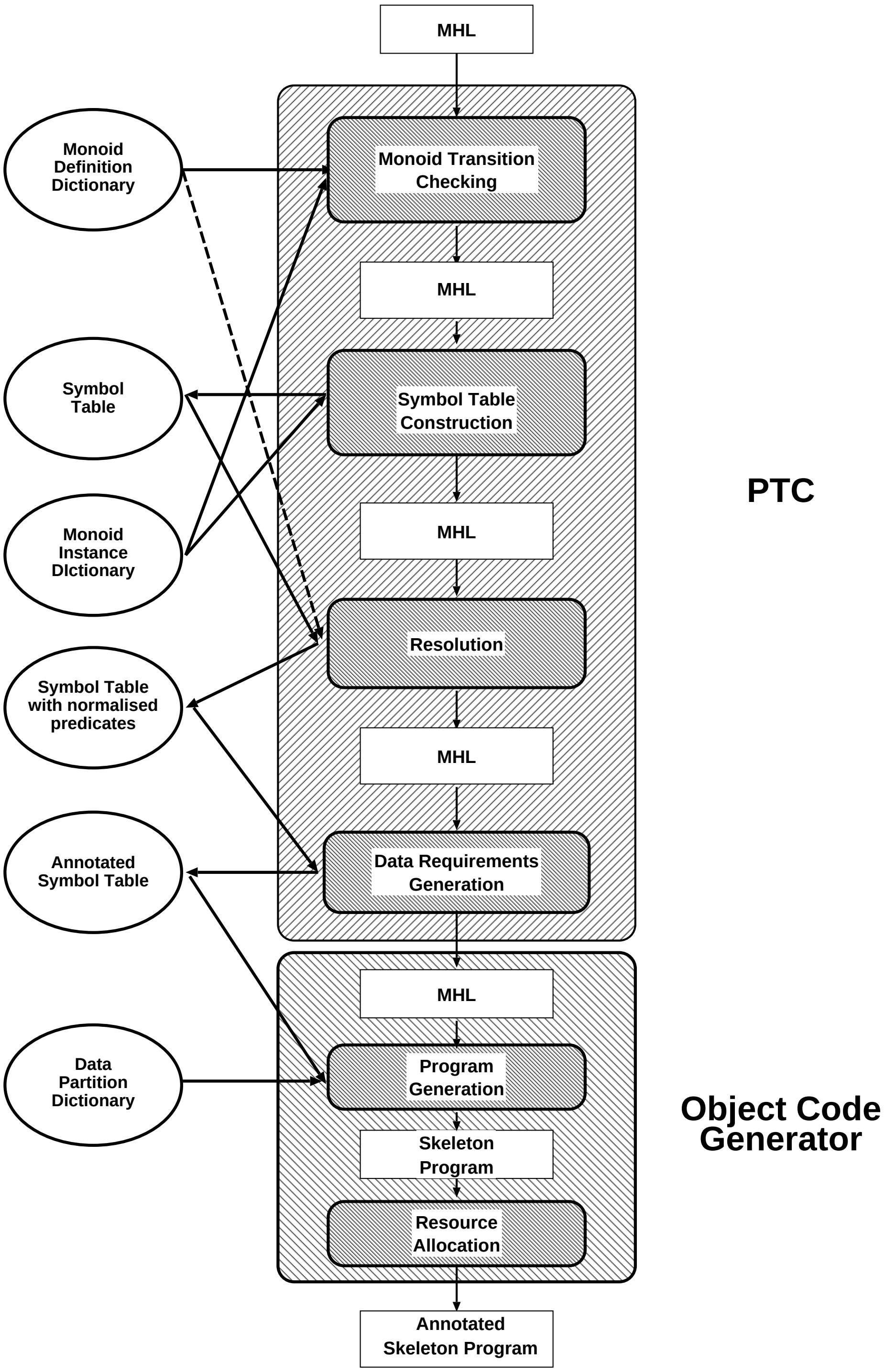


Figure 7.2: PTC and Object Code Generator

7.9.2 Symbol Table Construction

The symbol table is a collection of records each of which documents information about each variable which appears in an MHL query. The main purpose of the symbol table is to group all predicates in a query according to the variables that they are based on. Owing to the nested nature of MHL expressions, a predicate may be based upon variables that are instantiated in a homomorphism that ranges over a collection, or a homomorphism expression. The general form of each generated variable in relation to the collection from which it is generated is also recorded in the symbol table record pertaining to the variable. The algorithm which is used to construct the symbol table is presented in table D.1 in Appendix D. The symbol table construction function, $\mathcal{T}$, takes as input an MHL expression and returns a symbol table together with the head of the innermost `hom` in the query[8] in *VSL* notation (*Variable Scope Language*, § 7.9.2).

Algorithm

Consider the construction of the symbol table for a general MHL expression of the form:

$$(hom[\mathcal{M}_1, \mathcal{M}_2]\lambda\ x.\ (body)\ )\ data\_source$$

During its pass through an MHL expression the symbol table generation algorithm inserts a new record into the symbol table each time a homomorphism is encountered. The record consists of five pieces of information:

- *var — the variable itself;*
- *scope — the scope in which the variable is generated;* [9]
- *collection — the collection or partial result from which the variable is generated;*
- *tail — the generation path for the variable;*
- *pred_list — a list of predicates in which the variable appears.*

The symbol table which is constructed for the following MQL query, for example, is given in table 7.3.

$$\mathcal{M}_1\{x|x \leftarrow \mathcal{M}_2\{unit[\mathcal{M}_1](a)|a \leftarrow as, P_1(a)\}, y \leftarrow \mathcal{M}_3\{b|b \leftarrow bs, P_2(b)\}, P_3(x, y)\}$$

where terms of the form $P_i(\dots)$ denote base predicates.

Each of these components is now considered in turn.

[8] The head of an MHL term is equivalent to the head of an MQL term.

[9] The scope is of the form $\langle outer\_scope, inner\_scope\rangle$.

Variable	Collection	Tail	Scope
x	$\mathcal{M}_2\{unit[\mathcal{M}_1](a) \mid a \leftarrow as, P_1(a)\}$	$TL(\mathcal{M}_2\{HD(\Rightarrow \mathcal{M}_1(TL(\langle a, as, \langle 0,1\rangle, 2\rangle\rangle))))\})$	$\langle 0,1\rangle$
Pred_list	$\mathrm{P}_3(TL(\mathcal{M}_2\{HD(\Rightarrow \mathcal{M}_1(TL(\langle a, as, \langle 0,1\rangle, 2\rangle\rangle))))\}), TL(\mathcal{M}_3\{HD(TL(\langle b, bs, \langle\langle 0,1\rangle, 1\rangle, 2\rangle\rangle))\}))$		
y	$\mathcal{M}_3\{b \mid b \leftarrow bs, P_2(b)\}$	$TL(\mathcal{M}_3\{HD(TL(\langle b, bs, \langle\langle 0,1\rangle, 1\rangle, 2\rangle\rangle))\})$	$\langle\langle\langle 0,1\rangle, 1\rangle$
Pred_list	$\mathrm{P}_3(TL(\mathcal{M}_2\{HD(\Rightarrow \mathcal{M}_1(TL(\langle a, as, \langle 0,1\rangle, 2\rangle\rangle))))\}), TL(\mathcal{M}_3\{HD(TL(\langle b, bs, \langle\langle 0,1\rangle, 1\rangle, 2\rangle\rangle))\}))$		
a	as	$TL(\langle a, as, \langle 0,1\rangle, 2\rangle\rangle$	$\langle\langle 0,1\rangle, 2\rangle)$
Pred_list	$\mathrm{P}_1(TL(\langle a, as, \langle 0,1\rangle, 2\rangle\rangle)$		
b	bs	$TL(\langle b, bs, \langle\langle 0,1\rangle, 1\rangle, 2\rangle\rangle$	$\langle\langle\langle 0,1\rangle, 1\rangle, 2\rangle)$
Pred_list	$\mathrm{P}_2(TL(\langle b, bs, \langle\langle 0,1\rangle, 1\rangle, 2\rangle\rangle))$		

Table 7.3: Sample symbol table

Scope The scope in which each variable is instantiated by a homomorphism is recorded in order to detect the innermost symbol table record, and hence, collection to which a particular variable refers. This enables substitutions of the most appropriate tail term to be made into *all* symbol table predicate lists in which the variable appears. The *current_scope* is passed as a parameter in the symbol table construction algorithm $\mathcal{T}$ and is set to zero at the start of the construction algorithm. The *current_scope* changes when either a homomorphism or a conditional is encountered. Each time that a new homomorphism is encountered, the scope for the new variable instantiated by the homomorphism is set to $\langle current\_scope, 1\rangle$. The *current_scope* is also set to this value. The $BODY$ of the homomorphism is processed with $\mathcal{T}(BODY, \langle current\_scope, 1\rangle)$ whereas the *data_source* for the homomorphism is processed with the scope $\langle current\_scope, 2\rangle$.

When a conditional of the form if *cond_exp* then *then_part* else *else_part*, is encountered, the scope is increased to $\langle current\_scope, 3\rangle$ for the processing of the *cond_exp* part *only*. The other terms in the predicate are processed with $\mathcal{T}(then\_part, current\_scope)$. In such case, due to the uniform treatment of free monoids and ground monoids, it would not be possible to distinguish a Monoid that appears in the conditional part of a predicate. A predicate may take the form of a boolean–valued ground Monoid, or a complex predicate such as $sum\{head\_term \mid tail\_term\} < c$, where c is some constant. Hence, the need to process the predicate part of a conditional at a different scope.

Collection The data_source of each homomorphism pertaining to the symbol table record is recorded, verbatim, in the collection field of the record corresponding to the particular variable generation. The collection field is processed when/if an instance of the relevant variable appears in the head of the innermost MQL comprehension which signifies the final result of the MHL expression, or when its generated variable appears in a base predicate. The result of the processing of the *data_source* is recorded in the tail field of the record so that it does not have to be re–evaluated at a later stage.

Tail The tail component of each symbol table record denotes the generation path of the variable pertaining to the symbol table record and is evaluated using the information in the data_source field. If the *data_source* of a homomorphism is a base collection, then a record is constructed consisting of the variable, the scope at which the generation occurs and the name of the base collection. This is recorded in the tail field. Should the *data_source* be some other MHL term, then the symbol table construction algorithm is applied to it. The tail field is then set to $\leftarrow \mathcal{M}\{hv\}$, where hv is the value which is returned by $\mathcal{T}(data\_source, \langle generation, 2\rangle)$. This term denotes the presence of a homomorphism term. The expression that is recorded in the tail field is substituted, in place of the variable, into predicates that use the variable.

Pred_list The pred_list of a symbol table record is a list of base predicates[10] in which the variable appears. During symbol table construction, all base predicates are recorded in the pred_list component of the symbol table records of variables that appear in the predicate. The variables appearing in the predicates are substituted by their VSL equivalents (§ 7.9.2). Complex predicates, namely those which involve homomorphism expressions, either in the form of monoid homomorphism over boolean values, or a *relop* b, where a or b are Monoid homomorphisms with a ground Monoid result, are processed with the symbol table construction function $\mathcal{T}$. For example, the predicate $cd\_id(cd) = cd\_id(track)$ in the following query:

$$set\{\langle cd\_id(cd), duration(track)\rangle | cd \leftarrow cds, track \leftarrow tracks,\\ duration\_track(track) > 15, cd\_id(cd) = cd\_id(track)\}$$

would appear in the pred_list of the symbol table record for both variables cd and $track$.

Variable Scope Language

The *Variable Scope Language*, VSL (table 7.4), is a succinct *reduced variable* notation for the definition of the generation path of the variables instantiated within MHL queries. This notation is used in place of variable names in predicates and in the *tail* component of symbol table records. The generation path of a variable provides information on the level of nesting of the particular variable with respect to base collections. For example, consider the VSL expression given for variable x in symbol table 7.3:

$$TL(\mathcal{M}_2\{HD(\Rightarrow \mathcal{M}_1(TL(\langle a, as, \langle 0, 1\rangle, 2\rangle))))\})$$

The term $\mathcal{M}_2$ denotes the type of the collection from which a is generated and the enclosed HD term denotes the complete generation path for the variable x. The result of the query has a greater level of nesting than the argument free monoid xs. This is by virtue of the $unit[\mathcal{M}_1](x)$ constructor which appears in the head of the innermost comprehension. Hence, the presence of the $\Rightarrow$ immediately inside the HD constructor that indicates an increase in the level of nesting in VSL form. Note that the argument to the inner TL term denotes the free Monoid from which a particular variable, in this case x, is generated.

7.9.3 Resolution

The resolution phase performs the task of producing a set of data partitioning requirements for the *object code generator* from the information stored in the symbol table. This process

[10] A predicate is of the form a *relop* b where both a and b are literals, generated variables, or variable projections.

Form	Description
$NULL\_head$	null
$zero[\mathcal{M}]$	zero of a Monoid
c	c is a constant
xs	xs is a collection
v_i	v_i a variable
$\langle v_i, base\_collection, scope\rangle$	a "head" record
$\langle VSL_{exp}, \ldots, VSL_{exp}\rangle_r$	record
$att\_sel(VSL_{exp})$	projection
$VSL_{exp}.i$	i^{th} component of a record
$HD(VSL_{exp})$	VSL_{exp} appears in the head of an MHL expression
$TL(VSL_{exp})$	VSL_{exp} appears where variables are generated from a base collection or partial results
$\Rightarrow \mathcal{M}(VSL\_exp)$	normally appears in the head of MHL expression and indicates the application of the injection, and an increase in the level of nesting.
$\Leftarrow (VSL_{exp})$	generation from $f(var)$, where $f \in \{id, att\_sel_i\}$
$merge[\mathcal{M}](VSL_{exp}, VSL_{exp})$	merge of two VSL expressions
$\mathcal{M}\{VSL_{exp}\}$	$\mathcal{M}$ is a free Monoid
$SM\{VSL_{exp}\}$	where $\mathcal{M}$ is a ground Monoid (boolean/other)

Table 7.4: VSL

is termed *VSL normalisation*. During this phase, the predicates in the pred_list of all symbol table records are normalised. This normalisation process reduces the *variable expression*[11] in the pred_list of each symbol table record expression to a normal form.

VSL Normalisation

The VSL normalisation algorithm that is applied to all variable expressions within the pred_list of each symbol table record is given in Appendix D. Since many MHL expressions and the predicates contained within them have variables which are instantiated by homomorphisms that are applied to other homomorphism expressions, normalising the generation path of these variables reduces the predicates to simpler forms. Matching against specific predicate forms is inefficient and does not yield all of the possible matches against the data partition dictionary. VSL normalisation provides a common base for comparing query predicates in the predicate lists of symbol table records to data partition predicates in the data partition dictionary. By reducing the VSL expressions that are in place of each variable, the likelihood of finding a match against the predicate in the *data partition dictionary* is increased.

The normalisation algorithm follows a top–down path through the MHL expression denoting the query. The main part of the normalisation algorithm concerns homomorphisms that are applied to *free* monoids. The case in which terms are reduced occurs where there is a homomorphism from a *filtering* homomorphism. This occurs when the outer homomorphism is a collection and the data source is a homomorphism over a collection.

The treatment of the Monoid *merge* function deserves particular attention. Intuitively, it can be seen that the data partitioning requirements that are determined by the *merge* function are dependent upon both of its two arguments. The *divide* function, which

[11] A variable expression is defined to be a VSL expression that has been substituted into a simple predicate in place of a variable.

is given in the normalisation algorithm, replicates the whole outer and inner expression such that there are two versions of the original predicate, and hence, two sets of data requirements to be satisfied. Consider the following MQL query.

$$\mathcal{M}\{\langle a, b\rangle | a \leftarrow as, b \leftarrow merge[\mathcal{M}](exp_1, exp_2), att\_sel_i(a) = att\_sel_j(b)\}$$

The (un–normalised) VSL form of the predicate in the query is:

$$att\_sel_i(TL(\langle a, as, \langle\langle 0,1\rangle, 2\rangle\rangle)) = att\_sel_j(TL(merge[\mathcal{M}](VSL\_exp_1, VSL\_exp_2)))$$

Which, when normalised, gives:

$$att\_sel_i(TL(\langle a, as, \langle\langle 0,1\rangle, 2\rangle\rangle)) = att\_sel_j(VSL\_exp_1)$$
$$att\_sel_i(TL(\langle a, as, \langle\langle 0,1\rangle, 2\rangle\rangle)) = att\_sel_j(VSL\_exp_2)$$

where $\mathcal{M}$ is a free Monoid. The query is reduced to two normalised predicates; data partitioning requirements have to be satisfied for both.

Treatment of Idempotent Filters

Consider the VSL substituted filter given below:

$$P_j(TL(\mathcal{M}_i(HD(TL(\langle a, as, scope_a\rangle)))), TL(\langle b, bs, scope_b\rangle))$$

The leftmost VSL term in the filter matches against the rule $\mathcal{N}(TL(VSL\_exp))$ in the resolution algorithm. The applicability of the reduction rule depends on the category of the monoids $\mathcal{M}_i$ and as. Suppose that as is a general free Monoid and that $\mathcal{M}_i$ is a free Monoid in the *CIM* category. In this case, if the filter P_j is a non–idempotent filter, then the reduction rule may *not* be applied since the information regarding the non–idempotent filter is lost in the normalisation process. The application of the reduction gives the *incorrect* normalised form of the filter given below:

$$P_j(\langle a, as, scope_a\rangle, \langle b, bs, scope_b\rangle)$$

Hence, the application of the normalisation rule that deals with $\mathcal{N}(TL(VSL\_exp))$ has to incorporate some guards to take into account the properties of the base free monoids and interim result monoids. Issues relating to the treatment of such cases are discussed in Appendix D.

7.9.4 Data Requirements Generator

VSL normalisation reduces the VSL expression in place of each variable in predicates of the pred_list of each symbol table record. The subsequent phase of the compilation process, the *data requirements generator*, produces annotations for symbol table records, where each annotation outlines the data partitioning requirements for each generated variable. The annotations may take three forms: *null action* (denoted by $NULL$), a *duplicate* request on a specific collection (denoted by *DUPLICATE(collection)*), or, a *partitioning* request on a base predicate (denoted by *PARTITION(VSL_exp relop VSL_exp)*). The

PARTITION annotation does not prescribe a particular data partitioning method, but defines the properties that the data partitioning must satisfy. The algorithm which is used to determine the data partitioning requirements for each symbol table record, str_i, at a given scope is given in Appendix D. The algorithm detects cases where the query form does not prescribe any particular data organisation for a generated variable (symbol table record). This occurs when an instantiated variable does not appear in the result of the body of the particular homomorphism corresponding to the symbol table record and there are no predicates which use any other variable in its predicate list. In either case, the homomorphism is said to be *redundant*. If there is more than one variable in the final result,[12] and there are no common predicates on the result variables, then the algorithm tags the symbol table records pertaining to each of the variables with a *DUPLICATE* tag. This signifies the need to produce a *cross product*. On the other hand, if the pred_list of a record contains any multi–variable predicates, then each of these predicates is tagged with a data partitioning request on the predicate. The records that correspond to the other common variables are also marked with a *PARTITION* request.

The output of the data requirements generator is an annotated symbol table. The object code generator produces a skeleton program from the MHL query using the annotated symbol table.

7.10 Object Code Generation

Object Code Generation consists of two distinct subtasks: parallel *program generation*, which produces a skeleton program that realises the functionality of the query including data re–arrangements; and, *resource allocation*, which refers to the process of allocating processing resources for the evaluation of the query taking into account physical constraints such as the nodes that have access to secondary storage. Each of these roles is now considered.

7.10.1 Program Generator

During program generation, a skeleton program that realises the complete functionality of the MHL program is produced. The program generator processes information stored in the symbol table and correlates this against the partitioning information held in the *data partition dictionary* to produce a skeleton program from the input MHL. The information stored in the data partition dictionary is in normalised VSL notation, the same format as the symbol table annotations. Hence, the correlation process is simplified since the format of the information in the *data partition dictionary* is precisely the same as the annotated predicates of the predicate lists of symbol table records.

Each MHL program may be considered to have a homomorphism construct tree. The code generator processes each of the homomorphisms in the query tree in a top–down (left) breadth–first fashion such that the homomorphisms down the left spine of the query tree are processed first, where the *body* of the hom is on the left spine and the *data–source* of the `hom` is on the right spine. The left–spine constitutes the nested homomorphism

[12]The symbol table construction function $\mathcal{T}$ returns the symbol table and the *head* of the final result, where the *head* term is equivalent to the head of an MQL expression.

expression that drives the evaluation of the main query and is translated into nested `hom` instances in which each `hom` is inside the *body* of an outer `hom`. During this process, appropriate `hom` skeleton stubs are generated for each homomorphism of the data inputs to each of the nested `hom` skeletons, where each stub denotes a single `hom` within a nested evaluation. The input interface to a `hom` skeleton stub is also parameterised with the outer scope variables that are used within the body of the `hom`. If the homomorphism ranges over a collection that denotes a resident collection, then the collection input list of the skeleton is marked with the name of the resident variable. However, if the homomorphism ranges over a *homomorphism* expression that evaluates to a collection, then the data source is processed independently when the *next left spine down* of the tree is processed.

At each `hom` node within the query tree, the corresponding symbol table record is identified. The annotation accompanying each predicate in this symbol table record is then inspected. The annotation may either be *NULL* a flag that signifies no action, a *DUPLICATE* request, or a *PARTITION* request based on a multi–variable predicate. The *NULL* flag signifies that the particular generation does not prescribe any specific data partitioning requirements. The *DUPLICATE* and *PARTITION* requests are now discussed.

DUPLICATE

The *DUPLICATE* request normally occurs when the evaluation of part of a query results in a cross product of two collections, such as in the MHL query shown below:

$$set\{\langle x, y\rangle | x \leftarrow cds, y \leftarrow tracks, P_1(y), P_2(x)\}$$

For this query, the symbol table records pertaining to *each* of the generations would have the same *DUPLICATE* annotation. This signifies the need to duplicate *one* of the two collections. The choice as to which one is duplicated should be made in light of the size of the data that is duplicated, and the data requirements of other components of the query. One of the collections is chosen for duplication, the other symbol table record is marked as having been processed so that it is not considered again. Normally $n - -1$ *DUPLICATIONS* are required when there are n variables in the *head* of the final result that do not have any common predicates.

$$(hom\ \lambda x.\ (hom(\lambda y.unit[bag](\langle x, y\rangle)))(tracks))\ cds)$$

Table 7.5: Cartesian Product – Source

The duplication itself is realised through the insertion of a `pipe` skeleton between the data source of the `hom` and the main body of the `hom`. This `pipe` skeleton nests a `split` skeleton that is fed by a `hom` skeleton. For example, the source and processed forms of a query which evaluates a Cartesian product are given in table 7.5 and table 7.6, where α types `x`.

Note that the variable *bottom_level_cells* is set by the resource allocation phase and monoid-name is set to NULL since this particular instance of the `split` function performs data duplication.

```
main in() out(bag res)
  pipe P in() out(bag res)

       hom h1 in (bag resident(cds)) out(bag collated-cds)
           λz.  unit[bag](z)
       end hom

       split split1 in(bag collated-cds) out(bag duplicated-cds)

           int bottom_level_cells;
           monoid_name NULL;

           DUPLICATE;

       end

       hom h2 in(bag duplicated-cds) out(bag res)
           λx.  hom h3 in(α x,bag resident(tracks)) out(bag res)
                    λy.  unit[bag](⟨x,y⟩)
                end
       end
  end

 end
```

Table 7.6: Cartesian Product – Processed

PARTITION

PARTITION requests are caused by multi–variable predicates that appear in the predicate lists of symbol table records. The program generator compares each data partition request with the corresponding record in the data partition dictionary. If the comparison yields a positive match, then no data reorganisation is necessary for the particular `hom` instance. Should this not be the case, then the *data_source* to the *hom* instance needs to be re–organised, as determined by the annotation that accompanies the appropriate symbol table record. However, it may not be possible to satisfy all the partitioning requirements at the same time for anything other than equality based predicates. In such cases, the program generator produces a series of partial results, one for each conflicting partition request. This process may be considered as that of substituting an *n–ary* join with a number of *binary* joins.

In the current implementation, the only flavour of n–ary equality based multi–variable predicates for which support is provided in the program generator are those in which the attribute projections are the *same* across each instance of *each* variable in each of the predicates, as exhibited by the following MQL query:

$$\mathcal{M}\{\langle x,y,z\rangle | x \leftarrow xs, y \leftarrow ys, z \leftarrow zs, att\_sel_x(x) = att\_sel_y(x), att\_sel_x(x) = att\_sel_z(z), \\ att\_sel_y(y) = att\_sel_z(z)\}$$

Data re–organisation of the data_source is realised through the application of a (division) `split` skeleton and a generating `hom` skeleton between the data_source and the main *body* of the *hom* instance. For example, consider the MQL query given below:

$$set\{\langle cd\_id(cd), length(track)\rangle | cd \leftarrow cds, track \leftarrow tracks, cd\_id(cd) = cd\_id(track)\}$$

In the absence of a data partitioning on the cd_id attribute, the result of the code generation process for this MQL query is given in figure 8.5 in chapter 8.

7.10.2 Resource Allocator

The skeleton program that is produced by the program generator fulfills the functional requirements for the evaluation of the original MQL/MHL query. The resource allocation process assigns a process graph to the skeleton program. In order to realise this functionality, the resource allocation process assigns a *virtual process tree* (VPT) to each skeleton within a skeleton program (the VPT for the `hom` skeleton is shown in figure 7.3). The VPT identifies the parallel behaviour of the skeleton in terms of a number of parallel

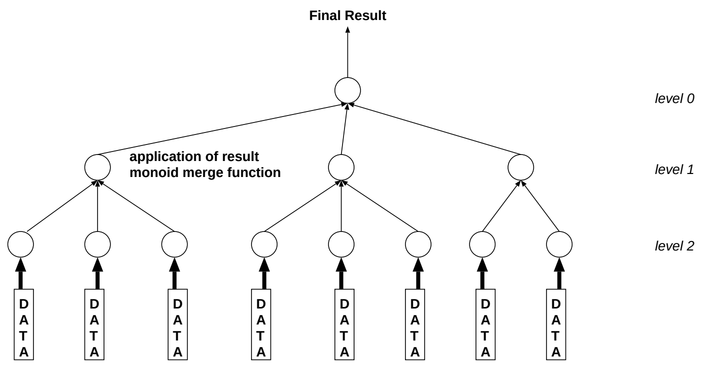


Figure 7.3: VPT for a `hom` skeleton

communicating processes. Although the nature of a skeleton dictates the particular VPT for the execution, this does not determine the *number* of physical *processors* that are involved in the execution of the skeleton or their arrangement. Furthermore, it also does not determine the mapping between the nodes in the VPT and the physical processors on the parallel machine: it is the resource allocator that prescribes a mapping between the nodes in the VPT and the physical cells on the parallel machine. The output of the resource allocator is annotated skeleton code.

The first part of the resource allocator's task, namely that of determining the number of processes involved in the evaluation of each skeleton, and that of determining the precise nature of the VPT, is a resource allocation problem. Nonetheless, there are certain restrictions that apply relating to the composition of some skeleton forms. A `hom` skeleton generates elements from either a ready–partitioned (*resident*) collection or the output of another skeleton, such as `split` which performs data re–organisation. In this case, the number of bottom–level nodes in the VPT of an instance of the `split` skeleton must be equal to the number of bottom–level nodes in the VPT of the `hom` skeleton which it *feeds*. Although such restrictions concern localised pairs of skeletons, in order to perform the resource allocation task efficiently it is necessary to consider the whole skeleton program as a complete unit. Some bottlenecks in the resulting code can be statically predicted through performing compile time transformations of the application constructor tree as in [BDP93]. The transformation process modifies the original parallel structure of the application without its semantics.

The current implementation of the resource allocator deals with these issues in a simplified way. For a single `hom` skeleton that generates elements from a resident *data–source*, the number of bottom–level nodes within the VPT is determined by the number of physical disk–cells[13] across which data has been partitioned for the particular resident collection. This also defines the mapping from the bottom–level nodes within the VPT to the individual cells on the machine. The VPT of each skeleton is an experimental parameter that is investigated in chapter 9.

7.10.3 VPT to Machine Node Mapping

Once the VPT for each skeleton has been determined, the resource allocator needs to determine and define the mapping between the VPT and physical processors on the parallel machine. The output from this process must succinctly define a mapping that enables the runtime system of each cell on the parallel machine to ascertain its evaluation and communication responsibilities on a per skeleton basis. Note that a cell may not have any responsibilities with respect to the evaluation of a skeleton. In this case, the annotation accompanying the skeleton defines no VPT position for the particular cell. Defining a 1–1 mapping between the nodes in the VPT and the parallel machine is inefficient, especially for massively parallel processing. Hence, an *algorithmic* approach is used in which the VPT is defined in terms of a specific tree walking algorithm and in terms of a declarative map parameterised by a physical starting processor.

Consider the following query which consists of a single homomorphism:

$$bag\{\ x \mid x\ \leftarrow cds,\ sales(x)\ >\ 30000\ \}$$

This MQL query is translated into the skeleton program given on the left hand side of figure 7.4. The resource allocator chooses an appropriate tree walking algorithm for the `hom` skeleton and prescribes the mapping from the VPT to the physical processor on the parallel machine. This mapping is recorded as an annotation to the skeleton. The annotation which takes the form of an array of integers defines the form and mapping of the VPT and is termed the *virtual topology map* (VTM). The annotated skeleton program for the above MQL query is given on the right hand side of figure 7.4. The VTM in the figure prescribes the individual cells which are involved in the evaluation of the particular skeleton. The contents of each array position, j, determine the number of processors which are utilised for level j of the evaluation tree. Level 0 of the evaluation tree defines the root level (the topmost node). There are three levels within the VPT that is prescribed in the VTM given in figure 7.4, having 1, 3 and 8 processors at levels 0, 1 and 2 of the VPT respectively.

The *style* parameter, $seq(0)$, determines the allocation algorithm and the starting node for the processor allocation. In this case, this is physical processor number 0. The $seq(0)$ parameter defines a sequential left to right, bottom–up allocation algorithm such that the bottom–level nodes within the VPT (figure 7.3) are assigned to processors 0 to 7, the 3 at

[13] Disk-cells are cells that have access to local disks. The cells in level 2 of figure 7.3 are disk-cells.

```
hom h1 in(bag resident(cds)) out(bag res)
    λx.  (if sales(x) > 30000
            then unit[bag](x)
            else zero[bag])
end

main in() out(bag res)
    h1 in() out(bag res)
end
```

Style	Lev	0	1	2	3	...	max_levels
seq(0)	*Procs*	1	3	8	-	...	-

Figure 7.4: Annotated skeleton program

the intermediate level to processors 8, 9 and 10, and the single one at the root to processor 11. The next section details each of the allocation algorithms.

Algorithms

Four algorithms are incorporated for the definition and interpretation of VTMs, termed *none*, *seq(*i*)*, *reuse(*i*)* and *total(*i*)*. These algorithms prescribe walks of the VPT pertaining to each skeleton except that the behaviour required for the `pipe` skeleton is implemented locally for all evaluations. The particular characteristics of the target parallel platform means that communication cost between a pair of distant cells is similar in cost to communication between adjoining cells [IMM93]. Hence, the issue of locality is not considered in the allocation of processors.

The four algorithms are now detailed.

none The *none* annotation indicates that the particular skeleton is to be evaluated locally at each cell, in which case, $\forall i; i = 0\ to\ 127.\ VTM[i] = 1$, since there are 128 cells on the particular parallel machine. Each location, i, of the VTM determines the responsibilities of cell i. If $VTM[i] = 0$, this signifies that cell i has no responsibility with respect to the evaluation of the particular skeleton. The *none* interpretation primarily occurs where an instance of an inner `hom` is to be evaluated *completely* locally and independently on each cell. It is used in complex MHL expression in which the physical data partitioning ensures that a nested query may be evaluated as the *sum* of the evaluations of the nested expressions on each leaf that is involved in the evaluation. For example, the VTM associated with the inner `hom` instance, `h2`, in the skeleton program given below, is a *none* VTM, if the two argument collections are distributed on the predicate $cd\_id(x) = cd\_id(y)$:

```
hom h1 in(bag resident(cds)) out(set res)
    λx.  (hom h2 in(α x, bag resident(tracks)) out(set res)
        λy.  (if cd_id(x) = cd_id(y)
                then unit[set](⟨f(x), g(y)⟩)
                else zero[set])
    end)
end
```

For the `pipe` and `farm` skeletons, the *none* annotation is used to indicate that the evaluation of the skeleton is to proceed completely locally.

There is a 1–to–1 mapping between the nodes in the VTM and the processors on the parallel machine when using the *none* algorithm.

seq(i) A round–robin allocation strategy is used to allocate nodes in the VPT to physical cells on the parallel machine. For the `hom` skeleton, physical cells are allocated from cell_id i to cell_id *number_of_partitions* + i − 1, in a round–robin depth-first fashion, with the leftmost leaf in the VPT being allocated first. The value of i depends on the information relating to the particular data source from which the data emanates and is recorded in the *data partition dictionary* (§ 7.3). In most cases, a base collection is partitioned such that there is one partition on cells from cell_id 0 to cell_id *number_of_partitions* − 1. For the `farm` skeleton, $VTM[0] = j$ where j prescribes $j - 2$ independent farm workers together with a single job emitter node at cell j and a result collector node at cell $j - 1$.

There is a 1–to–1 mapping between the processors in the virtual processor map and the physical processors when using the *seq*(i) algorithm.

reuse(i) Although an evaluation with the *seq*(i) algorithm may utilise pipeline parallelism by the delivery of elements from the lower–level nodes through to the interior nodes in the virtual processor tree *before* the termination of the leaf node evaluations, the number of processors used in an evaluation prescribed by the *seq*(i) algorithm is equal to the number of nodes in the virtual processor tree. In contrast, the *reuse*(i) algorithm allocates only as many (physical) processors as there are *leaf* nodes in the virtual processor tree. Once the allocation of leaf nodes has been completed, the interior nodes in the virtual processor tree are mapped in such a way that an interior node in the virtual processor tree is mapped to one of the (already allocated) cells on the parallel machine. The left–most node in the penultimate level in the virtual processor tree is mapped to the first cell, i. The second is mapped to *last_allocated_to_left_neighbour* + 1 and henceforth. For example, the VTM in figure 7.5 is mapped such that the leftmost node in the pen–ultimate level of the virtual processor tree is mapped to cell_id 0, the next node to cell_id 3 and the right–most to cell_id 6. In cases in which one of the cells in an intermediate level has a lower number of lower–level data–sources, the cell is not chosen as a merging cell for the next level (up) of the evaluation, as depicted in the VPT of the skeleton. The *reuse*(i) algorithm is not applied to the `farm` or the `pipe` skeletons.

total(i) A *total*(i) map defines a 1–to–1 allocation of each virtual node in the VPT and each cell on the parallel machine. A map is individually given for each node. The VTM value for location i in the array gives the real–node–to–cell mapping for *node* i in the virtual processor tree. Hence, the maximum size of the array is equal to the total number of nodes on the target parallel computer. This restriction limits the number of nodes in the VPT of any one skeleton, which may not exceed the total number of processing nodes on the target parallel machine. For the `hom` skeleton, $VTM[k]$ determines the physical cell that is assigned to node k in the VPT using the depth left–first walk. The mapping for the `farm` skeleton with $j + 1$ workers is such that $VTM[0]$ gives the mapping for the emitter node of the `farm`, $VTM[k]$ where $k : 1 \ldots j - 1$ gives the mapping for each of the worker processes, and $VTM[j]$ determines the mapping for the collector process. For `hom`, $VTM[i] = -1$ signifies the end of the current process grouping and $VTM[i] = NULL$ signifies the end of the row.

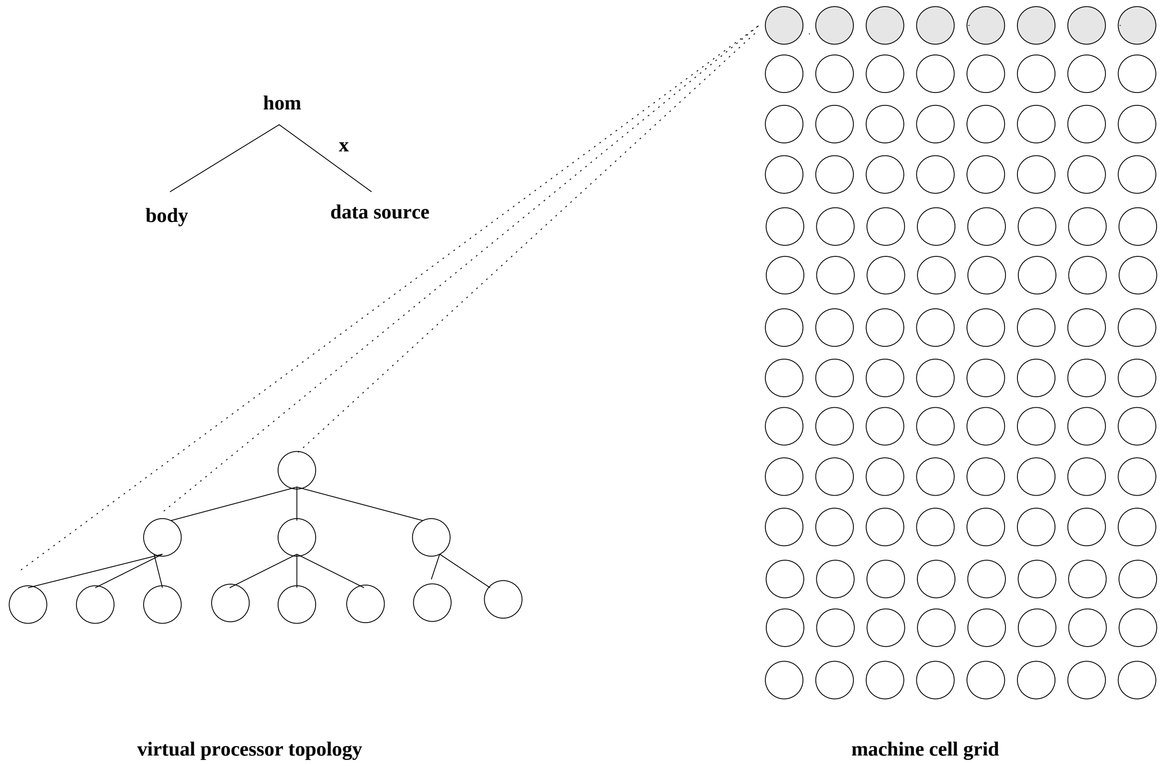


Figure 7.5: *reuse(i)* VPT to machine cell grid mapping for `hom`

7.11 Summary

This chapter presented the compilation tools that process MQL queries producing annotated skeleton code suitable for execution on the parallel machine. The input to the compilation process is a query in MQL; an SQL to MQL translator has also been implemented. Source MQL is translated into MHL that is input into the MHL rewriter and then into the PTC. The PTC is the main component of the front–end compilation tools. It takes an MHL form and generates a symbol table in which predicates are grouped according to the variables that they use. Each occurrence of a variable in a predicate is substituted by the VSL formulation for the predicate which is then normalised. The resolution phase of the PTC labels each symbol table record with an annotation that determines the data partitioning requirements of the query. The program generator realises the data partitioning requirements for each record in the symbol table taking into account the partitioning on resident collections in realising an implementation for the query. The resultant skeleton code then undergoes a resource allocation process in which processing resources are allocated to the evaluation of each skeleton within the overall program. The output of the object code generator is an annotated skeleton program that is suitable for evaluation on the parallel machine software architecture. This is discussed in Chapter 8.

Chapter 8

Runtime Architecture and Implementation

This chapter outlines the runtime software architecture (§ 8.1) and its prototype parallel implementation, for decision–support applications (§ 8.2), on a 128–node Fujitsu AP1000 parallel computer at the *Imperial College and Fujitsu Parallel Computing Centre* (IFPC). The software architecture is such that it can be implemented on a workstation cluster, or any (MIMD) parallel machine on which some nodes have access to disks.

8.1 Parallel Machine Software Architecture

The compilation tools presented in chapter 7 process an MQL (or SQL) query producing an Annotated Skeleton Program (ASP), which is the input to the parallel machine runtime software architecture. The parallel machine software architecture comprises a master process which executes on the front–end workstation which controls access to the parallel machine, and one worker process per node of the target parallel machine. For the evaluation of each query, the master runtime system broadcasts the ASP to all of the worker processes. The result of the processing of the ASP is communicated back to the master by the worker processes. The following sections detail the runtime software architectures of the master and worker processes (see figure 8.1).

8.1.1 Master Runtime System

The master's runtime system interacts with the last phase of the decision–support architecture and performs the task of co–ordinating the evaluation of a skeleton program on the target parallel machine. The main function of the master runtime system is to broadcast the ASP to all workers. Upon the completion of the evaluation of the ASP, the final result is returned to the master by one of the worker processes.

Data distribution/redistribution is the process of partitioning a collection between a number of processors. In implementations of parallel database management systems, it is carried out using bespoke mechanisms by the database server process, which is analogous to the master in the architecture presented in this thesis. In static data partitioning, a persistent collection is striped amongst the nodes of a parallel machine and each partition is stored on the local disks of each node across which partitioning has been carried out. In dynamic data partitioning a (already partitioned) collection is partitioned during the

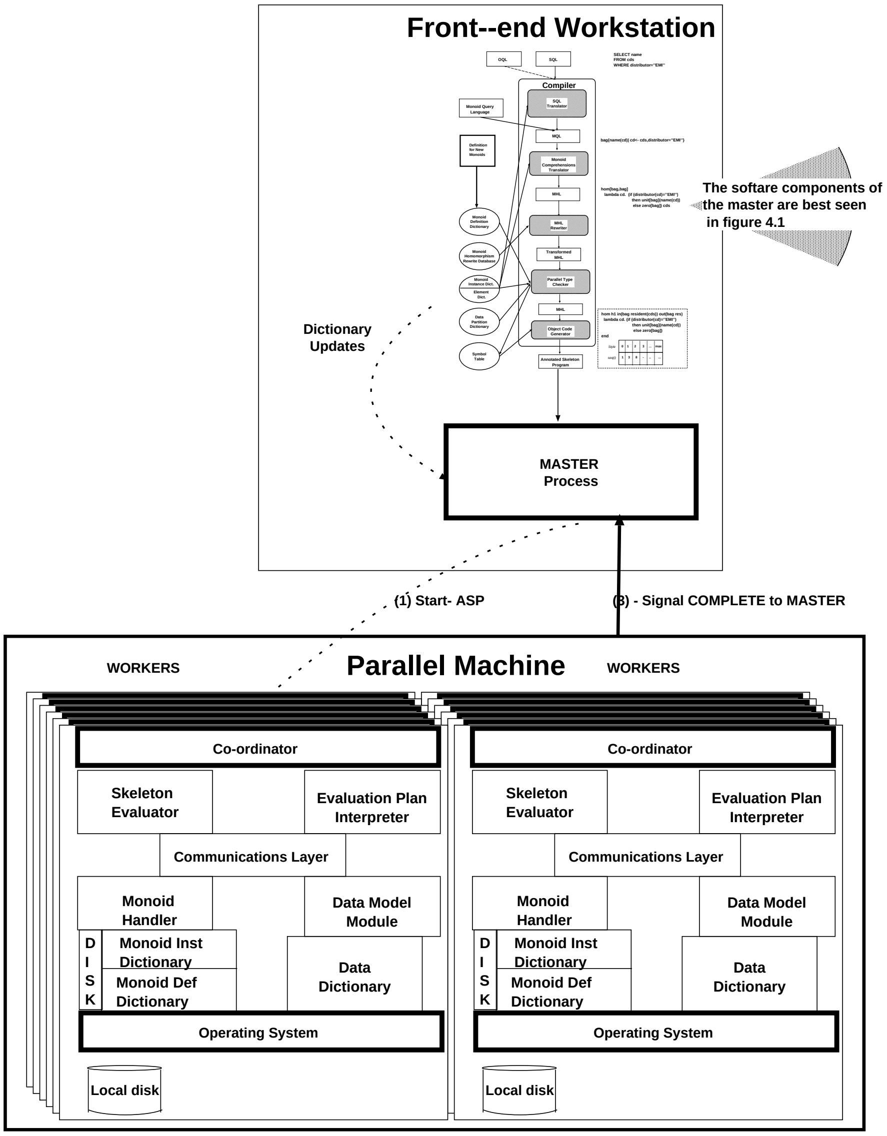


Figure 8.1: Parallel Machine Software Architecture on the Front–end Workstation and Nodes on the Target Parallel Machine.[1]

evaluation of a query which requires a data distribution which is different to the default one. In contrast with the bespoke mechanisms that are used to achieve these in parallel database management systems, an ASP is capable of expressing both static and dynamic data distribution and redistribution. Data partitioning is handled by the `split` skeleton which may be used in programs to achieve static or dynamic data distribution. For static partitioning the output parameter of the `split` skeleton is annotated with a *resident* annotation. This is all that is required to achieve static partitioning. Hence, the master does

[1]The software components of the master are best seen in figure 4.1.

not perform any special functions for the transfer of data to the workers running on the target platform. This task is carried out within the normal skeleton program evaluation environment. A parallel implementation of a hash–join is outlined in § 8.1.2.8 to show the ease with which dynamic partitioning is realised.

The master process also manages updates to the three dictionaries which store information on resident collections and are held by the worker processes. These are the the *monoid definition dictionary*, *monoid instance dictionary*, and the *element dictionary*. Updates to the monoid definition dictionary implement the data type extensibility property of the language model by allowing parallel query evaluation on user–defined data types. The language–level aspects of data type extensibility are discussed in chapter 6.

This completes the discussion of the master process. The next section outlines the architecture of the worker process(es).

8.1.2 Worker Runtime System

The worker process (figure 8.2) executes an ASP and communicates the result of the execution to the destination that is determined by the ASP. At runtime, all workers receive the same ASP. The ASP comprises two logical parts: a functional part which comprises a skeleton program, and a non–functional task determination component — termed VTM (virtual topology map) — which is associated with *each* skeleton in the skeleton program and determines the precise task that *each* worker has to perform in the evaluation of the skeleton. Each VTM within the ASP determines the global (virtual) interconnection between the workers involved at a particular stage of the evaluation.

The ASP is received from the master by the co–ordinator on each worker which forwards it to the local skeleton evaluator, as indicated by the arrow in figure 8.2. The co–ordinator on the worker conducts the flow of the evaluation that is carried out by the skeleton evaluator. The skeleton evaluator subsequently starts executing the ASP, skeleton by skeleton. For each skeleton the skeleton evaluator consults the evaluation plan interpreter. The evaluation plan interpreter uses the VTM that accompanies the skeleton to determine the worker's responsibilities with respect to the evaluation of the skeleton. It signals the skeleton evaluator to perform one of two actions: carry out the appropriate evaluation for the particular skeleton, as determined by the VTM; or skip to the next skeleton in the ASP.[2] The particular action indicated is determined by the parameters returned by the evaluation plan interpreter. The parameters take the form of labels which determine the role that the worker has in the evaluation of the particular skeleton. For the `farm` skeleton, for example, a worker may be the *emitter* that assigns jobs to slaves. Alternatively, it may be one of the *slave* processes, or a *collector* node gathering results from *slave* processes. For an instance of the `hom` skeleton, the skeleton evaluator needs to know whether the worker is an intermediate level cell or a bottom–level data processing worker. The evaluation plan interpreter determines this from the VTM that is provided to it by the skeleton evaluator. Upon the completion of the evaluation of each skeleton, the co–ordinator is informed of the destination of each skeleton evaluation. If the result of the evaluation of a skeleton is to be communicated to a remote worker, the co–ordinator organises the transmission of the results. Where pipelining is possible, the co–ordinator manages the early delivery of results. In order to facilitate this, a flexible buffer protocol should be implemented

[2] All VTMs are capable of identifying idle cells.

between the co–ordinator and the monoid handler.

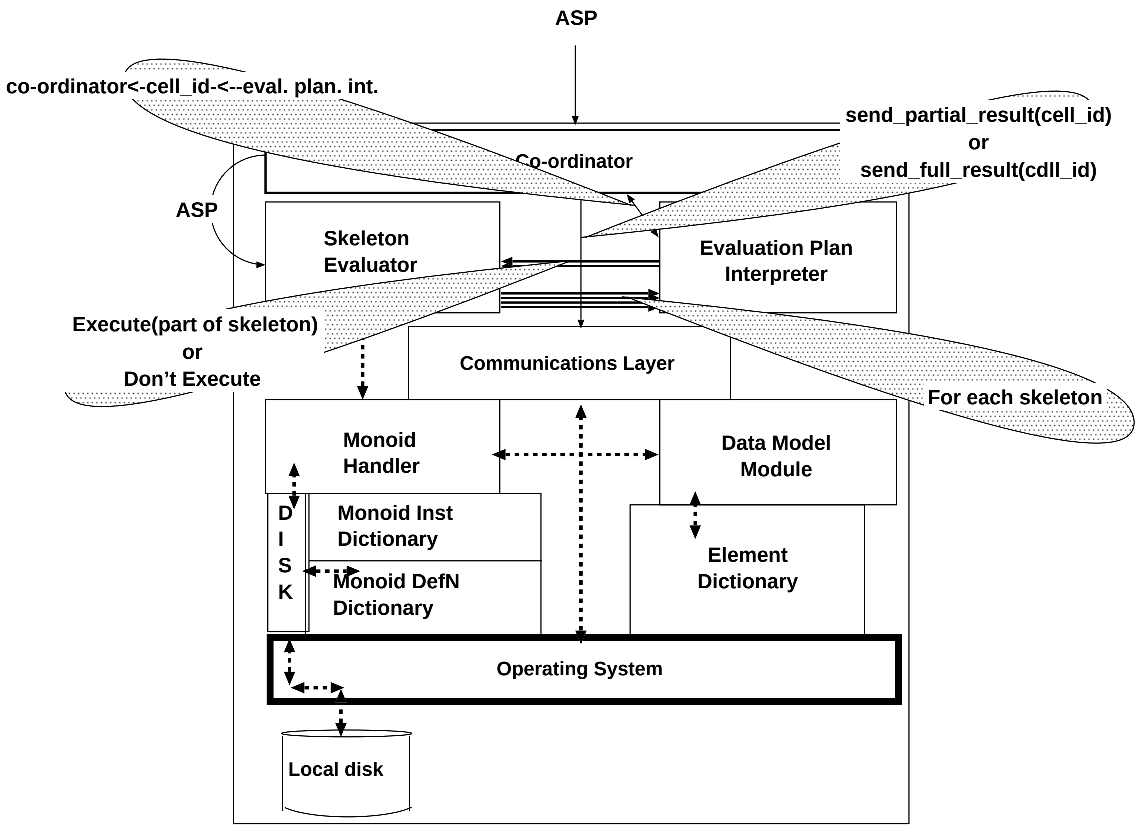


Figure 8.2: Worker Runtime Software Environment

The worker process consists of the following components (as depicted in figure 8.2):

- *evaluation plan interpreter* processes the non–functional part (VTM) of each skeleton in the ASP and determines the parallel behaviour of the skeleton.

- *skeleton evaluator* runs the appropriate part of the functional (skeleton code) component of the ASP and realises the parallel behaviour of each skeleton taking into account the process topology assigned to the skeleton by the evaluation plan interpreter.

- *co–ordinator* oversees the execution of ASPs and communication of results to destination workers as determined by the VTM accompanying each skeleton.

- *disk handler* provides a high–level interface to a local secondary storage device. Although the *disk handler* is a constituent part of the runtime system of all workers, this abstraction will have an in–memory implementation for workers that do not have access to (local) disks.

- *communications layer* handles all communications between workers and between the master and the workers.

- monoid related functions:
  - *monoid handler* provides a functional abstraction of a Monoid to the skeleton evaluator. This takes the form of a number of creation, initialisation, and cursor–based functions for manipulating Monoids. These are analogous to the cursor functions provided by a relational database management system.
  - *monoid definition dictionary* is a group of library functions which constitute the monoid functions for all systems and user–defined monoids. The definitions comprise the Monoid functions, namely the *zero* and *merge* functions for each Monoid, together with the *injection*. These functions are used to manipulate the appropriate Monoid structures.
  - *monoid instance dictionary* holds collection instance information. This is analogous to a data dictionary in a relational database management system.
- data model specific components:
  - *data model module* is a repository of data model specific *element* manipulation functions.
  - *element dictionary* retains structural information on elements.

The following sections discuss the roles of each of the main components of the worker runtime system.

8.1.2.1 Evaluation Plan Interpreter

The VTM that accompanies each skeleton is processed by the evaluation plan interpreter. The evaluation plan interpreter on each worker analyses the VTM assigned by the skeleton evaluator, and verifies the particular worker's position (if any) within the virtual process tree depending on the *allocation algorithm* used. Any one of the four algorithms presented in chapter 7 may be used. A description of each algorithm and its interpretation appears in § 7.10.3. A summary of each algorithm is provided in table 8.1. Since the structure of the process graph of each skeleton, as determined by the VPT (virtual process tree), is fixed, the mapping defined by the VTM also defines the fixed communications paths for each worker, namely, the workers from which it receives data, and those to which it sends data. Each VTM only prescribes the process graph of a single skeleton, and in so doing avoids excessive communications which can be caused by fully dynamic behaviour.

Once the skeleton interpreter completes the evaluation of one part of a skeleton, the worker may subsequently carry out a different stage of evaluation of the *same* skeleton, or it may move on to the evaluation of a *different* skeleton. For example, a node that processes data from disk may act as a result gatherer once it has finished its local processing. This occurs with a bottom level node in the VPT of a `hom` skeleton which is assigned the *reuse(*i*)* allocation algorithm.

[3]This refers to the number of workers across which data is partitioned at the bottom level of a `hom` skeleton.

Allocation Algorithm for Worker, *worker_id*
none This interpretation occurs where an instance of a skeleton is to be evaluated locally on each worker. In this case, $\forall i; i = 0$ *to* $127.\ VMT[i] = 1$, where this determines that *worker i* on the parallel machine must execute the skeleton locally. Note that $\forall j : j \neq i; VMT[j] = 0$ implies that the worker does not evaluate the skeleton.
seq(i) For the `hom` skeleton, workers are allocated from the worker_id i to worker_id *number_of_partitions*$^3 + i - 1$, in a round–robin depth-first fashion, with the leftmost leaf being allocated first. There is a 1–to–1 mapping between the processors in the virtual processor map and the processors in the physical processor map.
reuse(i) similar to the *seq(*i*)* interpretation except that the mapping between the nodes in the virtual process tree to the physical nodes is a many–to–one mapping. The total number of workers allocated for the *complete* evaluation is equal to the number of bottom level nodes in the virtual process tree for the `hom` skeleton.
total(i) map defines the 1–to–1 allocation of each virtual process tree node onto a real physical worker. The VTM value for $VTM[i]$ gives the mapping from node i in the virtual processor tree to worker i. The worker takes part if $\exists i.\ i = 0$ *to* 127 *s.t.* $VTM[i] = worker\_id$.

Table 8.1: Summary of Allocation Algorithms

8.1.2.2 Communications Layer

The communications layer implements the primitives for sending and receiving data. These primitives are used by the co-ordinator to transmit data during the evaluation of an ASP by the *skeleton evaluator.* The functions provided by the communications module include suitable *send* and *receive* primitives that incorporate the transmission of typed data. The communications layer on the transmitting cell and corresponding receiving cell will determine the nature of the communication before any data transmission commences. Further, data is marshalled at the communications layer of the sender cell and unmarshalled at the communications layer of the receiver cell.

8.1.2.3 Monoid Handler

The *monoid handler* provides an abstraction of a Monoid to the skeleton evaluator. It implements monoid manipulation operations.

The definition for each monoid includes definitions for the following functions:

- `monoid_handle*` *`zero_monoid`*`(monoid_type monoid)`

  — returns the *zero* of Monoid `monoid_type`.

- `monoid_handle*` *`merge_monoid`*`(monoid_type monoid, m_handle *m1, m_handle* m2)`

  — implements the Monoid *merge* operation for `monoid`.

- `monoid_handle*` *`unit_monoid`*`(monoid_type monoid, element_type *el_type)`

  — implements $unit_{\mathcal{M}}$, where $(S, unit_{\mathcal{M}})$ is the generator for the free Monoid $\mathcal{M}$ with underlying set S. `element_type` must exist in the element dictionary.[4]

[4] Note that the application of the $unit_{\mathcal{M}}$ function followed by the *merge* function allows the addition of single elements.

These functions use a lookup table in order to select the appropriate Monoid function for each particular `monoid`. In addition to these functions, a number of other monoid manipulation functions, including those that implement cursor manipulations, have been implemented. These are given in the next section.

8.1.2.3.1 Other Monoid Manipulation Functions

- `boolean` *`init`*`()`

  — initialise the Monoid handler. This involves reading the monoid definition dictionary, the monoid instance dictionary, and the element dictionary and processing the information within them. Returns *true* for success, *false* otherwise. May be invoked more than once.

- `monoid_instance*` *`create_collection(char *m_name, element_type el_type, boolean temp, monoid_type m_type)`*

  — create a collection of `m_type` with type of element `el_type` that is referenced by `monoid_instance`. `temp` indicates whether the monoid should be deleted next time an `init` is carried out. Note that there must be a record of element type `el_type` in the element dictionary. When the skeleton evaluator creates a partial result collection, it inserts a suitable record into the element dictionary before calling `create_collection`.

- `monoid_instance*` *`create_monoid(char *m_name, boolean temp, monoid_type m_type)`*

  — creates a Monoid of type `monoid_type` where `m_type` is a non–free Monoid. A suitable record must exist in the monoid definition dictionary for `monoid_type`.

- `monoid_handle*` *`init_monoid`*`(monoid_instance *monoid)`

  — initialise the particular Monoid denoted by `monoid_instance` so that it is ready for application of the `get_singleton` or `merge` functions. Returns a (new) `monoid_handle`.[5]

- `boolean` *`close_monoid(monoid_handle *m_handle)`*

  — closes manipulations of the Monoid referred to by `m_handle`.

- `boolean` *`delete_monoid(monoid_instance *monoid)`*

  — removes the Monoid from the database if all *`init_monoid`* calls have been terminated by a `close_monoid` call. The corresponding record in the monoid instance dictionary is removed if the `monoid` is a collection. Note that the record in the monoid element dictionary is *not* removed.

- cursor–based functions:

  - `monoid_handle*` *`get_singleton`*`(monoid_handle *m_handle)`

    — returns $\{e\}_{\mathcal{M}}$ such that $S = merge_{\mathcal{M}}(\{e\}_{\mathcal{M}}, S' \backslash \{e\}_{\mathcal{M}})$, where S represents the free Monoid referenced by `m_handle`. The precise implementation of the

[5] Since some multi–scan expressions may iterate over the same monoid_instance concurrently a unique handle issued by the disk handler is required.

operation depends on the type of the collection. For example, `get_singleton`, when applied to a list, corresponds to the list *head* function followed by the list constructor [ ].

- `element*` *`get_element`* `(monoid_handle *m_handle)`

  — given a singleton returns the sole element inside that singleton. `element` is a structure. Returns NULL if `m_handle` does not denote a singleton.

For monoids defined over $\mathbb{N}$ and $\mathbb{B}$, a suitable function is required in each case in order to allow access to the results of the Monoid operations:

- `int` *`get_val_nat`* `(monoid_handle *m_handle)`

  — returns the integer value associated with the monoid denoted by `m_handle`.

- `boolean` *`get_val_bool`* `(monoid_handle *m_handle)`

  — returns the boolean value associated with the monoid denoted by `m_handle`.

8.1.2.3.2 Data Model Module The monoid handler provides the functions that are necessary for the manipulation of Monoid structures. Functions for the manipulation of individual elements of a collection are provided by the *data model module*. The approach of grouping the data model specific functions into one logical unit helps to deliver data model independence. The *data model module* provides a data model specific interface to individual elements and is at the *same* level of abstraction as the monoid handler. The element based operations provide a means of accessing particular components of each data element. These include a function that provides the mapping from an element to a data–component, cf. relational projection:

- `f_type field_type` *`get_field_type(char *f_name, element *el)`*

  — returns the field type of the attribute.

- `val` *`apply_function(char *f_name, element *el)`*

  — returns the value, *val*, for the field (attribute) corresponding to *f_name* from the element `el`.

- `boolean` *`elements_are_equal(element *e1, element *e2)`*

  — implements structural equality. The equality function is also used by the *monoid handler* during the application of the Monoid *merge* function for Monoids in the *CIM* category, and in MQL/MHL queries for comparison between elements of a collection.

Note that the data model module functions invoke the corresponding element manipulation functions of the element dictionary.

8.1.2.4 Monoid Definition Dictionary

The *monoid definition dictionary* comprises a set of object files that represent procedural language implementations of the compiled Monoid library functions for each system and user–defined Monoid:

Monoid	**free**	**Underlying Set**	**functions**
set	$\surd$	–	zero_{set},merge_{set}, unit_{set}
bag	$\surd$	–	zero_{bag},merge_{bag}, unit_{bag}
		...	
sum	$\times$	$\mathbb{N}$	zero_{sum},merge_{sum}
		...	

The functions for each Monoid are the *zero* function, the *merge* function, and the *injection* (into the free Monoid from the underlying set) for free monoids. The underlying set of each *ground* Monoid is also recorded in the monoid definition dictionary. Since the monoid definition dictionary holds type information, the procedural language definition for the monoid functions for each collection type appears *once* in the monoid definition dictionary, irrespective of the number of instances of it. The design of the monoid handler helps to realise user–level data type extensibility by allowing procedural language definitions of monoid functions to be implemented and incorporated into the evaluation model *without* the need for re–compilation. These definitions may be dynamically linked into the monoid definition dictionary of the worker runtime system.

8.1.2.5 Monoid Instance Dictionary

The *monoid instance dictionary* holds information about each persistent collection, cf. a table in a relational database. Although the definitions for each *type* of Monoid are held in the monoid definition dictionary, a separate record is kept in the monoid instance dictionary of each collection instance:

Collection Name	**Element Type**	**Type**
cds	cd	bag
tracks	track	bag
distributors	distributor	bag
...		

This information correlates the information in the element dictionary with the information in the monoid definition dictionary since it records the instances of each Monoid.

8.1.2.6 Element Dictionary

The element dictionary holds information on the structural composition of the individual elements that comprise each resident collection. This is normally referred to as *schema–level* information in a database context. This information is currently kept as a list of typed information components for each type of element.[6] The information is accessible through data access functions:

Element Type	*Components*
cd	...
track	...
distributor	...
...	

For partial results, the skeleton evaluator creates *new* records in the element dictionary using the element creation functions provided by the element dictionary.

[6]See Appendix E for *components*.

8.1.2.7 Skeleton Evaluator

The skeleton evaluator is at the heart of the worker runtime system and performs the task of evaluating skeleton programs. This section outlines the functionality of the skeleton evaluation algorithm by example. Consider the evaluation of the skeleton program given in figure 8.3. At the root of the query lies a `pipe` skeleton that feeds the results of filtering

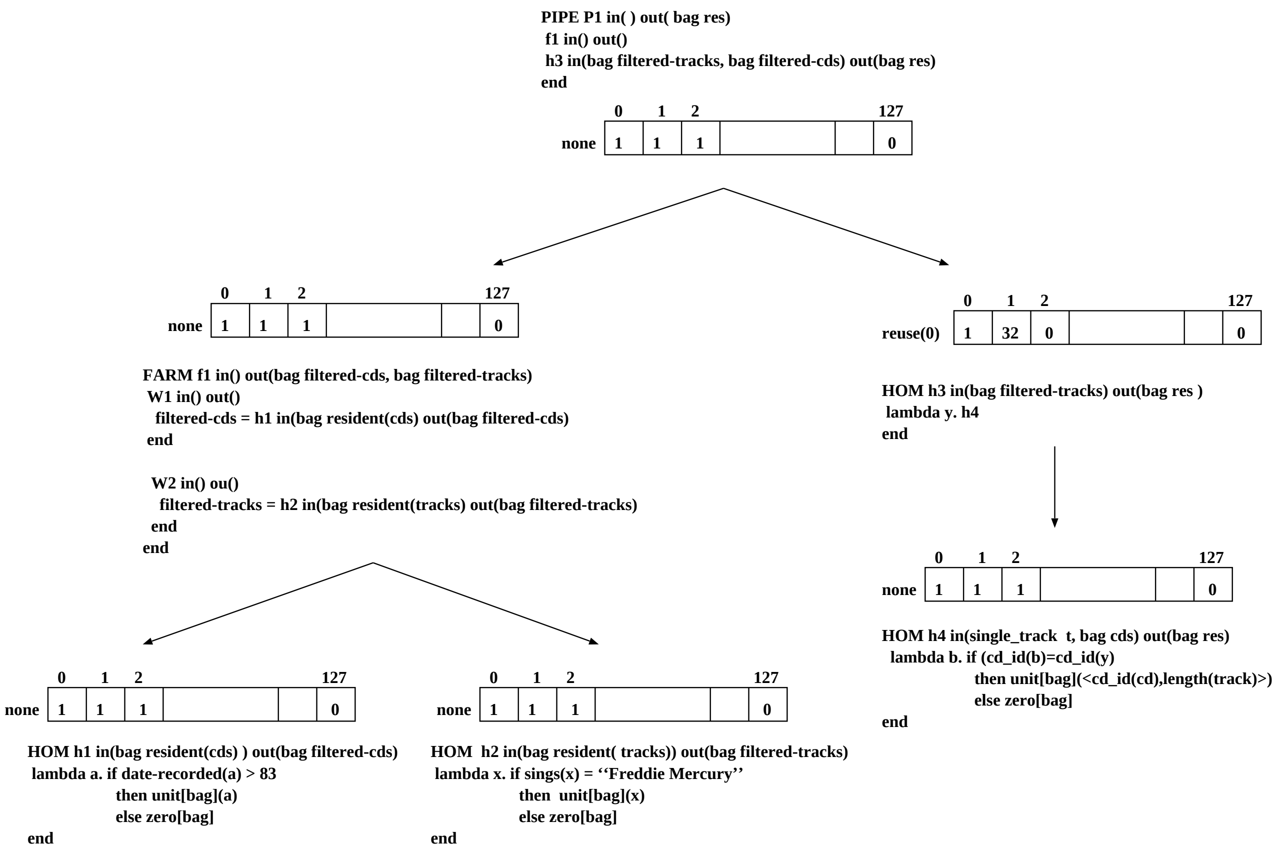


Figure 8.3: Skeleton Graph Depicting a Two–way Join

two resident collections, represented by the left hand branch, into the main evaluation, as signified by the right hand branch. The VTM associated with this skeleton determines that the skeleton is executed locally at each worker. The filtering of the resident tracks is carried out at the same time as the filtering of the resident cds by two independent `hom` operators within a `farm` skeleton. Notice that the VTM of `f1` determines that the skeleton is executed locally. This is the required behaviour since farm parallelism is not utilised in this ASP.[7] Since the results of each of these pre–filtering `hom` skeleton operations should be evaluated locally at each worker and not be amalgamated into one result, the VTMs associated with the `hom` skeletons are set with the *none* style parameter. The results of the two `hom` skeletons are fed by the `farm` and `pipe` skeleton to the nested `hom` evaluation that is indicated on the right hand side of the tree. The main evaluation is a nested–loop that is driven by `hom`, `h4`. The results from this evaluation are returned as the final result of the overall evaluation. This is evident in the style parameter of the VTM associated with `h4`.

[7]Usually, the `farm` skeleton determines a multi–processor distributed VTM.

The skeleton program depicted in figure 8.3 corresponds to the case in which data–partitioning is not required on the two argument collections. This is the case as the cds and tracks collections are already partitioned on the cd_id attribute. There are cases in which one of the partial results evaluated by `h1` and `h2` will need to be pre–filtered using elements from the other. This is depicted in the skeleton tree segment in figure 8.4 in which the VTMs are not shown. Note that `hj` probes the elements from the tracks col-

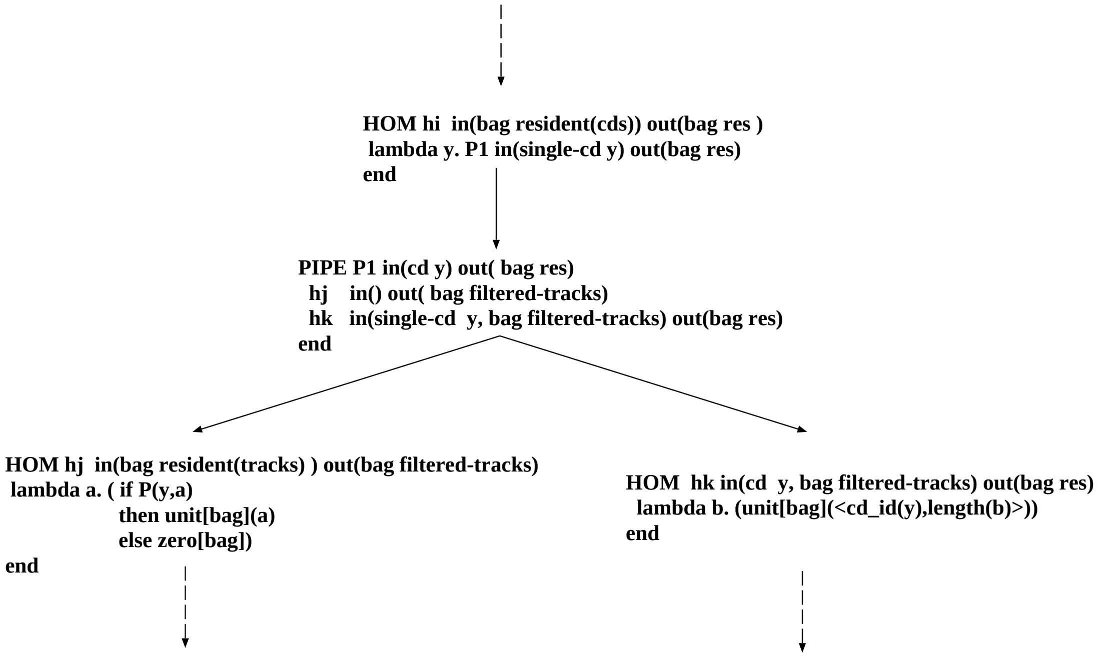


Figure 8.4: Skeleton Code Segment with a `pipe` within a Nested `hom` Expression

lection using a single element from the cds collection to form a partial result. This partial result is then *joined* with the single cd element by the *hom* `hk`. The overall evaluation is driven by `hi` in this case. This is in contrast with the skeleton program in figure 8.3 in which the evaluation is driven by `h4`. This difference in behaviour is attributable to the presence of the `pipe` skeleton.

The evaluation of the `hom` skeleton is dependent upon both its collection argument and its body. The cases in which there is a `hom` skeleton in the body of a `hom` skeleton needs to be dealt with in a different way to the cases in which there is any other skeleton. In the former case, a nested loop evaluation may be conducted from a `hom` skeleton further down from the node. The evaluation in the latter case, however, is substantially more complicated. Separate evaluations need to be carried out on the part of the skeleton that is *above* the `hom`, and the part that constitutes the body.

8.1.2.8 Efficient Implementation of Operators

The natural definition of MHL expressions outlines a nested–loop evaluation algorithm for nested homomorphisms. In order to address efficiency concerns relating to this, a more efficient formulation is given for the evaluation of nested `hom` expressions.

Uniscan relational operators (*relops*) require the data structure to be traversed only once and are hence of linear complexity with respect to the size of the input data structure. The *selection*, aggregate calculation (not grouping) and *projection* (without duplicate elimination) relops are members of the uniscan class. Uniscan relops can be parallelised easily because their input tuples can be processed independently. Pipelining can be utilised where operations produce results during their evaluation. Aggregate evaluation and sort are relops that cannot be piped.

In contrast, *multiscan* relops, such as the relation join, must scan their input several times to compute their result relation. Their complexity reflects this multiscan property. If $\mathcal{N}$ indicates the size of the input, then the complexity lies between $\mathcal{O}(\mathcal{N}\ log\ \mathcal{N})$ and $\mathcal{O}(\mathcal{N}^2)$. The input tuples cannot be processed independently. Thus, an efficient parallelisation of multiscan relops is difficult to realise. Multiscan relops either work on one data structure, for example, sort, group, duplicate elimination, or on different data structures concurrently. There exist two main methods to parallelise multiscan relational operators:

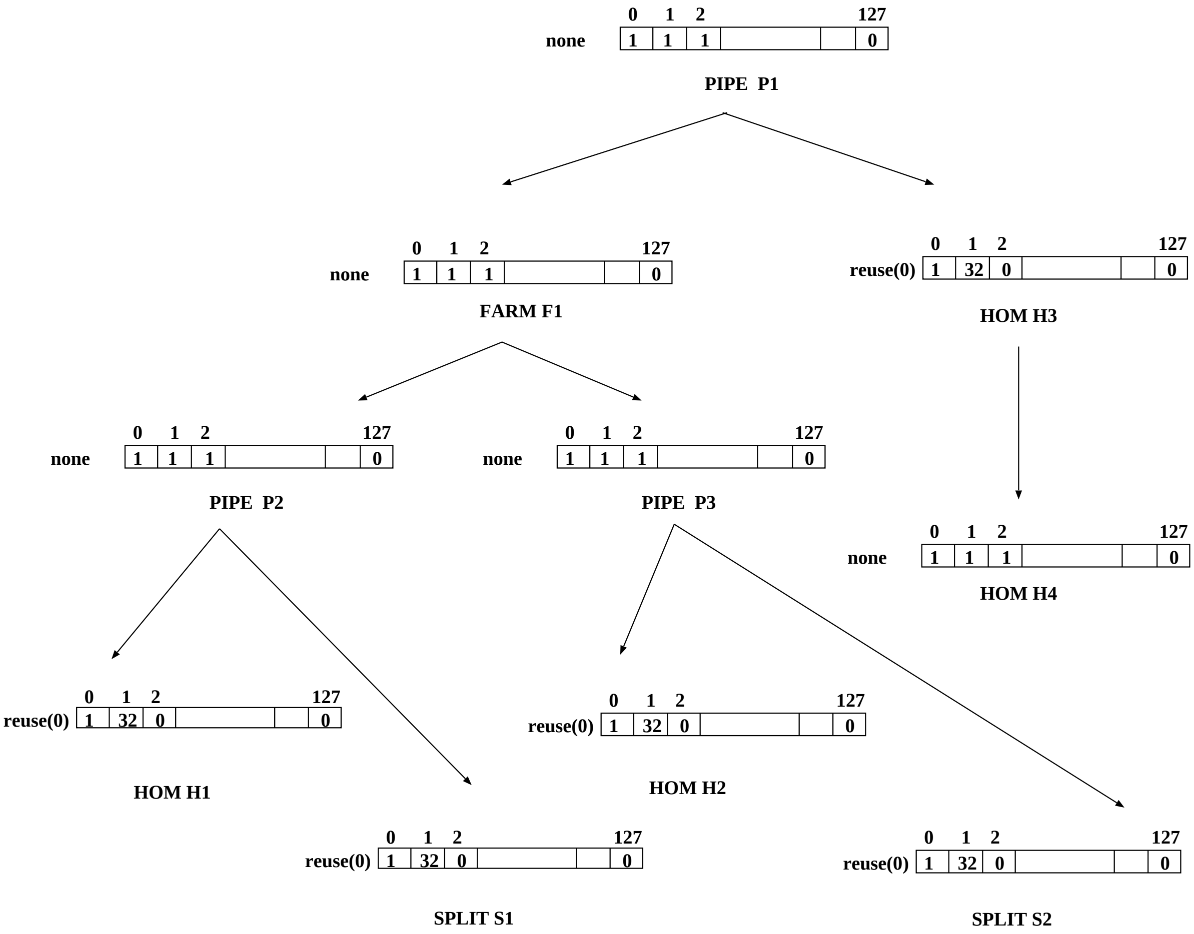


Figure 8.5: Hash–join Skeleton Tree

broadcast and *bucket-generation*. Broadcast is based on the replication of input tuples (of the smaller relation) and the replication of the multiscan relop. The larger relation is partitioned and processed on the fly when a replicate of the smaller relation is available. Bucket generation redistributes tuples independently of physical partitioning. This process results in a set of logical blocks, known as *buckets*. The generation of the buckets can be based on a hashing function or on continuous ranges which can be determined by splitting keys. Each bucket holds the tuples which need to be joined. Buckets can be processed independently. The power of this technique lies in that a join of two large relations can be reduced to the separate joins of many smaller relation buckets. Sort-merge join and hash joins are well suited to parallelised bucket-generation. Unlike broadcast, bucket generation permits pipelining within the operator tree. However, it is more susceptible to load imbalance which is due to buckets of different sizes (i.e. number of elements). Methods such as bucket size tuning or fragmentation alleviate such problems.

The skeleton model for data–intensive applications introduced in this thesis is expressive enough to model the parallel behaviour of both *broadcast* and *bucket–based* techniques.

8.1.2.8.1 The nested-loop join revisited In the *nested–loop* algorithm, each tuple of the outer collection is combined with every tuple of the inner collection. Such an algorithm may be used for the evaluation of the following MQL query, where both collections xs and ys are bags:

$$set\{join(x,y)|x \leftarrow xs, P_1(x), y \leftarrow ys, P_2(y), P_3(x,y)\}$$

To stipulate a bucked–based join algorithm within the MQL query model, assume a hash function $f : \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \mathbb{N}$ which given an element x' and an attribute att_i to hash upon, returns an integer. Further, assume that this function is applied to both elements x and y above, and that $att_i(x) = att_i(y) \Rightarrow f(x, att_i) = f(y, att_i)$; then a bucket–based join algorithm may be defined as:

$$\begin{aligned}&set\{\ join'(x',y') \mid x' \leftarrow grouped(V \times bag)\{\ \langle f(x, att_i(x)), \{\{x\}\}\rangle \mid x \leftarrow xs, P_1(x)\},\\ &\quad y' \leftarrow grouped(V \times bag)\{\ \langle f(y, att_i(y)), \{\{y\}\}\rangle \mid y \leftarrow ys, P_2(y)\},\ x'.1 = y'.1\}\end{aligned}$$

where $join'(x',y')$ is defined by $flatten(\ set\{f(x',y')|a \leftarrow x'.2, b \leftarrow y'.2\})$. Notice here that $grouped(V \times bag)$ corresponds to partitioning by hash values. Carrying out dependency-based optimisation, as proposed in [HK96], enables the factoring out of the two inner comprehension evaluations. This gives:

$$\begin{array}{ll}\textbf{\textit{let}} & qual_1 = grouped(V \times bag)\{\ \langle f(x, att_i), \{\{x\}\}\rangle \mid x \leftarrow xs, P_1(x)\}\\ & qual_2 = grouped(V \times bag)\{\ \langle f(y, att_i), \{\{y\}\}\rangle \mid y \leftarrow ys, P_2(y)\}\\ \textbf{\textit{in}} & flatten(\ set\{\ join'(x',y') \mid x' \leftarrow qual_1,\ y' \leftarrow qual_2,\ x'.1 = y'.1\}\ )\\ \textbf{\textit{end}} & \end{array}$$

The skeleton tree which is required for this evaluation is given in figure 8.5 for the case in which data is not ready–partitioned according to the join attribute(s). Inspecting the VTMs for each skeleton, it is clear that the evaluation gathers the results $qual_i$ using the two independent `hom` evaluations that are farmed out. This result is distributed to workers using the `split` skeletons `S1` and `S2` such that corresponding buckets from $qual_1$ and $qual_2$ fall onto the same physical processor. The rest of the evaluation proceeds locally at each worker with the `hom H4` driving the evaluation.

The join operation itself is implemented by the two `hom` skeletons, `H3`, `H4`, on the right–hand branch of the root. The function argument to the `split` function is used to cluster the two partial results $qual_1$ and $qual_2$ onto the corresponding physical processors. The ASP determines the mapping onto physical nodes and stipulates that the leaf nodes within the evaluation tree of each `split` skeleton are mapped to corresponding physical nodes. This ensures that the data allocation is correct for execution of the upper parts of the main skeleton tree.

Note that this efficient parallel join algorithm is expressed using a number of general skeletal forms, in contrast with DBMSs in which the QEP tree consists of specialised operators.

8.2 Prototype Implementation

This section discusses the implementation of the the parallel machine software architecture outlined in § 8.1 on a 128 node Fujitsu AP1000 at the IFPC. The features of the AP1000 relevant to this implementation are discussed in the following section. Where these features have had an impact on the prototype implementation, reference is made to the relevant module of the master or worker runtime system which is/are affected. The implementation of the master and worker runtime system on the AP1000 consists of over 32,000 lines of C–language code which is mainly distributed amongst the functional components of the worker runtime system.

8.2.1 The AP1000 and Related Implementation Issues

The Fujitsu AP1000 is a distributed memory SIMD parallel machine with 128 , 25 MHZ SPARC processor modules arranged in a 2–dimensional plane of size 8 by 16. Even though the machine is SIMD in principle, it is possible to obtain MIMD behaviour, a factor which is made use of in the prototype implementation. The external view of the machine is given in figure 8.6. The internal structure of the machine is as depicted in Figure 8.7. Each

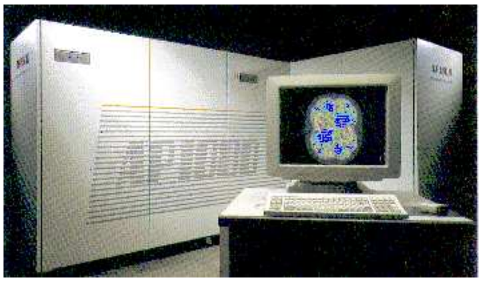


Figure 8.6: The Fujitsu AP1000 — External View

processor module, or *cell*, is equipped with a message controller, a routing controller, memory cache and a B–net interface. Thirty two cells are equipped with a 500 MByte SCSI hard disk which is connected through a SCSI controller to the integer unit on the motherboard. The maximum transfer rate from each disk is 3 MBytes/s for local transfer, although all 32 disks are accessible from the other cells for non–local data transfer. Access to the parallel machine is handled by the front–end SUN workstation, which is termed the *host*.

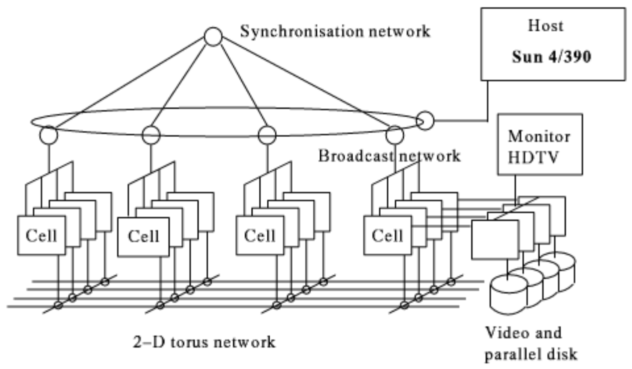


Figure 8.7: The Fujitsu AP1000 — Internal Structure

8.2.1.1 Communications Infrastructure and Support

The cells in the AP1000 are connected via three separate communications networks: the *B–net*, *S–net* and *T–net*. The choice of network used for communications is determined by the particular communications/synchronisation primitive used.

8.2.1.1.1 T–net The torus network (*T–net*) is used for point–to–point communication between cells and has a data transfer rate of 25 MBytes/s. All communication through the T–net is handled by the routing controller at each cell. This controls both routing (at the rate of 160 ns/word) and flow control. Messages on the T–net are routed in a fixed way, first in the x direction and then in the y direction. Messages that are transmitted between cells are normally transmitted using the T–net.

8.2.1.1.2 B–net The broadcast network (*B–net*) is solely for 1–to–N communication and connects the cells and the host computer. The B–net comprises hierarchical common buses with one ring network at the top level. The B–net is used for broadcasting, data distribution, and data collection and has a maximum transfer rate of 50 MBytes/s. The B–net is primarily used for the broadcast of the ASP at the start of each evaluation.

8.2.1.1.3 S–net The synchronisation network (*S–Net*) has a tree topology and is used for application–level barrier synchronisation that is provided in the native synchronisation libraries. Data is transferred serially to the root of the S–net along the tree topology. At intermediate nodes, data is merged by application of a logical conjunction until the final result is calculated at the root node. The final result is distributed to all cells along the tree. The synchronisation time on the S–net is independent of the number of cells taking part in the synchronisation.

8.2.1.2 Software Environment

Applications on the AP1000 are written in C, C++ or FORTRAN. Communication and synchronisation are implemented in the AP1000 parallel library. Also available in the parallel libraries are: *aggregation functions*, for calculation of global max/min/sum; *scatter–gather functions* for data dissemination or amalgamation at the host; *broadcast operations* for inter–cell 1–N communications; and *configuration definition* and *interrogation* functions.

8.2.1.2.1 Host Access to the parallel machine is controlled by the *caren* server which runs on the host machine. The caren server operates a queueing mechanism in which one job is dispatched to the machine at any one time. It also initialises the AP1000's hardware and sets the execution environment. A parallel application on the AP1000 consists of a host program and a set of *tasks* which execute on one or more cells that may co-operate with other usually remote, possibly different tasks. A host program is submitted to the caren server which runs the program. The host program configures the parallel execution by defining the number of cells involved in an evaluation and the creation and mapping of a number of tasks onto the cells of the parallel machine.

Related Implementation Issues The master process in the parallel machine software architecture is mapped to the host machine which controls access to the AP1000. The front–end compilation tools and the master process have been implemented as two independent processes which exchange data through files. Since the worker process has sole access to the parallel machine for the duration of its execution, this design reduces the worker process's parallel machine time. Note that the combination of the front–end compilation tools presented in chapter 7 and the master process which runs on the host machine of the AP1000 is analogous to a database server process in an implementation of a relational (or object–oriented) database management system.

8.2.1.2.2 Cell Cells run *CELLOS*, an operating system based on UNIX. There is no notion of a parallel operating system. CELLOS supports inter–task communication both for tasks on the same cell or different cells, and implements non–pre-emptive scheduling. Application–determined priority based scheduling is performed each time a task yields access to the processor, which occurs when a task engages in inter–task communication or performs a synchronisation request. There is no provision for *virtual memory* on any cell of the AP1000.[8]

Access to local (or remote) disks is handled by the distributed frame buffer task on each cell.[9] The space on each disk is partitioned into two equal–size partitions, one of which is available using the standard file manipulation libraries, with the other being assigned to the *HIDIOS* parallel file system.

Related Implementation Issues One instance of the worker process is mapped to each cell of the AP1000. The architecture and operating system support provided by the

[8] Virtual memory is available on the new generation AP3000 machine which is currently in production.

[9] This is similar to the Informix DSA servers which can retrieve meta data from any part of the system.

cells of the AP1000 faced the implementer with the following problems:

- *pre–emptive multi–tasking* — The lack of support for pre–emptive multi–tasking means that an executing *task* has sole use of the processor until it performs a kernel operation (including communications). Hence, partitioning the cell software system into suitable co–operating tasks is necessary to achieve a working implementation.

- *dynamic linking* — At the time of implementation, operating systems support for dynamic linking was not available in CELLOS. Hence, definitions for new monoids could not be dynamically linked into the monoid definition dictionary. The Monoid functions for each Monoid are currently incorporated as procedural language functions in a language library.

- *virtual memory* — The lack of virtual memory on each cell necessitates defensive programming. Significant time was spent ensuring that the developed software suffered from no memory leaks. Memory violations on a cell have the usual effect of "crashing" the cell. As far as the disk handler of the worker process is concerned, this means that all partial results are written to disk during evaluation. The disk handler of the workers which execute on the interior combining nodes of the VPT of the `hom` also write their partial results to disk as a result of this. Although an in memory-implementation would have been suitable in this case, this would have made it difficult to compare results between runs that process large amounts of data and those that process small amounts of data. Hence, the disk–based implementation was used for all evaluations, irrespective of the amount of data processed. Even with such conservative use of memory, during the prototype implementation phase, early experiments indicated that the file system would "hang" in some instances. This was attributed to file system inconsistencies. In each case, a file system re–initialisation was required to rectify the problems.

8.2.1.2.3 Communications Primitives Inter–cell communication is facilitated through message-passing. Send primitives are implemented synchronously whereas both synchronous and asynchronous implementations of receive primitives are available. A send operation defines a region in main memory together with the destination cell_id and a task_id[10] which identifies a task on cell cell_id. The destination cell_id may either be given as an absolute identifier, 0 to 127 or, a two–dimensional relative cell_id. A receive operation may either define the cell_id and task_id from which it accepts messages, or use the *ANY_CELL* wild card, indicating that a message from any cell is acceptable.

Communication is carried out using *line*–sending and *buffer*–receiving. In line–sending a message is sent directly from cache memory in a similar way to cache flushing, except that this does not invalidate the value in the cache. If the data is not available in the cache, DMA is activated. Once a message arrives at its destination, buffer–receiving moves the message from the network device to the circular buffer in main memory, without issuing interrupts to the processor. The size of the inter-cell message communication buffer on each cell of the AP1000 can be set between 64 KBytes and 512 KBytes.

[10] A *task* is similar to a process except that a task is only blocked when it performs a kernel operation.

Related Implementation Issues The communications layer of the worker runtime system implements the primitives for sending and receiving data that are used by the co-ordinator to transmit data that has been evaluated by the *skeleton evaluator*. The functions provided by the communications module include *send* and *receive* primitives that are implemented over the native libraries. These functions manage the task of marshalling and unmarshalling all transmitted data. The decision not to implement these functions over the local MPI 1.0 [mpi95] implementation is attributable to the lack of provision for *local* I/O in the MPI standard or its implementation on the AP1000.

As outlined, the size of the incoming message buffer on the cells of the AP1000 is fixed by the application programmer at compile time. For communication messages larger than this size, the receiving cell's processor is interrupted so that it may empty the message buffer. During the time that the message buffer is full, the sending cell is also blocked. If a number of cells initiate data transmission to the same destination at the *same* time, then the communications layer of the destination cell *cannot* determine a protocol for the selection of the sender cell. The underlying communications infrastructure would hence determine the cell that is chosen for transmission. Given the effects of locality, some cells may not manage to transmit their results to a destination cell for a time even if they request communications. The effects of this would become particularly acute for interior–node cells in the VPT of a `hom` operator since these would be burdened with data from all of the lower–level nodes in the VPT at the *same time*. Since the data distribution across the disk–cells attempts to ensure that there is even data distribution across cells, the same amount of data is transmitted from each lower–level cell in the evaluation of a `hom` skeleton. Even when this is not the case, for the most common queries, the data distribution is often tuned so that the amount of data returned from each cell is comparable in size. This means that lower–level cells terminate their evaluations at roughly the same time. If all lower–level cells communicate their results to their respective interior nodes at the *same* time, the receiving nodes would become flooded with data. In addition to the problems mentioned, this may also delay the merging of the incoming results, which in turn, would delay the termination. So that the communications layer on a sender cell can determine its own communication partners and in order to avoid both T–net congestion and overloading of receiving cells, and hence, a degradation in performance, a packet and capability–based high–level communications protocol has been devised and implemented to ensure fair scheduling. The devised flow–control algorithm ensures that only one cell among those requesting data transmission to a receiving cell is in possession of a (transmission) capability at any one time.[11] For some combinations of program and data distributions, there will inevitably be cases in which lower–level cells terminate their evaluations, and hence, their data transmission, at widely differing times. In such cases, the flow control algorithm does not degrade performance since is allows a transmission to continue in the absence of other transmission requests. An alternative to implementing a flow control scheme would be to stagger the termination time of the lower–level cells in such a way that they do not terminate their evaluations at the same but at increments equal to the data transmission time between the disk–cells and their receiving cells. This has not been investigated.

---

[11] This is not a token since it is tagged by the receiver with a different identifier for the specific part of the skeleton program of each transmitting cell.

8.2.2 Discussion

This section discusses design and implementation issues relating to the parallel machine software architecture that has been presented in this chapter.

The design of the architecture of the worker deserves further attention. In the design outlined in this chapter, the skeleton evaluator in conjunction with the evaluation plan interpreter determines the type of evaluation necessary for each skeleton in an ASP. Two options existed for the evaluation of an ASP:

1. *compiled solution* — The input to the master process of the parallel machine software architecture is an ASP.[12] It would be possible to substitute suitable implementation templates[13] (chapter 4) for each skeleton into the ASP, taking into account the VTM, to produce a procedural language implementation of the ASP. The output of this process could then be compiled to produce binary code for the worker processes. This code would then be broadcast to the runtime system of the workers which would dynamically load, link and execute it.

   Although a VTM is capable of expressing a range of process graphs that is adequate for the implementation of decision support queries, it may be judged to be too restrictive for complex applications which may required arbitrary process graphs. In this case, it would be equally possible to carry out skeleton code substitution at the object code generator phase (chapter 7) before VTMs are assigned to each skeleton. In this case, arbitrary process graphs could be generated for the parallel implementation of each skeleton.

   Note that a compiled solution removes the need for the skeleton evaluator and the evaluation plan interpreter of the worker runtime system. The other components of the worker runtime architecture would still be required, along with a minimal runtime system to co-ordinate the running of the program. A compiled solution is usually regarded as being efficient. However, compiled solutions lack portability.

2. *interpreted solution* — The input to the master process is an ASP which is broadcast to all worker processes. The runtime software architecture of each worker *would* include an ASP interpreter, which is not present in the runtime system of the worker in the compiled solution. Many database management systems incorporate facilities for compiling a SQL query down to a query execution plan. Hence, the interpreted solution is substantially the same as the approach of compiling queries down to a query execution plan.

   Interpreted solutions are said to lack efficiency, but are more readily ported to different parallel machines. Given that a serious concern with parallel application developers is portability, an interpreted solution seems attractive.

Option (2) was adopted for the prototype implementation. Although traditional thinking is that interpreted solutions cannot compete in speed with compiled solutions, there is a strong case for an interpreted solution. Modern interpreters for high-level procedural languages such as Java are relatively efficient.

[12] Note that an ASP resembles a *query execution plan* compiled by a DBMS.

[13] Static network of processes.

A possible combination of the two approaches would be to require that the language used in both the body of skeletons and in the skeleton implementation templates is a language for which byte code can be readily generated, such as Java or Lisp. Option (1) is initially pursued so that the implementation templates for each skeleton are substituted into the ASP. The resulting Java code could then be broadcast to the worker processes (as byte code), and executed. That is, provided the worker runtime system incorporates a suitable virtual machine. This solution would deliver portability with reasonable efficiency. For greater efficiency, using just-in-time compiler technology would enable some parts of the program to be compiled into libraries and used at runtime.

The lack of software support on the AP1000, and the restricted prototype development time, have meant that the interpreted solution that has been adopted for the prototype is not capable of interpreting all ASPs. The ASPs that are required for the experimental evaluation that is presented in chapter 9 are limited to those involving the `hom` skeleton primarily, which does not have procedural language statements in its body. Hence, a procedural language interpreter has not been implemented for executing the bodies of skeletons other than the `hom` skeleton. The skeleton evaluator incorporates no implementation for the `split` skeleton and a basic (single–worker) implementation of the `farm` and `pipe` skeletons that allows the implementation of the skeleton code segments given in figure 8.3 and figure 8.4. Although the skeletons in an ASP are normally executed in a pipelined fashion, when two skeletons, one of which pipes data through to the other, are mapped to the same physical cell, pipelining is not utilised. This can be easily remedied by making use of operating system threads on the cell to execute the two skeletons "concurrently". Note that the user does not program with threads.

8.3 Summary

This chapter has described a machine–independent software architecture and its prototype implementation on the Fujitsu AP1000 parallel computer at the IFPC. The design of the systems architecture is such that it facilitates data model independence and data type extensibility through the addition of new Monoid definitions. The architecture consisted of a master runtime system for the front–end host machine, and a worker runtime system for each node of the parallel parallel machine. The input to the master runtime system is an ASP that is output by the *object code generator*. This ASP is broadcast to all workers on the parallel machine which process the ASP at the skeleton evaluator. The result is returned to the master process.

The following chapter presents an evaluation of the decision–support architecture and its implementation using the Swiss Radio Corporation's compact disk catalogue (SR-CD).

Chapter 9

Performance Evaluation

9.1 Introduction

This chapter presents an evaluation of the run–time software architecture implemented on the AP1000 parallel computer. The implementation outlined in chapter 7 and chapter 8 enabled experiments to be carried out in a largely automated fashion. The evaluation is performed using a number of queries against the SR-CD database. The results show that near linear speedup was achieved for a range of queries.

9.1.1 Parallel Performance Metrics

The goal of parallel algorithms and systems is to obtain speedup and scaleup. Speedup considers additional hardware resources for a constant problem size, and is the extent to which more hardware can solve the same problem in less time than the original system. With added hardware, speedup holds the task constant and measures time savings. Speedup is defined by p/s for sequential execution time s and parallel execution time p. Linear speedup is the best a parallel algorithm can achieve when compared to a fastest sequential algorithm for a given number of processors: it is the ideal, although it is seldom achieved. It is attained when d times as many resources solve a constant problem in $1/d$ of the time. A relatively few sequential instructions in some parallel programs will have a limiting factor on the speedup of the program such that adding more processors may not make the program run faster. In response to this, *Amdahl's Law* measures the fraction of the sequential program, f, for which linear speedup is attained, as defined by $p = f * \frac{s}{d} + (1 - f) * s$, for sequential execution time s, parallel execution time p, and degree of parallelism d. Resolved for f, this gives $f = \frac{(s-p)}{(s-s/d)}$ which equals $\frac{1-p/s}{1-1/d}$. This may be considered to be a measure of parallelism realised. Amdahl's Law measure gives higher percentage values than speedup or scaleup since it takes into account those parts of the application that are inherently sequential as depicted in figure 9.1. This chapter uses speedup as a metric for evaluating the performance of the parallel machine implementation. In order to test for linear speedup, graphs of the inverse evaluation time, $1/time$, versus the number of processors used in the evaluation, n, are presented for both two and multi–level process configurations and different skeleton programs. Linear speedup is achieved when the points in this graph lie in a straight line — the gradient of which gives the single–cell execution time. Graphs of speedup versus the number of processors in the

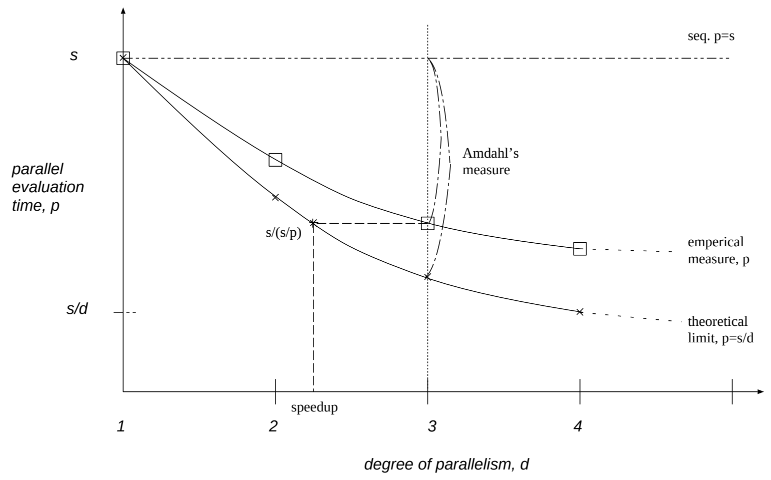


Figure 9.1: Amdahl's Measure

configuration are also presented.

Five runs were performed for each graph plot point with the median value from each of these runs being used for each plot point. Since the standard deviation of the results for each plot point was small[1], no error bars appear on the graphs. Note that some of the graphs presented in this chapter have lines drawn through the plot points. The lines between plot points do not denote interval behaviour but are drawn to enable the extraction of trends from graphs that present results pertaining to a number of queries.

9.1.2 Terms of Reference

The skeleton programs used in this evaluation are derived from SQL queries against the three tables of the SR-CD database: cds, tracks and distributors, and may be found in Appendix F. The queries mainly comprise single collection queries against the cd collection and join queries on the cds and tracks collections. Queries are categorised according to the selectivity of the predicates in the query, where each selectivity denotes the percentage of elements returned as a fraction of the total number of elements processed.

The results that are presented in this chapter are based on the *reuse(i)* § 7.10.3 mapping between the logical process graph and the physical process graph of a program. For a `hom` skeleton, the *reuse(i)* mapping allocates as many cells on the parallel machine as there are nodes at the bottom level of the process graph. In the evaluation presented in this chapter, process graphs are limited to those in which the outer `hom` skeleton uses a *reuse(i)* VTM mapping and all inner `homs` use a *none* VTM mapping (§ 7.10.3). In the presentation of the results, the form of the process graph for the outer `hom` of an evaluation is denoted by a sequence of numbers each of which is separated by an underscore. The first number denotes the number of cells at the root level, namely 1 for all configurations. Each following digit denotes the number of processors that are at each subsequent level within the process graph of the `hom`. For example, the process graph denoted by the sequence 1_2_8 has three levels with 1, 2 and 8 cells at the root, middle and lower–level respectively.

[1]The AP1000 operates in a single–user dedicated mode in which each application has overall control of the whole of the parallel machine.

Note that owing to the use of the *reuse(i)* algorithm, a 1_2_8 configuration only uses 8 cells: each intermediate node in the process graphs is mapped to one of these cells as the evaluation passes through its different stages.

9.1.3 Structure

The effects of varying the packet–size of inter-cell communications, which is set by the programmer at compile time, are explored in § 9.2. These tests are necessary since the current implementation of the machine's communications libraries does not allow the default size of communication packets to be set dynamically. The default size is fixed for the rest of the experiments. Section 9.3 presents results for runs in which the process graph of the outermost `hom` skeleton is limited to two levels: a top–level consisting of one cell which merges results from bottom–level data–processing nodes with secondary storage facilities. Section 9.3 considers the case where the total number of cells taking part in the evaluation is varied. Experiments conducted with a three level process graph are presented in § 9.4. Results for three–level configurations are presented and compared with the equivalent two–level implementations. The results from this investigation show the utility of a multi–level evaluation. Section 9.4 also considers the case where the total number of disk–cells[2] (and hence, processors) is fixed, but the number of nodes at the intermediate level in the process graph is varied. The results presented in this section are interesting in that they investigate the relationship between the number of disk–cells across which data is partitioned and the number of intermediate middle-level merging cells.

9.2 Buffer size variation

The default size of the communications packets was investigated for a number of two and three–level configurations. The results of this investigation were used to fix the size of the communications buffer for the rest of the experiments.

Graph 9.2 presents results of experiments in which the size of the communications packets is varied from 1 KBytes through to 500 KBytes for a number of 2 and 3 level configurations using a query that returns the cd_id of all records in the cds collection (the result size is 364 KBytes). The label assigned to each query is of the form:

{argument collection} {predicate selectivity} {result Monoid}*{number of attributes returned}+atts[-({result_size}KB)]

where the term in the square brackets is optional. The caption of graph 9.2 indicates that the result of the query is a bag, the selectivity of the attributes are all 100%, a single attribute is returned from each element processed, and the total result size is 364 KBytes.

For runs with configurations of 1_10 and 1_2_10, the packet size does not have a large impact on the overall evaluation time. The smallest packet sizes such as 1024 and 2048 Bytes give relatively poor performance compared to packets of larger sizes. The optimal packet size for this query, namely the one that gives the lowest overall evaluation time, is 38000 Bytes. Graph 9.3 represents results for a query that returns the total number

[2] A disk–cell denotes a cell that has access to local secondary storage. The bottom level of the VPT (virtual process tree) of the `hom` skeleton is always mapped to disk–cells.

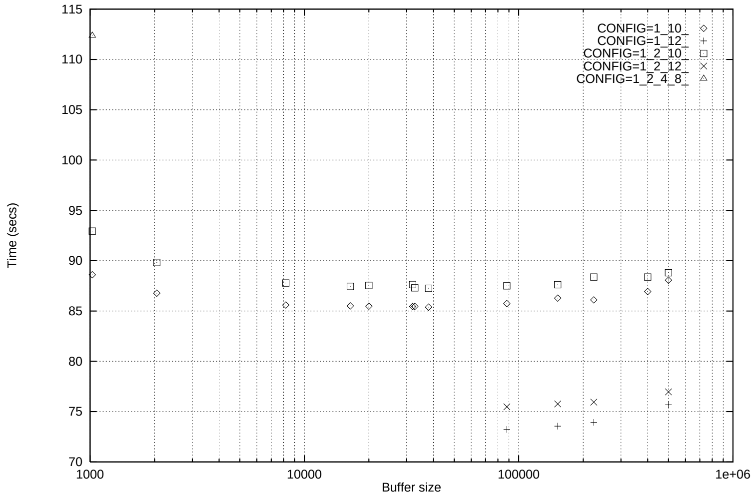


Figure 9.2: TRACKS Bag100*1+atts-(364KB)

of elements in the cds table — a small amount of data. The packet size seems to have little effect on the overall performance of this query. For a configuration of 1_2_4_20, for example, the optimal packet size for this query turns out to be 2048 Bytes. Larger packets sizes only give a performance degradation of 5% even for the two–level evaluations. This is to be expected since the major cost of transmitting small amounts of data is initialisation and synchronisation.

The size chosen for the message buffer for the rest of the experiments was set to 38000 Bytes, since packets of this size reduce the overhead of transmitting small amounts of data whilst catering for queries which return large amounts of data.

9.3 Two–level Processor Topology

This section presents results for two-level process configurations. The process graph of two–level evaluations consists of a single root–cell and a number of lower–level cells, each processing their local disk data (as depicted in figure 9.4). Runs were carried out on the cds collection with queries of selectivities ranging from 1% to 100%. The *simplified* skeleton source code for one of the queries, `CDS 100bag*3+atts`, is given below; the results are presented in graph 9.5.

```
hom in(bag resident(cds)) out(bag res)
    lambda evar1
           (if date_entered(e1) > 19920000
               then unit[bag](⟨ cd_id(evar1), total_duration(evar1),
                                date_entered(evar1)⟩)
               else zero[bag])
end

main in() out(bag res)
    h1 in() out(bag res)
end
```

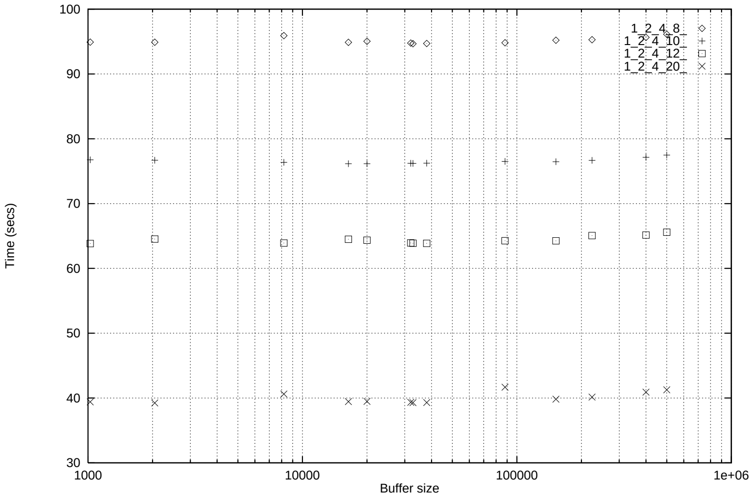


Figure 9.3: TRACKS sum100*1+atts-(4B)

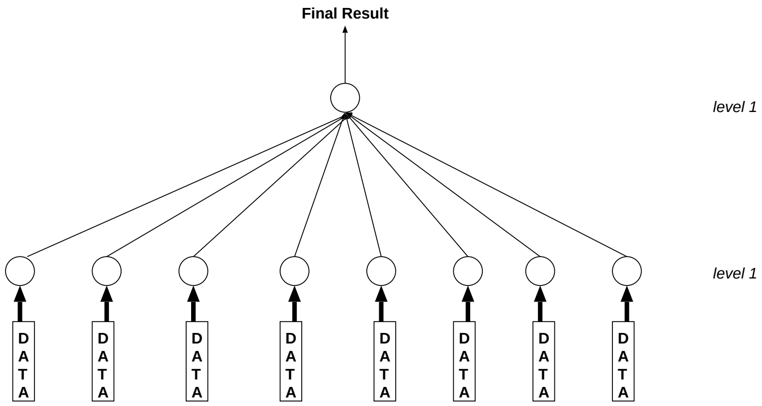


Figure 9.4: Two–level Process Graph

The type of the result, selectivity of predicate, number of attributes projected and size of the result in KBytes for each query are given on the legend of the graph. Note that the query `CDS 100bag-and-max*1+atts` evaluates two results, a bag and a maximum value, for a single iteration of its collection argument (see § 6.3 for the definition of such query forms).

All queries experience a reduction in their running times as the number of processors is increased. Since the graph shows $1/time$ on the y-axis, the query which evaluates in the least amount of time for each cell configuration is `CDS 2.98bag*1+atts`. The query that takes longest to evaluate is `CDS 100bag*3+atts` and not `CDS 2.98bag*all+atts`, as one would expect by taking into account only the raw amount of data returned by each query. The 100% selectivity query takes longer due to the greater number of disk accesses that are required to write the partial results to disk.[3] The queries with the longer execution times, for example `CDS 100bag*3+atts` and especially `CDS 1.12bag*all+atts`, show deviations from the linear relationship between the inverse execution time and number of processors, as the number

[3]Note that cells on the AP1000 have no virtual memory. Hence, the disk–handler of the cell runtime system writes partial results to disk, as necessary.

of processors is increased.

Graph 9.6 presents a speedup graph for the results presented in graph 9.5. The query which experiences the greatest speedup is the aggregate evaluation `CDS 100min*1+atts`. The query which experiences the least speedup is the query that returns the most data, namely `CDS 2.98bag*all+atts` which returns 887 KBytes of data. The query `CDS 100bag*1+atts` experiences similar speedup characteristics compared to the best performing, aggregation query. The amount of data returned by this query is significantly more than that returned by the other queries which experience similar speedups, such as `CDS 2.98bag*1+atts`, `CDS 17.3bag*1+atts`, and to a lesser extent, `CDS 153bag*1+atts`.

A breakdown of the evaluation times for the 15% and the 100% selectivity queries is shown in table 9.1. The table shows the disk–cell evaluation time and a breakdown of the root evaluation time, which consists of three components: the maximum time taken to transmit a partial result from a *remote* cell to the root, the maximum time taken for the root cell to merge a result from a remote cell into the local partial result, and the total evaluation time at the root level. Query `CDS 15.3bag*1+atts` has a higher merge cost than the 100% selectivity

Query	Disk–cell$_{average}$	Root			Total
		Rec$_{max}$	Merg$_{max}$	Total	
CDS 15.3bag*1+atts-(52KB)	20.144	0.0942	0.0161	6.086	23.800
CDS 100bag*1+atts-(112KB)	31.058	0.0705	0.0126	2.649	33.765
CDS 2.98bag*1+atts-(5KB)	42.464	0.0749	0.0481	5.587	46.762

Table 9.1: Breakdown of Evaluation Times with a Configuration of `1_24` — 1

query. Although it completes its disk–cell evaluations 35% quicker than `CDS 100bag*1+atts`, overall it is only 29.5% quicker. The maximum merge time is 0.0161 secs compared to just 0.0126 for the 100% query which deals with twice as much data. An analysis of the partial results that are produced at bottom-level cells in the evaluation of this query shows that the smallest result size is 442 Bytes whereas the largest is 6340 Bytes. The resultant performance deviations are caused by skew in the case of `CDS 15.3bag*1+atts`. Although the less selective query ought to give greater speedups considering its smaller result size, the two queries `CDS 2.98bag*1+atts` and `CDS 15.3bag*1+atts` experience similar speedups. Table 9.1 also presents a breakdown of the evaluation times for each of these queries. It shows that the lower selectivity query has a much higher maximum merge cost, namely 0.0481 secs, compared to the 100% selectivity query which has a merge cost of 0.0126 secs. However, the total root-cell evaluation time of the 2.98% selectivity query is also significantly higher than the 100% query but notably less than that of the 15% query.

This may be due to the small size of the partial results involved in the evaluation, although this is unlikely given that the query that experiences the best speedup is, in fact, the query that only returns 4 Bytes of data, `CDS 100min*1+atts`. A more likely explanation is data skew to which the more selective queries are more prone. Data skew arises when the amount of data returned from each node that processes data off secondary storage is not the same. The negative effects of skew may also be seen in the evaluation of the query `CDS 17.3bag*1+attribute` which experiences less speedup than `CDS 15.3bag*1+atts`.

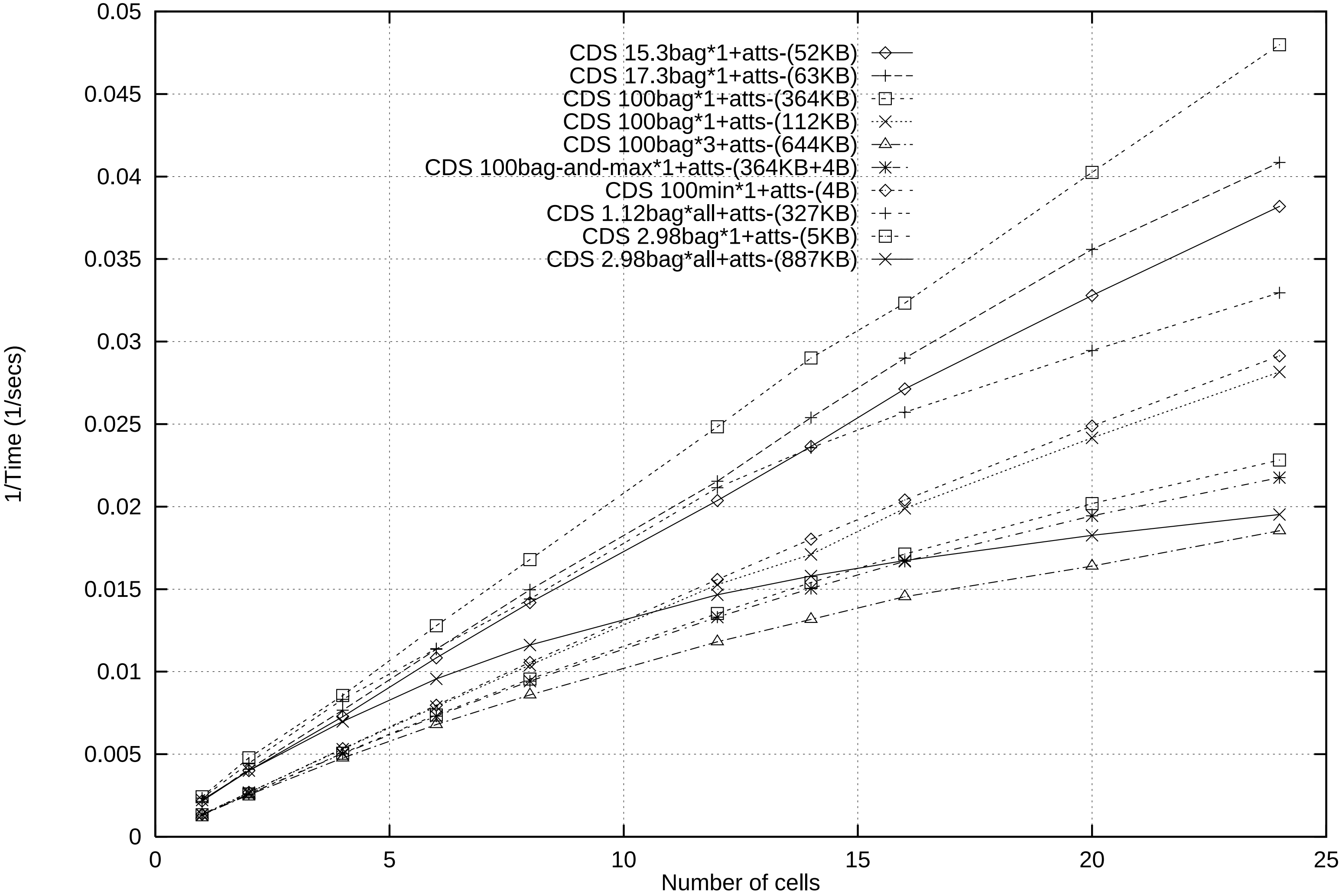


Figure 9.5: 1/t vs Processors with Configurations: 1 , 1_2 , 1_4 , 1_6 , 1_8 , 1_12 , 1_14 , 1_16 , 1_20 , 1_24

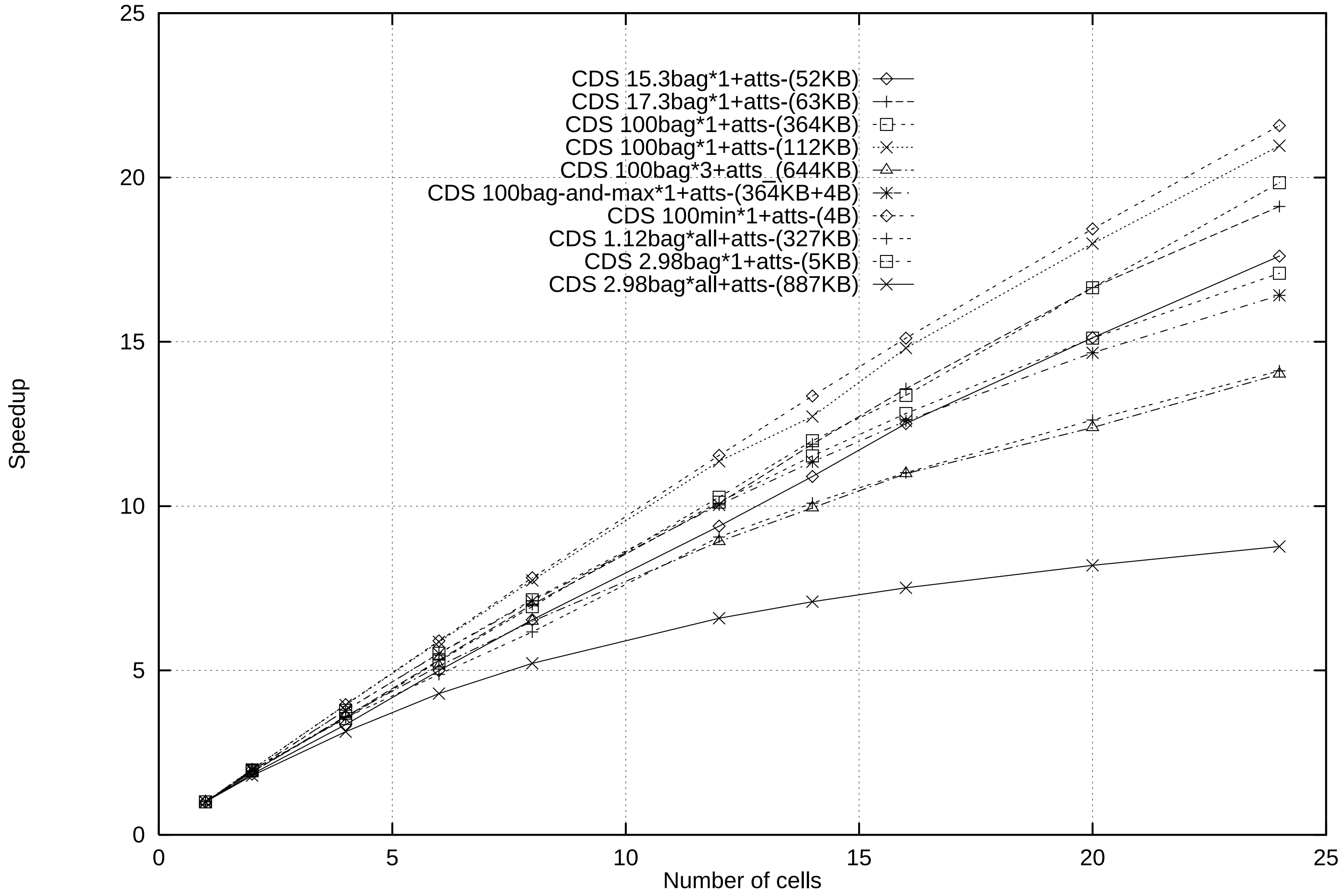


Figure 9.6: Speedup vs Number of Processors with Configurations: 1 , 1_2 , 1_4 , 1_6 , 1_8 , 1_12 , 1_14 , 1_16 , 1_20 , 1_24

Now consider the two queries `CDS 100bag-and-max*1+atts`[4] and `CDS 100bag*1+atts`. The query that returns two results one of which is a sum evaluates in similar time, and has similar speedup characteristics, when compared to the query that only returns a bag. These queries perform in a similar way since the calculation and communications overhead of the *max* evaluation over and above the evaluation of just the bag result is small. The two queries `CDS 1.12bag*all+atts` and `CDS 100bag*3+atts` which return 344KBytes and 644KBytes respectively also exhibit similar speedup characteristics. The main difference between these two queries, other than the amount of data returned, is their selectivity: one is 1% whereas the other is 100% selective. The results of further investigations on these queries are presented in table 9.2, which shows a breakdown of the execution times for these queries based on a configuration of 24 disk–cells and 1 root cell. The single disk–cell execution

Query	Disk–cells$_{max}$	Root			Total
		Rec$_{max}$	Merg$_{max}$	Total	
CDS 1.12bag*all+atts-(327KB)	20.346	1.996	1.2845	11.383	29.051
CDS 100bag*3+atts-(644KB)	31.794	0.2187	0.466	20.717	52.207
CDS 100bag*1+atts-(364KB)	31.262	0.153	0.2399	11.888	43.100

Table 9.2: Breakdown of Evaluation Times with a Configuration of `1_24` — 2

times of the two queries are not comparable. The *average* disk–cell evaluation time of the 1% selectivity query is 36.5% less than the three-attribute projection 100% selectivity `CDS 100bag*3+atts` query. However, the *average* disk–cell evaluation time for the 1% query is some 35% faster than the `CDS 100bag*3+atts` query which returns the *same* amount of data as the 1% selectivity query. The explanation for this is that the 1% selectivity query performs a smaller number of disk writes overall than the less selective query. The total merge time for the `CDS 1.12bag*all+atts` query is 30% of it's total running time whereas the total merge time for the `CDS 100bag*1+atts` query is 27% of it's total running time. However, the merge time of the `CDS 100bag*3+atts` query which returns 644 KB is only 39% of it's running time.

The more selective query, `CDS 1.12bag*all+atts`, experiences wide variance in both merge and communications times implying that the evaluation of the query experiences a significant amount of skew. The combining process takes longer in the case of the more selective query. An inspection of the partial result sizes shows that there is wide variance between the amount of data that is produced at each disk–cell: the minimum is 16 KBytes, the maximum is 90 KBytes. This is as a result of skew to which the more selective predicate is subject. This can also be seen from the maximum merge times given in the table: the maximum merge time for the 1% selective query is 1.2845 whereas that of the more selective query is 0.2399. This explains why the speedup of the `CDS 1.12bag*all+atts` query is worse than `CDS 100bag*1+atts` which returns the same amount of data. In fact, it is comparable to the `CDS 100bag*3+atts` query which returns almost twice the amount of data. Note that the 100% selectivity queries experience no skew since the *same* amount of data is returned from each disk–cell and merge–cell.

[4]See § 6.3 for the definition of such query forms. This particular query evaluates two results, a bag and a maximum value, for a single iteration of its collection argument.

9.4 Three–level Processor Topology

Only two–level process graphs[5] have been considered so far. This section explores the use of three–level process graphs using a subset of the queries used in § 9.3.

9.4.1 Fixed Number of Disk–Cells

The evaluation times for configurations with 12 and 24 disk–cells are presented in graphs 9.7 and graph 9.8 for a number of queries. Owing to the use of the *reuse(i)* VTM mapping algorithm, the total numbers of cells that take part in each of the respective evaluations are the same. Hence, the x–axis of the graph shows the number of middle–level cells involved in each three–level evaluation and not the total number of cells.

Consider the runs with 12 disk–cells first. Queries returning large amounts of data, `CDS 2.98bag*all+atts`, `CDS 100bag*3+atts`, `CDS 15.3bag*1+atts` and `CDS 17.3bag*1+atts`, perform better with a two–level configuration, `1_12`, than with any three–level configuration. Amongst the configurations with intermediate middle-level merging nodes, the configuration with only 2 middle-level merging nodes, `1_2_12`, performs best with these queries. There is a noticeable drop in performance for the other three–level configurations. The queries returning small amounts of data, `CDS 100min*1+atts` and `CDS 2.98bag*1+atts` but also `CDS 100bag*1+atts` (112 KB), experience little change in absolute performance across the 12 disk–cell configurations. This is due to the small size of both the partial results and the final results. However, for some configurations, there is a performance improvement in favour of the configurations with middle-level merging cells. This trend can be observed with the query `CDS 1,12bag*1+atts` and to a much greater extent with `CDS 100bag*1+atts` (364 KB) and `CDS 1.12bag*all+atts` which return more data.

Now consider the configurations with 24 disk–cells. Amongst these, `CDS 2.98bag*all+atts`, `CDS 15.3bag*1+atts` and `CDS 17.3bag*1+atts`, perform best with the two–level configuration of `1_24`. The queries transmitting relatively small amounts of data, `CDS 2.98bag*1+atts` and `CDS 100bag*1+atts` (112 KB), show comparable performance between two–level and three–level evaluations. As with the 12 disk–cell case, queries `CDS 100bag*1+atts` (364KB) and especially `CDS 1.12bag*all+atts`, experience significant performance gains with the configuration `1_6_24` compared to `1_24`. Queries `CDS 100bag*3+atts` and `CDS 100min*1+atts` experience lesser improvements with the three–level configuration `1_6_24` compared to `1_24`. This is not the case with any of the 12 disk–cell three–level configurations.

Now consider the respective speedups which are presented in graph 9.9 and graph 9.10. Dealing with the group having 12 disk–cells first, queries returning large amounts of data, namely `CDS 2.98bag*all+atts`, `CDS 100bag*3+atts`, `CDS 15.3bag*1+atts` and `CDS 17.3bag*1+atts`, deliver the best speedup when evaluated using configurations that have no middle–level merging cells. This observation is most noticeable with the query returning 887 KB. There is little difference in speedup across the configurations with different numbers of intermediate middle-level merging nodes. The queries returning relatively small amounts of data, `CDS 100min*1+atts` and `CDS 2.98bag*1+atts` but also `CDS 100bag*1+atts` (112 KB), experience little change in speedup between each of the 24 disk–cell configurations. However, the speedup improves with three–level configurations. This trend can be observed with the queries `CDS 100bag*1+atts` (364 KB) and particularly with `CDS 1.12bag*all+atts`.

[5] At the outermost `hom`.

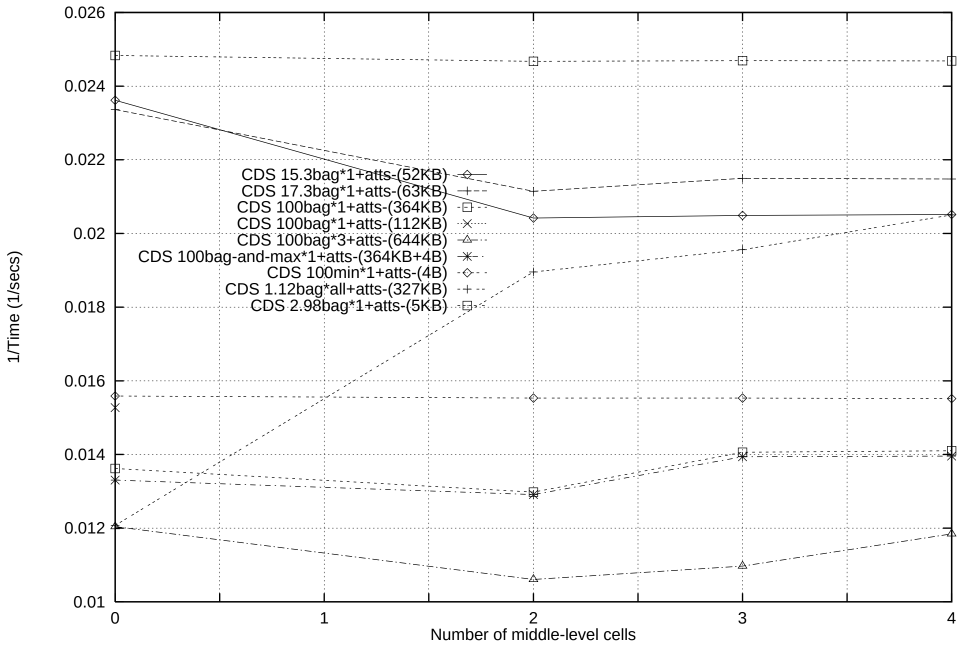


Figure 9.7: Inverse Time Graph for Configurations: 1_12, 1_2_12, 1_3_12, 1_4_12

Queries CDS 2.98bag*all+atts, CDS 15.3bag*1+atts and CDS 17.3bag*1+atts, perform best with the two–level configuration of 1_24. The queries transmitting relatively small amounts of data, CDS 2.98bag*1+atts and CDS 100bag*1+atts show the same speedups for each of the three–level configurations with 24 disk–cells. As with 12 disk–cell configurations, queries CDS 100bag*1+atts (364 KB) and CDS 1.12bag*all+atts experience improved speedups with the introduction of an intermediate level when compared to the respective one level configurations. However, CDS 100bag*3+atts and CDS 100min*1+atts experience modest improvements in speedups with a three–level configuration, in particular 1_6_24, when compared to the two–level configuration of 1_24. This is in contrast with with the three–level 12–disk–cells runs in which no three–level evaluation observed speedup improvements.

The results presented in this section show that three–level configurations can lead to performance improvements when compared to two–level configurations that have the *same* number of disk–cells. For the batch of queries and configurations used in this evaluation, the queries returning around 300 KBytes of data seem to perform particularly well with some three–level configurations. The introduction of a middle leads to a significant loss in speedup only for the query returning the largest amount of data, CDS 2.98bag*all+atts.

In the following subsections an analysis of the execution times of two queries running under two different three–level configurations is carried out.

9.4.1.1 Investigation, 1_2_24

A breakdown of the constituent components of the evaluation costs of two queries which return comparable amounts of data under a configuration of 1_2_24 is presented in table 9.3

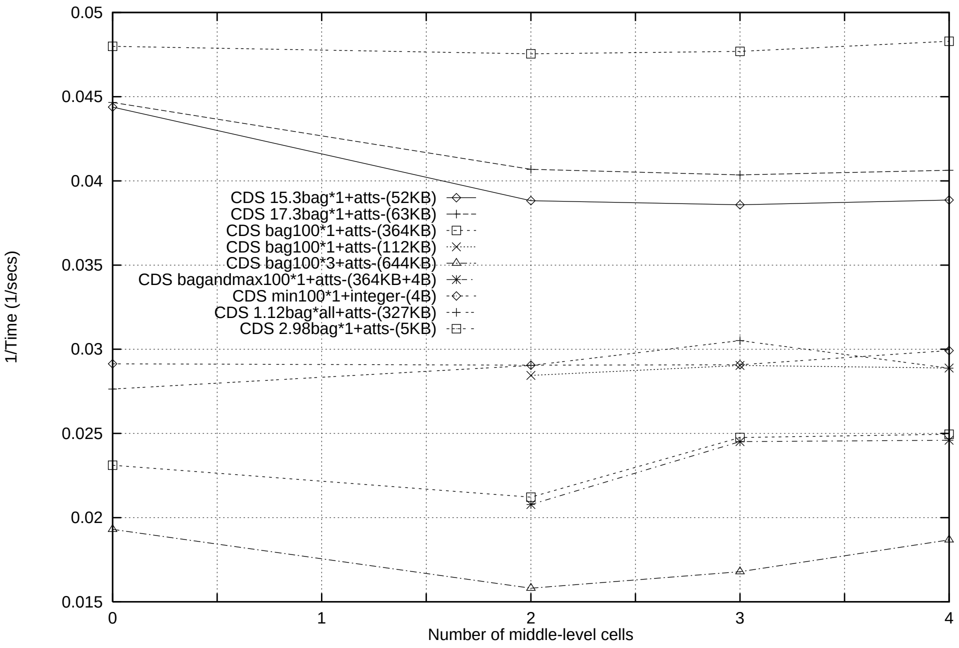


Figure 9.8: Inverse Time Graph with Configurations: `1_24`, `1_2_24`, `1_3_24`, `1_4_24`, `1_6_24`

for queries `CDS 1.12bag*all-atts` and `CDS 100bag*1+atts` (364 KB). Note that average times are given for the 100% selectivity query whereas maximum and minimum times are given for the 1% selectivity query (the maximum and minimum times are separated by a dash). Also note that owing to the use of the *reuse(i)* mapping algorithm, the intermediate-level nodes are mapped to a subset of the disk–cells, and that the root of the evaluation is mapped to one of the intermediate–level cells.

The maximum root–level and overall evaluation times presented for the 1% selectivity denote the case in which the root–cell is mapped to the intermediate–level which already has the largest data partition. The minimum root–level and overall evaluation times denote the case in which the root–cell is mapped to the intermediate–level cell that has the smallest data partition. Further, note that the middle–level execution time includes the communications time for forwarding partial results to the root cell.

Query	Disk–cell	Middle	Root Rec	Root Merg	Root Total	Total
CDS 1.12bag*all+atts-(344KB)	17.754-20.351	2.109-6.973	0.367-5.123	2.888-5.776	4.265-13.336	28.727–36.762
CDS 100bag*1+atts-(364KB)	33.119	3.834	2.638	4.412	7.410	43.305

Table 9.3: Breakdown of Evaluation Times with a Configuration of `1_2_24`

There is a notable performance difference between the two queries although they both

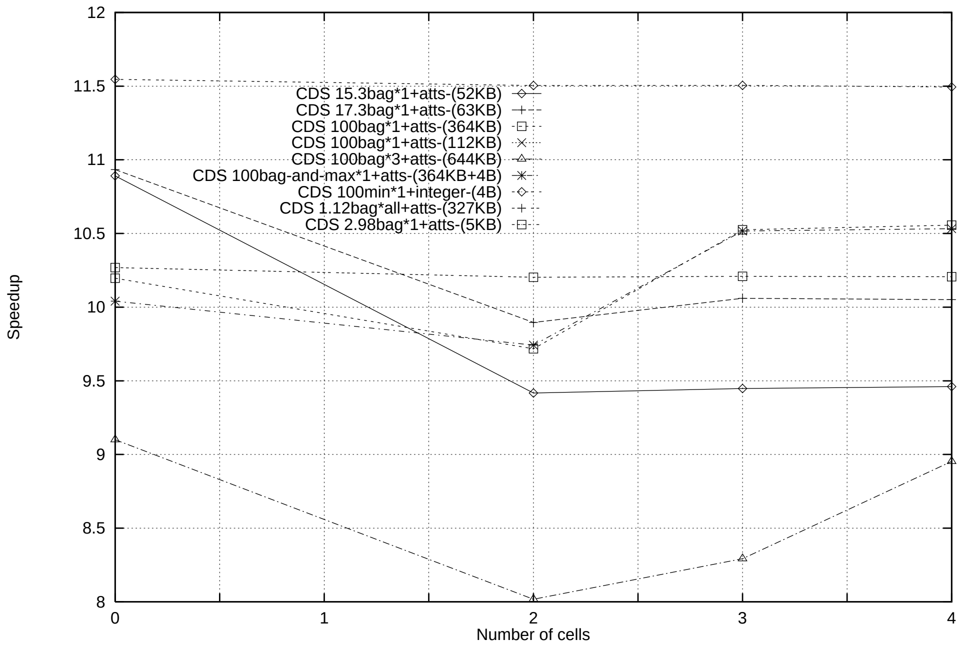


Figure 9.9: Speedup vs Number of Processors with Configurations: `1_12` , `1_2_12` , `1_3_12` , `1_4_12`

return the same amount of data. The 1% selectivity query completes in between 15% and 33% less time than the 100% selectivity query. The disk–cell evaluation times for the less selective query range from 46% to 39% quicker than the 100% selectivity query. The intermediate–level merging and data communications times also show a wide variance for the low selectivity query: with one middle-level merging cell 42% quicker and the other 80% slower than the 100% selectivity query. Inspection of a breakdown of the root–level times reveals that the root-cell has a total idle time ranging from 0 to 4.860 under the 1% selectivity query, out of a total evaluation time of between 4.265 and 13.336, whereas the less selective query has little idle time. The minimum root–level merge time for the 1% selectivity query is substantially less than that of the 100% selectivity query. This case occurs when the root–level is mapped to the intermediate–level node that hosts the largest data partition, since one of the partial results for the 1% selectivity query is much larger than the other. The converse relationship also holds. In the case of the 100% selectivity query, both partial results are the same size. The performance characteristics of the 1% selectivity query underlines the effects of data skew.

9.4.1.2 Investigation, 1_4_24

This section investigates the breakdown of the evaluation times for the two queries investigated in the previous section and the query `CDS 100bag*3+atts`. Table 9.4 shows a breakdown of the evaluation times under the configuration of `1_4_24`.

The disk–cell evaluation time of the 1% selectivity query is significantly lower than that

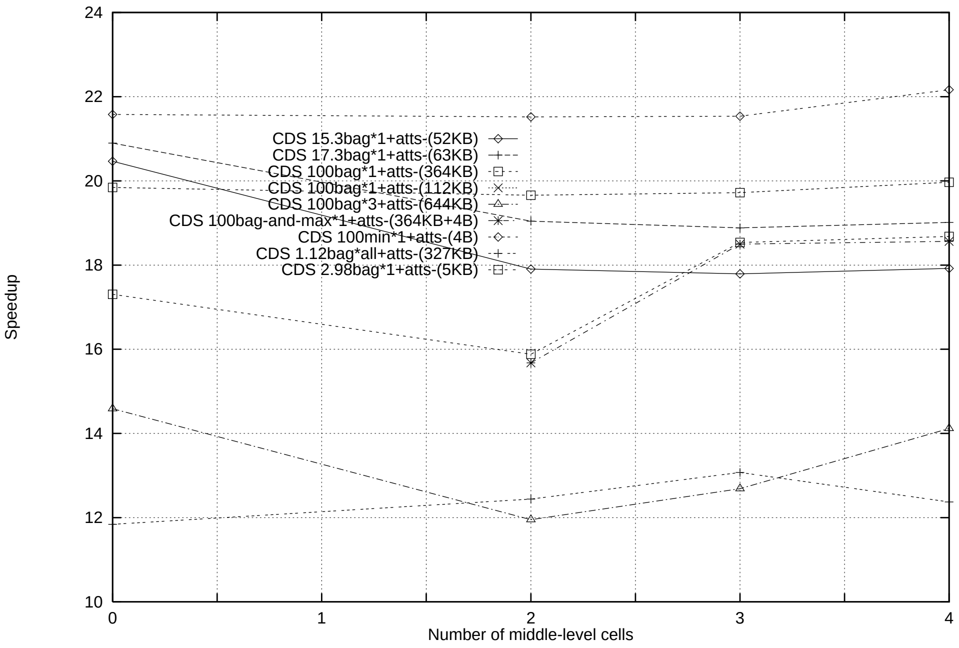


Figure 9.10: Speedup vs Number of Processors with Configurations: 1_24, 1_2_24 , 1_2_24 , 1_3_24, 1_4_24 , 1_6_24

of the 100% selectivity queries which have comparable evaluation times. This implies that the higher evaluation time of the query that return 644 KBytes compared to the one that returns 364 KBytes is due to the communications and merging costs at the intermediate and root levels. Close inspection of the intermediate–level results shows that the 1% selectivity query has the largest maximum merge time. The 1% selectivity query also has the longest maximum root–cell merge time, which is 47% greater than that of the 100% selectivity query returning 644 KBytes and up to 230% slower than that of the 100% query returning 364 KBytes. Furthermore, it also has the longest middle-level evaluation time which is some significantly slower than the 100% selectivity query returning the same amount of data. The average intermediate–level evaluation time of the 100% selectivity

Query	Disk–cell	Middle		
		Rec	Merg	Total
CDS 1.12bag*all-(344KB)	17.637-20.50	0.084-0.799	0.0125-1.199	1.036-4.658
CDS 100bag*3-(644KB)	31.794	0.199	0.198	2.247
CDS 100bag*1-(364KB)	31.561	0.1059	0.0257	1.208

Query	Root			Total
	Rec	Merg	Total	
1.12bag*all-(344KB)	0.0890-4.693	0.0690-4.195	3.834-12.586	33.243
100bag*3-(644KB)	1.545	2.850	17.271	50.923
100bag*1-(364KB)	0.74	1.267	15.996	38.422

Table 9.4: Breakdown of Evaluation Times with a Configuration of 1_4_24

query returning 644 KBytes is about double that of the query returning 364 KBytes.

The total overall execution time of the 1% selectivity is only 13% faster than the 100% selectivity query returning a similar amount of data. The communications cost incurred in transmitting partial results from the disk–cells to the middle-level merging nodes, and from the middle-level merging nodes to the root-cell, is observed to be near-linear with respect to the amount of data being transmitted for the 100% selectivity queries. However, the wide disparity between the amounts of data forwarded from each disk–cell in the case of the 1% selectivity query, which is subject to skew, make this comparison difficult.

Comparing the results gained from the investigation of a `1_2_24` configuration with a `1_4_24` configuration, it can be seen that the total execution time for the 100% selectivity query returning 364 KBytes is substantially less with `1_4_24`, whilst there is little difference in the case of the 1% selectivity query. The query returning the largest amount of data performs better with the `1_2_24` configuration.

The results presented in this section reinforce the notion that the best performance with multi–level configurations is obtained with queries that have very low data skew.

9.4.2 Varied Number of Disk–cells

This section explores configurations in which the number of disk–cells and middle-level nodes is varied. The performance and speedup graphs obtained are presented in graph 9.11 and graph 9.12. The bottom axis on each of these graphs represents the total number of

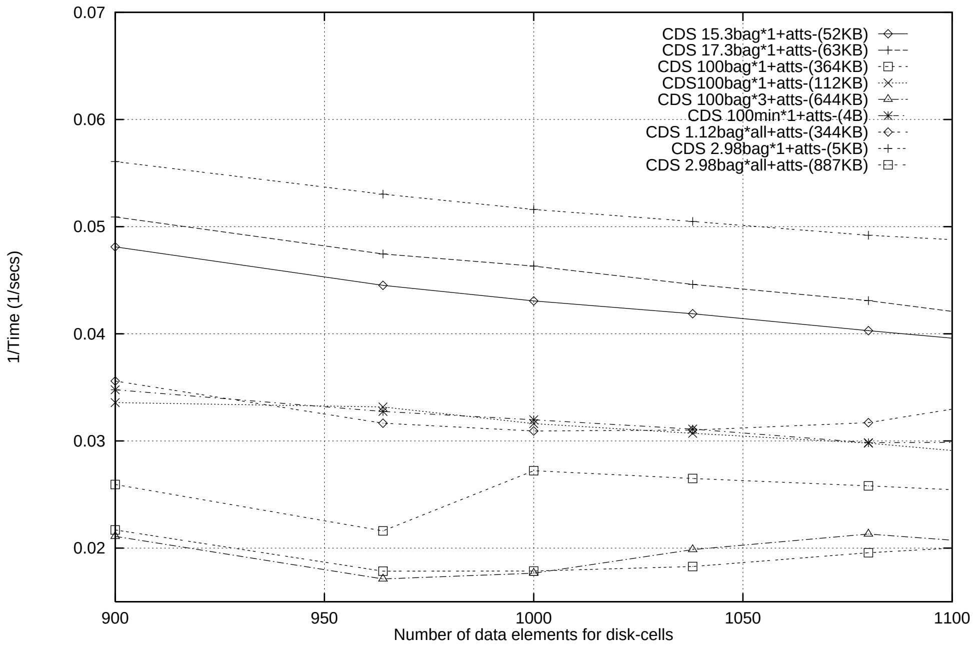


Figure 9.11: Inverse Time Graph for Configurations: `1_30 (900), 1_2_28 (964), 1_3_27 (1000), 1_4_26 (1038), 1_5_25 (1080)`

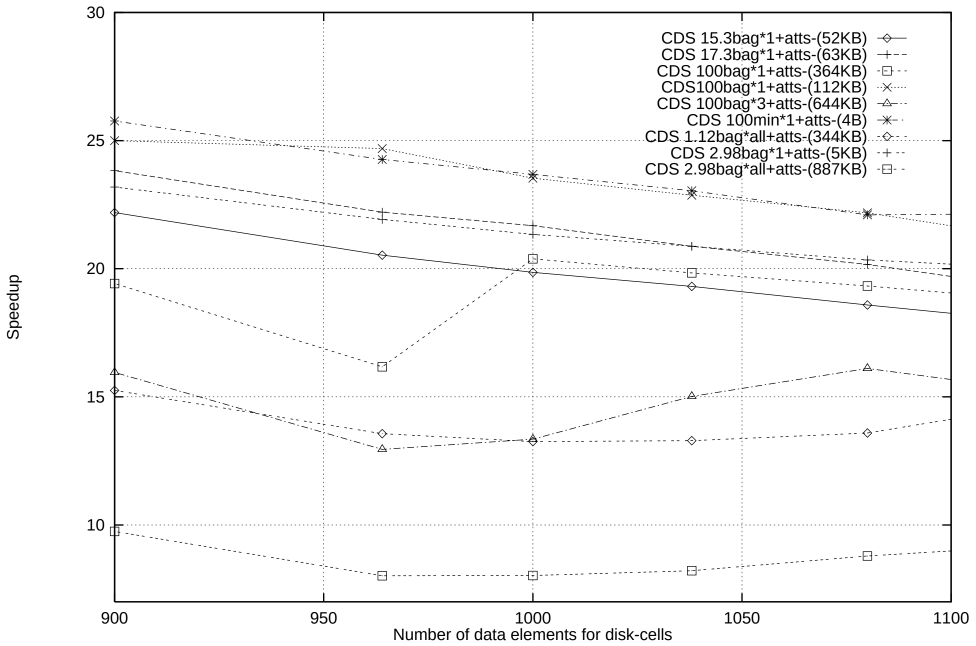


Figure 9.12: Speedup Graph Configurations:`1_30 (900), 1_2_28 (964), 1_3_27 (1000), 1_4_26 (1038), 1_5_25 (1080)`

data elements from the cds collection that each disk–cell processes in that particular configuration. For the queries returning a relatively small amount of data (up to around 100 KBytes), such as `CDS 100min*1+atts`, `CDS 17.3bag*1+atts`, `CDS 15.3bag*1+atts`, `CDS 1.12bag*all+atts` and `CDS 100bag*1+atts`, the best performing configuration is the two–level configuration that has 900 elements per disk–cells, namely `1_30`. There is a gradual drop in performance as middle level cells are added.

The queries which return the largest amount of data, `CDS 2.98bag*all-atts` and `CDS 100bag*3+atts`, experience the best performance with two–level configurations. For the query `CDS 100bag*1-+atts` (364 KB), the best performance is obtained using a configuration with 1000 elements per bottom level cell, namely, `1_3_27`. The configuration `1_5_25` performs better than the two–level configuration of `1_30`. This query shows better performance with configurations that have 26 and 27 disk–cells than it does with 30 disk–cells. This implies that the number of middle level merging nodes can reduce the overall evaluation time. The query `CDS 1.12bag*all+atts` shows similar performance characteristics with comparable performance between the two–level configuration of `1_30` and the three–level configuration of `1_5_25`. Graph 9.12 shows the speedup characteristics of the same queries. The trends from the time graph 9.11 are echoed in the speedup graph. The two queries `CDS 100bag*1+atts` (364 KB) and `CDS 1.12bag*all+atts` give improvements in speedup for the three–level configurations `1_3_27` and `1_4_26` respectively. The queries returning significantly larger amounts of data experience the best speedups with the two–level configuration `1_30`, as is the case with those returning much smaller amounts of data.

9.5 Join Queries

Parallelising join operations is complex because, for optimal parallelism, each operand collection must be declustered in the same way. If collections R and S both are declustered using the same function into n fragments on the join attribute, the operations $join(R, S)$ is then equivalent to the union of n parallel operations, $join(R_i, S_i)$ where $i = 1 \ldots n$. If each relation is not declustered in the same way, data replication or re–partitioning has to be carried out to enable the evaluation of the end result to be obtained by the union of parallel join operations.

Since the current implementation of the `split` skeleton is incomplete, joins that require data repartitioning are not possible. The skeleton interpreter (§ 8.1.2.7) carries out a nested–loop implementation that does not utilise indexes for nested `hom` expressions. Consider the case in which there are 887 and 7735 elements of the cds and tracks collections partitioned across each of 32 disk–cells respectively. Using the nested–loop strategy without indexes gives 6,860,945 loop iterations. In order to allow the gathering of performance results for the implementation of nested `hom` expressions, a subset of the overall data is used to enable runs to complete in a reasonable time. In order to keep the total number of iterations fixed, the number of elements from the cds collection is varied depending on the number of disk–cells. The number of elements from the tracks collection is fixed to 150 elements per cell, irrespective of the number of total cells or disk–cells taking part in the particular run. Hence, the total number of loop iterations is kept constant across different configurations.

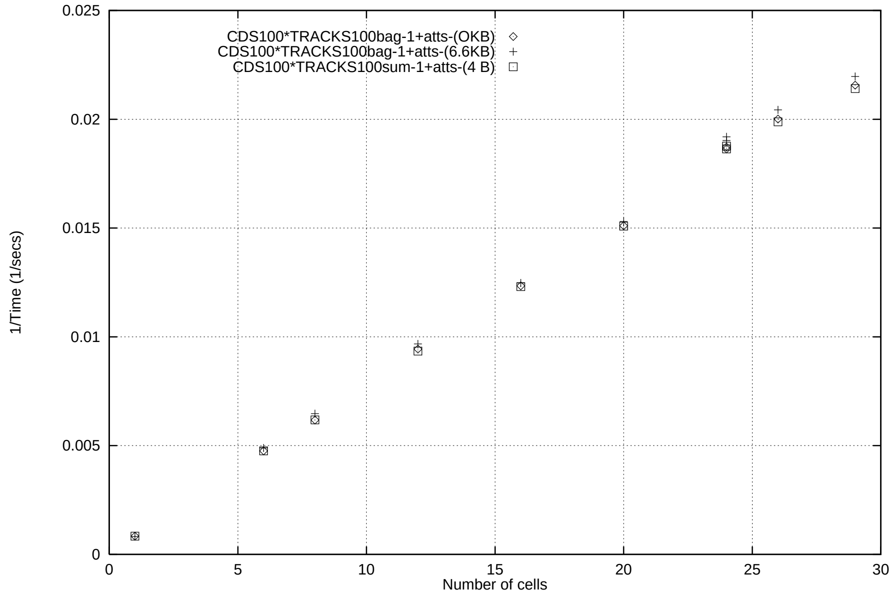


Figure 9.13: Inverse Time Graph for Join Queries

Three join queries were used in the evaluation (the skeleton code for each of these can be found in Appendix F.2). The queries involved an equijoin operation on the cd_id attribute of the tracks and the cds collection. Different predicates of 100% selectivity were applied to elements from both relations. The results for the evaluation of the join queries are presented in graph 9.13 for configurations of `1, 1_6, 1_8, 1_12, 1_16, 1_20, 1_24, 1_26, 1_29`. Graph 9.14 shows the speedup experienced by the queries. The results for the evaluation

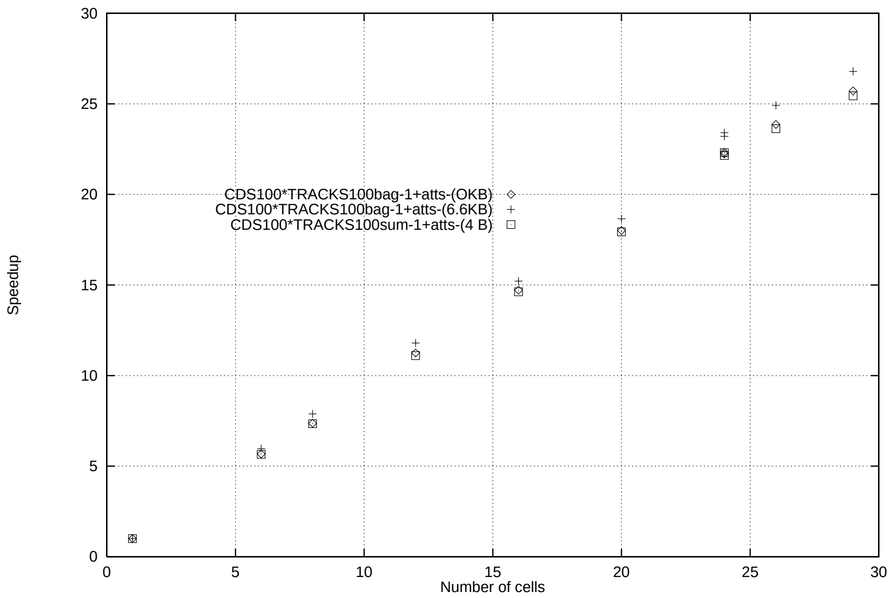


Figure 9.14: Speedup Graph for Join Queries

of join queries shows that speedup is achieved for all configurations. The query `CDS 100* TRACKS 100bag*1+atts-(6.6 KB)`[6] experiences the greatest speedup. The configuration with 16 cells experiences a speedup of 15.

9.6 Transformed Queries

This section presents performance results for a number of optimisations which are applied by the MHL rewrite system (§ 7.8). The rewrite rules appear in Appendix C. Although the optimisations are based on MHL forms, the equivalent MQL expressions are given in this section for clarity.

9.6.1 Ax 7

Ax 7 reduces an expression with a nested **hom** into a single hom. For example, the following MQL query finds the minimum date at which a cd was received at RI in the cds collection.

$$min\{date\_entered(e1) | e1 \leftarrow set\{e2 | e2 \leftarrow cds, date\_received\_RI(e2) < 19920900\}, Year\_circa(e1) = \text{"}c\text{"}\} \equiv$$

$$min\{e1 | e1 \leftarrow cds, date\_received\_RI(e1) < 19920900, Year\_circa(e1) = \text{"}c\text{"}\}$$

[6] This denotes a the join of the cd and tracks collections where the predicates on both collections are 100% selective. The query returns a single attribute per element. The total amount of data returned is 6.6KB.

The corresponding (syntactically simplified) skeleton code for both the unoptimised and the optimised forms of this query are given in code segment given in figure 9.15.

```
hom h2 in(set filtered-cds) out(min res)
 hom_lambda hom_evar 1
   (if Year_circa(hom_evar 1) eq "c"
    then Date_entered(hom_evar 1)
    else 0)
end

hom h1 in(bag resident(cds) out(set filtered-cds)
 hom_lambda hom_evar 2
   (if Date_received_RI(hom_evar 2) < 19920900
    then hom_unit[set](hom_evar 2),
    else hom_zero[set])
end

pipe p1 in() out(min res)
 h1 in(bag resident(cds)) out(set filtered-cds)
 h2 in(set filtered-cds)) out(min res)
end

main in() out(min res)
 p1 in() out(min res)
end
```

Unoptimised query

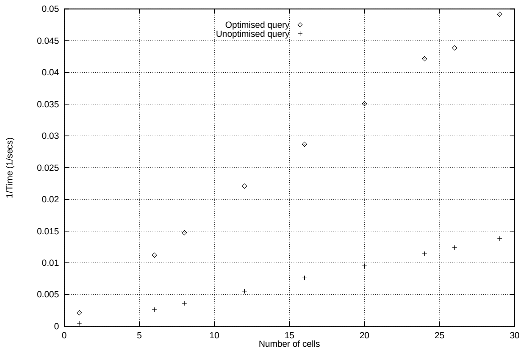
```
hom h1 in(bag resident(cds)) out(min res)
 hom_lambda hom_evar 1
   (if hom_and(Date_received_RI(hom_evar 2) < 19920900,
      year_circa(e1) = "c")
    then Date_entered(hom_evar 1)
    else 0)
end

main in() out(min res)
 h1 in() out(min res)
end
```

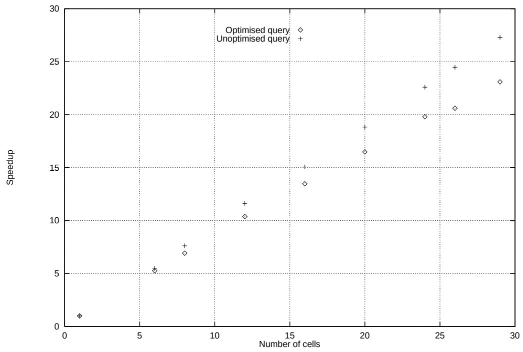
Optimised query

Figure 9.15: Unoptimised and Optimised Skeleton Programs for Ax 7

The inverse time graph and speedup graph for this query are shown in graphs 9.16. The optimised query performs significantly better than the unoptimised query for all configurations, although there is a degradation in performance which is seen with configurations that have 25 and 27 cells. The following MQL query in which the predicate on *e1* has



Inverse Time Graph



Speedup Graph

Figure 9.16: Inverse Time and Speedup Graphs for Application of Ax 7

a selectivity of 14.8% and the predicate on *e2* has a selectivity of 98% is also suitable for the application of Ax 7.

$$some\{Date\_entered(e1) > 19920000 | e1 \leftarrow bag\{e2 | e2 \leftarrow cds, Date\_received\_RI(e2) > 19910300\}\}$$

which is transformed to the following query after the application of the axiom.

$$some\{Date\_entered(e2) > 19920000 | e2 \leftarrow cds, Date\_received\_RI(e2) > 19910300\}$$

The inverse time graph for the optimised and unoptimised versions of this query is shown in graph 9.17. From the graph it can be seen that the optimised query performs better than the unoptimised query, as with the previous case.

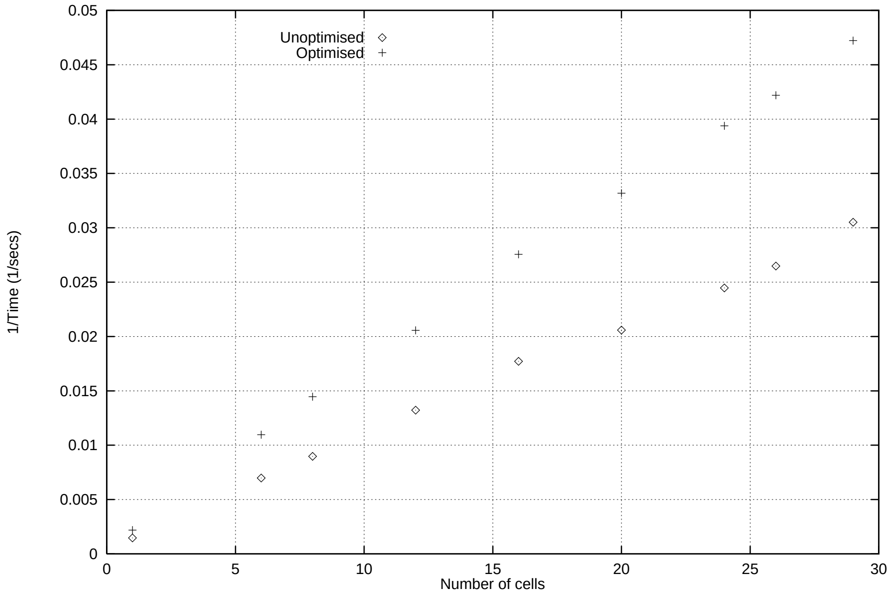


Figure 9.17: Inverse Time Graph for Application of Ax 7 – Query 2

The single–cell evaluation times for the unoptimised and optimised queries are 457.62 and 679.38 secs respectively. With a configuration of 29 disk–cells and 1 root cell, the evaluation times for unoptimised and optimised queries are 32.771 and 21.175 respectively. The total merge time for both the unoptimised and the optimised queries is the same for any configuration since the total amount of data that is merged is the same in the two cases. Hence, any performance improvement must come from a lower sequential evaluation time at the disk–cells. The sequential, single-cell, evaluation time of the unoptimised query is greater than the optimised query by an amount which is dependent on the total number of elements processed n, the cost of generating a single element G, and the cost of writing a single element as partial result W, and is approximated by $\frac{148}{1000}.n.(W+G)$.[7] Therefore, the performance improvement is linear with respect to the total number of elements processed. For the parallel evaluation, this means that as the number of cells is increased, the sequential evaluation time for all of the cells reduces by the *same* factor, thereby giving a commensurate reduction in the overall execution time. This is borne out in the results: the percentage reduction in overall execution time is comparable between the single-cell evaluation and the parallel evaluations: in the case of the single-cell evaluation, the unoptimised query is 33% slower than the unoptimised query, with a 30 cell evaluation the optimised query is 35% slower.

[7]Note that W and G are both dependent on the amount of data. Writing a whole element is more expensive than writing an attribute.

9.6.2 Ax 9

Ax 9 promotes filters in order to reduce the cost of join queries. Consider the following query in which the filter on the CDS table is 82.7% selective and the filter on the TRACKS table is 100% selective.

$$bag\{cd\_id(e1)|e1 \leftarrow CDS, e2 \leftarrow TRACKS, Date\_entered(e1) = Date\_entered(e2), Disk\_status(e1) = \text{``T''}\}$$

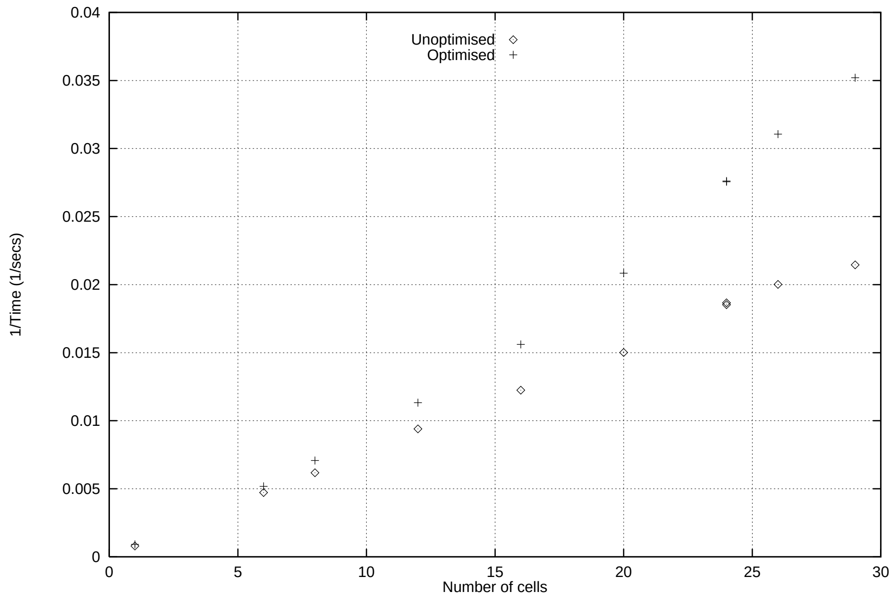


Figure 9.18: Inverse Time Graph for Application of Ax 9

Transformation Ax 9 produces an extra partial result which is produced by the application of the predicate on the cds collection:

$$bag\{cd\_id(e1)|e1 \leftarrow bag\{e1|e1 \leftarrow CDS, Disk\_status(e1) = \text{``T''}\}, e2 \leftarrow TRACKS, Date\_entered(e1) = Date\_entered(e2)\}$$

The inverse time graph for this query is figure 9.18. The single-cell sequential evaluation times of the two forms are 1223.68 and 1134.94 respectively for the unoptimised and optimised forms. The configuration with 6 disk–cells gives evaluation times of 204.145 and 198.674 respectively, whereas the 29 disk–cell configuration gives 43.031 and 42.815 secs respectively. The percentage improvement in speedup of the optimised version over the unoptimised versions are 7%, 2.8% and 0.5% respectively.

Since the merge time of this query is small due to the small size of the final result, differences in sequential disk–cell evaluation times can have a large effect on the overall result. Take the case with 6 disk–cells. The average sequential execution time for the unoptimised query is 194. That for the optimised query is 177. However, there is wide variance between the evaluation times for different runs of the optimised query with values ranging from 165 to 193.

9.7 Summary

This chapter has presented an evaluation of the run–time software architecture that has been implemented on the AP1000. The complete evaluation process of queries is carried out by the automated implementation. The results presented show speedup for queries of varying selectivities and data skew for two–level and three–level configurations. The utility of multi–level configurations has also been investigated. The results obtained show that some queries seems to perform better with a particular multi–level than they perform with an equivalent two level configuration. This is not the case for queries returning a large amount of data. The effects of skew have been investigated. As expected, the queries most prone to performance degradation as a result of skew are those of low selectivity. Performance results have also been presented for a number of optimised and unoptimised queries. Near linear speedup has been achieved for queries that do not return large amounts of data. The feasibility of the model of parallelism in an implementation has been established.

Chapter 10

Conclusions

In recent years, rapid progress in parallel hardware platforms has not been matched by the advancement of software development environments capable of exploiting their power. Developing parallel applications is a time consuming and error prone process that is restricted to applications that can be solved with acceptable response time only through parallel computation. In contrast, data–intensive applications supported by relational, and more recently, object–oriented, database technologies have experienced a faster transition to parallel computing. This has been achieved by parallel evaluation of database queries associated with the application. However, for some data–intensive applications, parallelism is only exploited within the query component whereas *both* the query and computational aspects are subject to scale, and would hence benefit from parallel evaluation.

10.1 Summary

This thesis proposes a model, an open architecture and an implementation that deliver parallelism shared between the database, system and application, thus enabling the integration of the conventionally separated query and non–query components of data–intensive applications. This is a challenge, since to compete with current database query languages the model needs to provide a declarative notation for stipulating queries, and should also be part of a general model of parallelism that is suitable for exploiting parallelism in the computational parts of data–intensive applications.

Chapter 1 outlined the research area and identified the need for a unified model that delivers parallelism across both query and non–query components of data–intensive applications. The common forms of parallelism that are utilised in parallel applications, and issues relating to parallel hardware architectures were discussed. Related work in the areas of parallel models and languages, parallelism in commercial and research database management systems, and query modelling was reviewed in Chapter 2. Many existing models for general parallelism require the programmer to deal with problem partitioning and mapping, and managing communication and synchronisation. Applications developed using languages based on such models are invariably difficult to program and are seldom portable. In contrast, database management systems in the research and commercial domains have delivered both programmability and portability through the use of a declarative language that is amenable to parallel evaluation.[1] However, existing database management systems lack an underlying model of parallelism for query evaluation. Instead, they rely on parallelism that is extracted from specific query forms. Such ad–hoc mechanisms form a poor basis for a model to support application–level parallelism.

In order to realise a unified model of parallelism for both query and non–query components of data–intensive applications, Chapter 3 identified a set of criteria that related work should be evaluated against, and identified shortcomings in current approaches. The following requirements were identified for a unified model of parallelism for data–intensive applications:

- support for parallelism across both the query and non–query parts of data–intensive applications in a unified way such that both query and non–query components have the same underlying parallel operational model.
- easy definition of the parallel behaviour of programs such that concerns of problem partitioning, mapping, and managing communication and synchronisation are reduced or eliminated.
- efficient realisability across a range of parallel architectures. This is a basic requirement for a general model if it is to compete successfully with parallel DBMSs that have been realised on a range of parallel architectures.
- amenability to performing proofs of correctness. Proving the correctness of applications that are developed using ad–hoc models of parallelism is difficult. It is easier to prove correctness properties for applications developed using abstract models that are based on theoretical principles.
- data type extensibility, such that operations over newly defined data types can be absorbed into the parallel evaluation automatically.

These areas provide the focus of later chapters.

The architectural framework presented in chapter 4 is formed from these requirements. The framework is data–model independent and takes into account the need to support both decision–support applications, which comprise queries, and complex applications, which comprise procedural language statements with embedded queries. The architecture encompasses a unified model of parallelism and the realisation of this model in the form of a language within which it is possible to describe both the query and non–query components of data–intensive applications. The language encourages the construction of parallel applications through the hierarchical composition of portable parallel constructs, termed algorithmic skeletons. Each of these parallel forms implements a flavour of task or data parallelism. Collectively, these constructs are powerful enough to model both the forms of parallelism found in the computational parts of data–intensive applications, and the forms of parallelism *currently utilised* in the parallel evaluation of database queries. Hence, queries are parallelised using standard parallel constructs as opposed to bespoke parallel implementations of individual database operations. This generality enables a tight coupling between the computational and the data–processing parts of applications

[1] SQL and OQL were not originally designed for parallel evaluation. The adoption of a language standard in each case has also contributed to portability of these database query languages.

which in turn enables the realisation of an efficient parallel implementation for the whole application.

The requirement for a declarative query notation based on a formal foundation led to chapter 6 which presented a model for queries based on theoretical principles. The paradigm of *monoid programming* was advocated as a basis for defining and modelling queries in data–intensive applications. It is based upon modelling data types as monoids and programming using the *homomorphism*, a function for which well–defined parallel operational semantics were presented. The monoid homomorphism forms the basis of the *Monoid Homomorphism Language* (MHL). Programmability was addressed through the provision of a declarative language, the *Monoid Query Language* (MQL), which is a high–level syntactic form for MHL. The monoid homomorphism, which is one of the operators in MHL is, in fact, one of the skeletons which realises the model of parallelism. Hence, it may be used both in MHL queries and in general skeleton programs. This approach gives a tight coupling between the query language and the computational language.

The way in which new data types may be added to MHL was presented with emphasis on the ease with which parallel implementations over new data types may be realised using the monoid programming paradigm. Query optimisation is an important issue in the efficient implementation of declarative languages, such as SQL and MQL. Although SQL is based on a well–defined model, Codd's relational model, it has many additional features that cannot be modelled by it. Hence, proving the correctness of SQL programs, or equivalence between alternative query forms, is difficult. This in turn makes it hard to prove the correctness of optimisations applied to query forms. Since the model presented is based on theoretical principles, it enables the correctness of many of the optimisations currently in use in DBMSs to be established. Proofs of a number of optimisations were presented to show how this may be carried out.

A sketch of the model for complex applications was outlined in chapter 5. In the language realisation of the model, complex data–intensive applications may be constructed using the composition of the algorithmic forms, one of which is the `hom` skeleton that may be used to define query actions. Query actions that are embedded in complex applications may also be defined using the declarative MQL form. Each MQL form that is embedded in a complex application is processed by the *Parallel Type Checker* and the *Object Code Generator*, constituent parts of the decision–support architecture (chapter 4). This produces a skeleton code segment that is substituted into the original application in place of the MQL query form. This substitution process is repeated for all MQL forms that are embedded in a skeleton program. The whole application is then processed according to the complex applications architecture. Notice that both the query and computational aspects of the application are translated into a single program.

A prototype implementation of the architectural framework for decision–support queries was presented in chapter 7, which outlined the design and implementation of the front–end compilation tools, and chapter 8, which outlined the back–end parallel machine architecture and its implementation. An SQL to MQL translator was implemented to enable the use of existing queries from the compact disk database (SR-CD). As part of the complete implementation of the decision–support model, an MQL query was compiled out to MHL. In order to show the applicability to the query model introduced in this thesis of the optimisations currently used in RDBMSs, a number of source–to–source rewrite rules that formulate existing database optimisations were applied to the MHL query. The resultant

MHL query was subject to a type checking process which ascertained its data partitioning requirements. The type checking process constructed a symbol table which was used in conjunction with the MHL query by the object code generator to produce an *Annotated Skeleton Program* (ASP). The skeletons in the ASP realised the functional part of the original MHL query including data (re)distribution. The annotation that accompanied each skeleton within the ASP defined the mapping between the virtual process topology and the physical process topology of the skeletal form.

The runtime software architecture consisted of a master process and a number of worker processes, both of which were presented in chapter 8. Issues relating to the mapping of this architecture onto the AP1000 parallel computer at the IFPC were discussed.

At the start of the evaluation of an ASP, the master runtime system broadcast the ASP to all workers on the parallel machine. The skeleton interpreter of the worker process on each cell processed the ASP. A worker's responsibilities with respect to the evaluation of each skeleton in the ASP were determined by the evaluation plan interpreter on each cell, which processed the annotation accompanying each skeleton. The skeleton interpreter then executed the skeleton in the way prescribed by the evaluation plan interpreter. The design arrives at a clear separation of concerns between the two distinct tasks of (i) determining which parts of a program to execute, that is carried out by the evaluation plan interpreter, and (ii) carrying out the execution itself, which is carried out by the skeleton interpreter. Data model independence was realised by an interim software layer that implemented the data model specific aspects of the manipulation of elements and collections. Data type extensibility was realised by the incorporation of procedural language definitions for monoid functions which may be compiled and dynamically linked into the monoid handler, a software module that provided a monoid abstraction to the skeleton handler. The results obtained from the experimental evaluation (chapter 9) show that the implementation delivered the performance gains expected while offering a convenient definition of the parallel environment. Near–linear speedup was achieved for a range of SQL queries from the Swiss Radio's compact disk catalogue. The design principles of the runtime architecture ensure that the software architecture can be ported to any MIMD machine without re–engineering.

Providing a high–level query interface which is compiled to constructs that implement well–defined forms of parallelism enables data–intensive queries and complex application logic to be evaluated within the same underlying parallel runtime abstraction. The programmer is saved from the burden of devising process graphs and handling inter–process communication; tasks such as these are managed automatically by the system. The parallel abstractions can be realised on a variety of platforms, thus enabling complete applications to be made portable across parallel architectures.

10.2 Future Work

Models that unify the parallelism in query and non–query components of data–intensive applications are still largely undeveloped and require the application of ideas and techniques from general models of parallelism, parallel query evaluation, and query modelling. This thesis has explored some of the techniques and technologies necessary to derive a suitable model and realise a practical system to support application–level parallelism. Some areas for future work are:

- *theoretical basis for the query model* — a central part of the query model that has been developed in this thesis is the use of the monoid homomorphism to manipulate Monoids. The model is based on the universal mapping property which guarantees that any set function from the underlying set of a free Monoid to a target Monoid is uniquely determined by a monoid homomorphism. However, a candidate Monoid to Monoid function could still be a homomorphism provided a number of equational properties are satisfied. Such a function could be used to model a query form in which an argument Monoid has not been proven to be free, or is not free. This is within the present query model. At the compilation level, such functionality could be realised by including a dictionary of built–in homomorphisms. An alternative is to incorporate user annotations confirming the homomorphic nature of Monoid to Monoid functions in which the argument Monoid has not been proven to be free. This would enable the representation and evaluation of query forms that cannot currently be modelled.

  Certain data–processing operations cannot be realised through the use of monoid homomorphisms. Further research could be carried out into extending the framework in order to incorporate the required operations at both the model and language levels. At an implementation level, a multi–level evaluation can often be realised for such query forms through the use of a combining `hom` skeleton followed by a dividing `split` skeleton. The query model could be extended to deal with this.

- *language features* — the model for complex applications currently supports the main forms of task and data parallelism. The body of the data parallel `hom` skeleton is limited to the syntax defined by MHL. In practice, the non–query procedural components of some applications may benefit from greater functionality from the combining tree data parallel form. The way in which some of the restrictions on the `hom` skeleton may be reduced, and the repercussions of this on the complex applications architecture should also be investigated.

  In general, there is limited experience of the use of skeletons. Hence, practical experience with the approach is required in order to identify the skeletons that are best suited to the forms of parallelism found in particular application areas.

- *complex applications architecture* — the prototype implementation presented in this thesis is mainly geared towards decision–support applications that comprise query components, although the decision–support and complex applications architectures share a number of data sources and functional components. The remaining aspects of the complex applications architecture could be implemented in order to better support the computational components of complex applications.

  Recent developments in multi–processor client machines constructed from standard building blocks have enabled the exploitation of cost–effective client–side parallelism in data–intensive applications. Issues should be explored relating to the way in which such client–side parallelism can be utilised alongside the server–side parallelism found on the host parallel machine.

- *rewrite system* — the current implementation of the MHL Rewriter incorporates a limited set of transformations. The present rule base could be extended to deal with a greater range of optimisations. The realisation of an efficient process graph from a skeleton program is aided by the application of optimising skeleton rewrite rules. The interaction between the MHL rewrite rules that are applied by the *MHL Rewriter* and the skeleton rewrite rules also deserves further attention.

- *support for object–oriented database queries* — the realisation of efficient parallel implementations for object–oriented database queries requires the addition of suitable optimisations to the homomorphism rewrite database.

  At the system level, the software architecture of the worker runtime enables object–oriented queries to be evaluated through the substitution of a suitable implementation of the data model module. However, in order to accommodate this at the data manipulation level, the disk handler would have to incorporate features required for object–oriented query evaluation.

- *dynamic process graph* — Only models of parallelism that limit the frequency of communication or are restricted enough to make local placement easy to compute can be efficiently implemented on a full range of target parallel computers. The static process graph that is produced as a result of the compilation process carried out on data–intensive applications fulfills this criteria by limiting communications. However, the evaluation of skeletal forms in which there is a `hom` skeleton that processes resident data can benefit from *runtime* changes to the process graph in response to load imbalances across nodes. The repercussions of this on the implementation of the individual skeletal forms and the runtime software architecture are worthy of investigation.

Appendix A

Notation

Monoid–related — *when referring to a monoid,* $T(t)$	
$zero[T(t)]$	the zero element of the monoid
$merge[T(t)]$	the merge function of the monoid
$unit[T(t)]$	injection into the free Monoid from the underlying set t
$\emptyset$	The empty collection over any set.
$[\,]$	The zero element of general sequences.
$\ominus$	The Monoid operation on general sequences
$\sim$	choose a representative sequence.
len(c)	length of a sequence c.
$@$	The Monoid operation on the *list* Monoid
$[\,]$	The zero element of the *list* Monoid
$\cup$	The Monoid operation on the *set* Monoid
$\{\}$	The zero element of the *set* Monoid
$\uplus$	The Monoid operation on the *bag* Monoid
$\{\{\,\}\}$	The zero element of the *bag* Monoid
Δ	The Monoid operation on the $grouped(\mathcal{M} \times \mathcal{N})$ Monoid
$\otimes$	The Monoid operation on the $sort(K \times V)$ Monoid
$\mathbb{N}$	the set of natural numbers
$\mathbb{B}$	the set of boolean values
$\diamond$	the end of the case within the proof
$\Box$	the end of the proof
$\circ$	function composition
$\exists^1$	there exists one such
$\Rightarrow$	implication
$\bowtie$	relational join
$t^{u/x}$	replace all occurrences of x in t with u
s.t.	such that
(* *a comment* *)	a comment

Table A.1: Notation

Appendix B

Alternative Formulation — grouping

B.1 The Grouped Monoid

Def B.1 *Given a monoid* $(\mathcal{M}, \circ, e)$*, and the monoid* $(\mathcal{N}, \star, I)$*.*
Define $grouped(\mathcal{M} \times \mathcal{N}) = (bag(\mathcal{M} \times \mathcal{N}), *, \{\{\ \}\})$

$$
\begin{array}{l}
where\ S * T \stackrel{def}{=} if\ S=\{\{\ \}\} \\
\qquad then\ T \\
\qquad else\ if\ T=\{\{\ \}\} \\
\qquad\quad then\ S \\
\qquad\quad else\ (*\ S\ \&\ T\ non\text{-}empty\ *) \\
\qquad\qquad let\ \{\{\langle x, y\rangle\}\} \uplus S' = S \\
\qquad\qquad in \\
\qquad\qquad\quad let\langle m, r\rangle\ = sf(x, T) \\
\qquad\qquad\qquad in \\
\qquad\qquad\qquad\quad let\ ns = cs(m) \\
\qquad\qquad\qquad\qquad in \\
\qquad\qquad\qquad\qquad\quad if\ m = \{\{\ \}\} \\
\qquad\qquad\qquad\qquad\qquad then\ \{\{\langle x, y\rangle\}\} \uplus S' * T \\
\qquad\qquad\qquad\qquad\qquad else\ \{\{\langle x, y \star ns\rangle\}\} \uplus S' * r \qquad \text{(LAST)}
\end{array}
$$

where:

$$
\begin{array}{l}
sf(x, T) = fold\ (\ \lambda\langle a, b\rangle.\lambda\langle m, u\rangle.\ if\ a=x \\
\qquad\qquad then\ \langle\{\{\langle a, b\rangle\}\} \uplus m, u\rangle \\
\qquad\qquad else\ \langle m, \{\{\langle a, b\rangle\}\} \uplus u\rangle\ )\ \langle\{\{\ \}\}, \{\{\ \}\}\rangle\ T
\end{array}
$$

$$cs(T) = fold\ (\ \lambda\langle -, a\rangle.\lambda temp.a \star temp\ \ )\ \{\ \}\ \ T$$

where $fold\ f\ a\ \{\{el\}\} \uplus els = fold\ f\ (f\ a\ el)\ els$*, and* $el \notin els$

Lemma B.1 $grouped(\mathcal{M} \times \mathcal{N})$ *is a Monoid.*

Proof

RTP: $*$ is associative.

namely: $S * (T * U) = (S * T) * U$

By induction on S.

Zero case: S = {{ }}

- **LHS**

  $\{\{\ \}\} * (T * U)$
  $= (T * U)$ unfold
  $= (\{\{\ \}\} * T) * U$

  $\diamond$

Inductive case:

let $S = \{\{\langle x_1, y_1\rangle\}\} \uplus S_1$
$T = \{\{\langle x_2, y_2\rangle\}\} \uplus T_1$
$U = \{\{\langle x_3, y_3\rangle\}\} \uplus U_1$

- **LHS**
  - $[\ m\ ]^1 = \{\{\ \}\}\ ]$ $\Rightarrow x_1 \notin (T * U)$
    $\{\{\langle x_1, y_y\rangle\}\} \uplus (S_1 * (T * U))$ (a)
  - $[\ M \neq \{\{\ \}\}\ ]$
    $\{\{\langle x_1, ns_1\rangle\}\} \uplus (S_1 * r)$ (b)

  Consider (a) $\Rightarrow$ by induction $= \{\{\langle x_1, y_1\rangle\}\} \uplus ((S_1 * T) * U)$
  $x_1 \notin U^2$ $= (\{\{\langle x_1, y_1\rangle\}\} \uplus (S_1 * T)) * U$ fold
  $x_1 \notin T^3$ $= ((\{\{\langle x_1, y_1\rangle\}\} \uplus S_1) * T) * U$ fold
  $= (S * T) * U$

  $\diamond$

  Consider (b) $\Rightarrow$ by induction $= \{\{\langle x_1, y_1 \star ns_1\rangle\}\} \uplus (S_i * r_1)$
  where $\langle m_1, r_1\rangle = sf(x, T * U)$ $(|r_1| < |T * U|)$
  $ns_1 = cs(m_1)$
  $m \neq \{\{\ \}\}$ $(\therefore, x \in (T * U))$

  let

  $\langle m_1, r_1\rangle = sf(x, T * U)$
  $\langle m', r\rangle = sf(x, T)$
  $\langle m'', r''\rangle = sf(x, U)$

  Lemma B.2 $r_1 = r' * r''$

  Lemma B.3 $cs(m') \star cs(m'') = ns_1$

  3 cases:

  - $r' = T \wedge r'' \neq U \Rightarrow m' = \{\{\ \}\} \wedge cs(m') = e$
    LHS
    $= \{\{\langle x_1, y_1 \star ns_1\rangle\}\} \uplus (S_1 * (r' * r''))$ Lemma B.2
    $= \{\{\langle x, y \star ns_1\rangle\}\} \uplus ((S_1 * r') * r'')$ I.H.
    $\equiv \{\{\langle x, y \star cs(m') \star cs(m'')\rangle\}\} \uplus ((S_1 * r') * r'')$ Lemma B.3
    $= \{\{\langle x, y \star cs(m'')\rangle\}\} \uplus ((S_1 * r') * r'')$ $cs(m') = e$
    $= \{\{\langle x, y\rangle\}\} \uplus ((S * r') * r'')$ $m \neq \{\{\ \}\}$, *marked* (LAST)
    $= ((\{\{\langle x, y\rangle\}\} \uplus S_1) * T) * U$
    $= (S * T) * U$

$\diamond$

– $r' \neq T \wedge r'' \neq U \Rightarrow m'' = \{\!\{\ \}\!\} \wedge cs(m'') = e$

LHS

$$
\begin{array}{ll}
\{\!\{\langle x, y \star ns_1\rangle\}\!\} \uplus (S_1 * (r' * U)) & \\
= \{\!\{\langle x, y \star ns_1\rangle\}\!\} \uplus ((S_1 * r') * U) & \text{I.H.} \\
\equiv \{\!\{\langle x_1, y \star cs(m')\rangle\}\!\} \uplus ((S_1 * r') * U) & \text{fold} \\
= (\{\!\{\langle x_1, y \star cs(m')\rangle\}\!\} \uplus S_1 * r') * U & \\
= ((\{\!\{\langle x_1, y\rangle\}\!\} \uplus S_1) * T) * U & \\
= (S * T) * U &
\end{array}
$$

$\diamond$

– $r' \neq T \wedge r'' \neq U$

LHS

$$
\begin{array}{ll}
\{\!\{\langle x_1, y_1 \star ns_1\rangle\}\!\} \uplus (S_1 * (r' * r'')) & \\
= \{\!\{\langle x_1, y_1 \star ns_1\rangle\}\!\} \uplus ((S_1 * r') * r'') & \text{I.H.} \\
\equiv \{\!\{\langle x_1, (y_1 \star cs(m')) \star cs(m'')\rangle\}\!\} \uplus ((S_1 * r') * r'') & \text{Lemma B.3} \\
= (\{\!\{\langle x_1, y_1 \star cs(m')\rangle\}\!\} \uplus (S_1 * r')) * U & \\
= (\{\!\{\langle x_1, y_1\rangle\}\!\} \uplus S_1 * T) * U & \\
= (S * T) * U &
\end{array}
$$

$\diamond$

$\square$

---

[1]Note that both m and r appear in the algorithm, as the *matches* and the *remainder*.

[2]$x_1 \notin T \times U \ \Rightarrow\ x_1 \notin U$

[3]$x_1 \notin T \times U \ \Rightarrow\ x_1 \notin T$

Appendix C

Transformations

C.1 Introduction

This appendix presents rewrites rules which are applied to MHL queries. For clarity of presentation, these optimisations are expressed as MQL equivalences.

C.2 Legend

- $\mathcal{M}_{id}$ - Monoid $\mathcal{M}_{id}$.
- $\mathcal{M}_{id}(\alpha)$ - free Monoid $\mathcal{M}$, $(\alpha, \times, u)$, modelling collection. This notation is not used for collection-valued attributes or when the generation is of the form $v_i \leftarrow unit[\mathcal{M}_{id}](a)$. Generators may only be used over free monoids.
- a *boolean–valued Monoid* is a Monoid which is defined over $\mathbb{B}$. For example, the Monoid *all*, $(\mathbb{B}, \wedge, true)$ is referred to as a boolean–valued Monoid.
- the notation $t^{u/x}$ means replace all free occurrences of x in t with u.
- att_sel_i - selection of attribute i on the element of free Monoid.
- $inv\_att\_sel(\mathcal{M}_j(\alpha), att\_sel_i)$ e - collection of items from collection $\mathcal{M}_j$ with given attribute value e for attribute att_sel_i.
- function $type\_of(db\_monoid\_instance)$ returns the type of the collection.
- a variable x is said to be *bound* in an expression e if it occurs in a sub-expression of e of the form $\lambda x.e'$; otherwise it is *free* in e. *FV(e)* denotes the set of variables with at least one free occurrence in e.
- terms of the form CT_i appear in the tail of a single comprehension and signify any number of MQL terms which may be of the form:
  - null, i.e. no term.
  - the application of a monoid homomorphism, termed a *generation* ($\leftarrow$), to:
    - a collection.

    * a comprehension term, e.g. $bag\{x|x \leftarrow cds\}$.
    * $unit[\mathcal{M}_i](a)$.
    * $merge[\mathcal{M}_i](a, b)$.

  where a and b are MQL terms.

  - a filter term of the form:
    * simple filter consisting of terms involving attribute and literal values and a comparison operator, e.g. $date\_recorded(cd) > 19920000$.
    * boolean–valued Monoid filter, e.g.
      $some\{number\_of\_disks(x) > 2|x \leftarrow cds\}$
    * complex filter consisting of a comparison operator with at least one comprehension term, e.g.
      $sum\{1|x \leftarrow tracks, cd\_id(x) = 45634, duration(x) > 10\} > 4$

- Regular expression syntax is used to indicate the number of simple filters in comprehension expressions. P_i^* indicate(s) zero or more filters. P_i^+ indicates at least one filter. A filter of the form $P_j(v_i, \ldots, v_p)$, where $p < n$ is a filter that is restricted to variables $v_i, \ldots, v_p$, where a total of n variables are instantiated in the comprehension.

- the head of comprehension is denoted by wild card r_i. Terms of the form $r_i(v_i)$ signify $FV(r_i(v_i)) = \{v_i\}$. r_i may be of the form:

  - an integer or boolean constant.
  - v_i, where v_i is a generated term within the comprehension body or an outer scope.
  - $att\_sel_i(var)$.
  - a *comprehension*.
  - $\langle comprehension\_term, ...\rangle$.
  - filter, in case of boolean–valued Monoid.

  where $r_i(v_1)$ means that the expression r_i may only use variable v_1.

Each rewrite rule presented in this chapter may be applied wherever the rule matches an expression in the head or tail of part of a query. Unless otherwise indicated, the free Monoid $\mathcal{M}_i$ does not match with monoids of the form $\langle \mathcal{M}_j, \mathcal{M}_k \rangle$. The application of each of these rules is subject to guards which are incorporated into the implementation. For brevity, they have been omitted from this appendix.

C.3 Production of Partial Results

This group of rewrite rules reduce the size of partial results which are generated during the evaluation of a query by delaying the more costly operations.

C.3.1 Qualifier Interchange

Ax 1 $\mathcal{M}_1\{\ r\ |CT_j,\ \ v_1 \leftarrow \mathcal{M}_{id}(\alpha), v_2 \leftarrow \mathcal{M}_{id}(\beta),\ CT_k\} \equiv$
$\mathcal{M}_1\{\ r\ |\ CT_j,\ v_2 \leftarrow \mathcal{M}_{id}(\beta),\ v_1 \leftarrow \mathcal{M}_{id}(\alpha),\ CT_k\ \}$

$\mathcal{M}_1\ \in\ CM$, $v_1 \notin\ FV(\mathcal{M}_{id}(\beta))$ and that $v_2 \notin\ FV(\mathcal{M}_{id}(\alpha))$.

A consequence is that any two generations which successively iterate over the same collection Monoid can commute. This is especially important for the optimisation of iteration over recursively defined collections, subject to the condition that the variable part of a generator may be used in a later qualifier, but not in an earlier qualifier. Effectively this means that when a variable is bound by an first generator and used by the next, reversing the qualifier will make that variable free.

C.3.2 Qualifier and Filter Interchange

Ax 2 $\mathcal{M}_1\{\ r\ |\ CT_i, v_1 \leftarrow \mathcal{M}_j(\alpha), P_1^+, P_2^+\ ,\ CT_j\} \equiv$
$\mathcal{M}_1\{\ r\ |CT_i, P_1^+, v_1 \leftarrow \mathcal{M}_j(\alpha), P_2^+,\ CT_j\ \}$

---

where $\mathcal{M}_1\ \in\ CM$, and $v_1 \notin FV(P_1)$.

C.3.3 Filter Promotion, Simple Filters

Ax 3 $\mathcal{M}_1\{\ r\ |\ CT_i,\ v_i \leftarrow \mathcal{M}_i(\alpha),\ CT_j,\ P_i,\ CT_k\ \} \equiv$

$\mathcal{M}_1\{\ r\ |\ CT_i,\ \ v_i \leftarrow \mathcal{M}_i(\alpha),\ P_i,\ CT_j, CT_k\ \}$

---

where CT_i is a comprehension expression, $FV(CT_j)\ \cap\ FV(P_i) = \{\}$, and P_i is a simple filter (contains no comprehensions).

Filter promotion reduces the size of partial results, offering improvements in cases in which the the filter P_i is selective and CT_j involves a expensive operations.

C.3.4 Filter Promotion, Complex Filters

Ax 4 $\mathcal{M}_1\{\ r \mid CT_i,\ \mathcal{M}_2\{\ P_{11} \mid CT_j\},\ P^{+}_{nm},\ CT_k\} \equiv$

$$\mathcal{M}_1\{r \mid CT_i,\ P^{+}_{nm},\ \mathcal{M}_2\{\ P_{11} \mid CT_j\},\ CT_k\}$$

where $\mathcal{M}_1 \in CM$, CT_i is not null and $\mathcal{M}_2$ is a boolean valued Monoid.

C.3.5 Filter Promotion – from inner to outer comprehension

Ax 5 $\mathcal{M}_1\{\ r \mid CT_i,\ v_1 \leftarrow \mathcal{M}_i(\alpha),\ CT_j,\ \mathcal{M}_2\{\ P_{11} \mid CT_k, P_{1n}(v_1),\ CT_l\},\ CT_m\} \equiv$

$$\mathcal{M}_1\{r \mid CT_i,\ v_1 \leftarrow \mathcal{M}_i(\alpha),\ CT_j,\ \ \mathcal{M}_2\{P_{11} \mid CT_k,\ CT_l\},\ \ P_{1n}(v_1),\ CT_m\}$$

where $\mathcal{M}_1 \in CM$, and for all comprehension filters $\mathcal{M}_u$ within $\mathcal{M}_2$, $\mathcal{M}_u$ is in the same category as P_{1n} if P_{1n} contains comprehensions with the range of a Monoid defined over $\mathbb{B}$.

This axiom reduces the number of times the filter is evaluated by removing it from a comprehension where there is no dependence between the filter and the variables within the comprehension (i.e. there are no variables within the filter that are bound within the comprehension).

C.4 Redundant Qualifier Elimination

Ax 6 $\mathcal{M}_1\{\ v_1 \mid v_1 \leftarrow \mathcal{M}_1(\alpha)\ \} \equiv\ \ \mathcal{M}_1(\alpha)$

Ax 7 $\mathcal{M}_1\{\ r_1 \mid v_1 \leftarrow \mathcal{M}_2\{r_2 \mid v_2 \leftarrow M_i(\alpha), P_1^*(v_2)\ \},\ P_2^*(v_1)\ \} \equiv$

$$\mathcal{M}_1\{\ r_1^{r_2(v_2)/v_1} \mid v_2 \leftarrow M_i(\alpha), P_1(v_1),\ P_2^{r_2(v_1)/v_2}(v_1)\}$$

where if P_1 is a comprehension with the range of a Monoid, $\mathcal{M}_b$ defined over $\mathbb{B}$, then $\mathcal{M}_b \in CIM$. This rule is restricted to cases in which r_2 does not contain comprehension terms since if the term r_2 is a comprehension term, then the cost of calculating this value a number of times for both r_1 and P_2 will reduce the efficiency of this transformation.

C.5 Join

C.5.1 Filter and Map Promotion through Join

Ax 8 $\mathcal{M}_1\{r_1|v_3 \leftarrow \mathcal{M}_1\{r_2|v_1 \leftarrow \mathcal{M}_i(\alpha), CT_1, v_2 \leftarrow \mathcal{M}_j(\beta), CT_2, P_i(v_1, v_2)\}, P_j^*(v_3)\} \equiv$

$$\mathcal{M}_1\{\ r_1|v_1 \leftarrow \mathcal{M}_i(\alpha), CT_1, v_2 \leftarrow \mathcal{M}_j(\beta), CT_2, P_i(v_1, v_2), P_j^{\ r_2/v_3}(v_3)\ \}$$

where $v_1 \notin FV(r_1)$, where $\mathcal{M}_1$ is limited to list, set or bag, depending on the category.

C.5.2 Partial Result Generation

Ax 9 $\mathcal{M}_1\{\ r \mid CT_1,\ \ v_i \leftarrow \mathcal{M}_i(\alpha),\ P_1^+(v_i),\ CT_2\ \} \equiv$

$$\mathcal{M}_1\{\ r \mid CT_1,\ v_i \leftarrow \mathcal{M}_i\{\ v_{ii} \mid v_{ii} \leftarrow \mathcal{M}_i(\alpha),\ P_1^+(v_i)^{v_{ii}/v_i}\ \},\ CT_2\ \}$$

where CT_1 contains at least one generation (i.e. is not null), and CT_2 contains no filters solely on variable v_i.

C.5.3 Join Generator Elimination

In the presence of a filter $P(v_2, v_3)$ at the head of the inner comprehension:

Ax 10 $\mathcal{M}_1\{\ r(v_1) \mid v_1 \leftarrow \mathcal{M}_1\{\ \ P_1(v_2, v_3) \mid v_2 \leftarrow \mathcal{M}_1(\alpha),\ P_2^*(v_2),$
$v_3 \leftarrow M_2\{v_4|v_4 \leftarrow M_2(\beta), P_3^+(v_4)\ \}, \},\quad v_1 = true\}\ \equiv$

$$\mathcal{M}_1\{\ r(v_1)^{P_1(v_2,v_3)/v_1}\ \mid\ \ v_2 \leftarrow \mathcal{M}_1(\alpha),\ P_2^*(v_2),\ v_3 \leftarrow M_2\{v_4|v_4 \leftarrow M_2(\beta), P^+(v_4)\ \},\ P_1(v_2, v_3)\ \}$$

C.5.4 Join Filter Demotion

Ax 11 $\mathcal{M}_1\{\ r\ |CT_1, v_2 \leftarrow \mathcal{M}_j\{v_{2i}|v_{2i} \leftarrow \mathcal{M}_k(\alpha), P_2^+(v_{2i}), Ps_3^+(v_1, v_{2i})\},\ CT_2\} \equiv$

$$\mathcal{M}_1\{r|CT_1,\ v_2 \leftarrow \mathcal{M}_j\{v_{2i}|v_{2i} \leftarrow \mathcal{M}_k(\alpha), P_2^+(v_{2i})\}, Ps_3(v_1, v_{2i})^{v_2/v_{2i}},\ CT_2\}$$

where Ps_3 is not a boolean–valued comprehension.

C.6 Rules for Primitive Boolean Monoids

C.6.1 Eliminating Redundant Comprehensions

Ax 12 $\mathcal{M}_1\{\ r \mid CT_i,\ v_1 \leftarrow \mathcal{M}_i(\alpha),\ CT_j,\ \mathcal{M}_2\{\ P_{11}(v_1)\ |v_j \leftarrow \mathcal{M}_j(\beta)\},\ CT_k\} \equiv$

$$\mathcal{M}_1\{r|\ CT_i,\ v_1 \leftarrow \mathcal{M}_i(\alpha),\ CT_j,\ P_{11}(v_1),\ CT_k\}$$

---

where $\mathcal{M}_2$ is the idempotent boolean–valued Monoid, *all* or *some*.

C.6.2 All

Inappropriate use of *all* or other boolean–valued monoids within comprehensions. In particular, when a collection is joined onto itself, there are usually simpler equivalent queries which evaluate to the same result.

Ax 13 $\mathcal{M}_i\{\ r\ |CT_i,\ \ v_1 \leftarrow \mathcal{M}_1(\alpha),\ \ P^*_{11}(v_1),\ \mathcal{M}_2\{\ att\_sel_i(v_1)\ RELOP\ att\_sel_i(v_2)\ |$
$v_2 \leftarrow \mathcal{M}_1(\beta)\},\ P^*_{21}(v_2),\ P_{1n}(v_1)\ \}\ \} \equiv$

$$\mathcal{M}_i\{\ r \mid v_1 \leftarrow \mathcal{M}_1(\alpha),\ P^*_{11}(v_1)\ \}$$

---

where $\mathcal{M}_2$ is the boolean-valued grounded Monoid *all*. $\mathcal{M}_i$ is max, if $RELOP$ is $\geq$, is min if $RELOP$ is $\leq$. CT_i does not contain any generation terms, including nested ones.

C.6.3 Reordering Terms within Monoids Over $\mathbb{B}$

C.6.3.1 Single Free Monoids

Ax 14 $\mathcal{M}_1\{P_1(v_1)|\ v_1 \leftarrow \mathcal{M}_1(\alpha), P_2(v_1), P^*_3(v_1)\} \equiv \mathcal{M}_1\{P_2(v_1)|\ v_1 \leftarrow \mathcal{M}_1(\alpha), P_1(v_1),\ P^*_3(v_1)\}$

---

where $\mathcal{M}_1$ is a boolean-valued Monoid.

C.6.3.2 Multiple Collections

Ax 15 $M_1\{\ r \mid CT_1, v_i \leftarrow \mathcal{M}_i(\alpha),\ CT_2,\ M_2\{\ P_1(v_n) \mid v_n \leftarrow \mathcal{M}_n(\beta),\ P^*_2(v_n),$
$P_3(v_i, v_n),\ P^+_4(v_1, \dots, v_n)\},\ CT_3\ \} \equiv$

$$M_1\{\ r \mid CT_1,\ \ v_i \leftarrow \mathcal{M}_i(\alpha),\ CT_2,\ \ M_2\{\ P_3(v_i, v_n)|\ v_n \leftarrow \mathcal{M}_n(\beta),\ P^*_2(v_n),\ P_1(v_n),\ P^+_4(v_1, \dots, v_n)\},\ CT_3\}$$

---

where $\mathcal{M}_1$ and $\mathcal{M}_2$ are boolean-valued monoids.

C.7 Optimisations based on Collection Identities

C.7.1 General Collections

Ax 16 $\mathcal{M}_i\{\ r(v_1) \mid v_1 \leftarrow merge[\mathcal{M}_i](\mathcal{M}_1(\alpha), S_2)\ \} \equiv$

$merge[\mathcal{M}_i](\mathcal{M}_i\{\ r(v_1) \mid v_1 \leftarrow \mathcal{M}_1(\alpha)\ \}, \mathcal{M}_i\{\ r(v_1) \mid v_1 \leftarrow S_2\ \})$

Ax 17 $\mathcal{M}_i\{\ r(v_1)\ |v_1 \leftarrow merge[\mathcal{M}_j](\mathcal{M}_1(\alpha), S_2),\ P_1^*(v_1)\} \equiv$

$merge[\mathcal{M}_i](\mathcal{M}_i\{\ r(v_1)\ |v_1 \leftarrow \mathcal{M}_1(\alpha),\ Ps_1(v_1)\}, \mathcal{M}_i\{\ r(v_1)\ |v_1 \leftarrow S_2,\ P_1^*(v_1)\})$

Ax 18 $merge[\mathcal{M}_i](\mathcal{M}_i\{r_1|v_1 \leftarrow \mathcal{M}_j(\alpha)\}, \mathcal{M}_i\{r_2|v_2 \leftarrow \mathcal{M}_j(\alpha)\})$

$\equiv \mathcal{M}_i\{\ merge[\mathcal{M}_i](unit[\mathcal{M}_i](r_1), unit[\mathcal{M}_i](r_2)) \mid v_1 \leftarrow \mathcal{M}_j(\alpha)\ \}$

C.7.2 Set Identities

C.7.2.1 with $-$

$e\ \in\ (\mathcal{M}_1(\alpha)\ -\mathcal{M}_2(\alpha))\ \equiv\ (e\ \in\ \mathcal{M}_1(\alpha))\ \wedge\ not(e\ \in \mathcal{M}_2(\alpha))$

The comprehension equivalent of this is:

Ax 19 $some\{\ e = v_1\ |v_1 \leftarrow set\{v_2|v_2 \leftarrow \mathcal{M}_1(\alpha),\ all\{v_2 \neq v_3|v_3 \leftarrow \mathcal{M}_2(\alpha)\}\}\}\ \equiv$

$merge[all](some\{e = v_1|v_1 \leftarrow \mathcal{M}_1(\alpha)\}, all\{e \neq v_{1i}|v_{1i} \leftarrow \mathcal{M}_2(\alpha)\})$

C.7.2.2 with $\cap$

$e\ \in\ (\mathcal{M}_1(\alpha)\ \cap \mathcal{M}_2(\alpha))\ \equiv\ (e\ \in \mathcal{M}_1(\alpha))\ \wedge\ (e\ \in \mathcal{M}_2(\alpha))$

Equivalent Monoid comprehension term:

Ax 20 $some\{\ e = v_1 \mid v_1 \leftarrow set\{\ v_{1i} \mid v_{1i} \leftarrow \mathcal{M}_1(\alpha),\ some\{\ v_2 = v_{1i} \mid v_2 \leftarrow \mathcal{M}_2(\alpha)\ \}\ \}\ \}\ \equiv$

$merge[all](some\{\ v_1 = e \mid v_1 \leftarrow \mathcal{M}_1(\alpha)\}, some\{\ v_2 = e \mid v_2 \leftarrow \mathcal{M}_2(\alpha)\})$

C.7.2.3 with $\cup$

$e\ \in\ (\mathcal{M}_1(\alpha)\ \cup\ \mathcal{M}_2(\alpha))\ \equiv\ (e\ \in \mathcal{M}_1(\alpha))\ \vee\ (e\ \in\ \mathcal{M}_2(\alpha))$
Monoid comprehension equivalent:

Ax 21 $some\{\ e = v_1 \mid v_1 \leftarrow (\mathcal{M}_1(\alpha) \cup \mathcal{M}_2(\alpha))\ \}\ \equiv$

$merge[some](some\{\ e = v_1 \mid v_1 \leftarrow \mathcal{M}_1(\alpha)\},\ \ some\{\ e = v_2 \mid v_2 \leftarrow \mathcal{M}_2(\alpha)\ \}$

C.8 Miscellaneous Identities

The identities listed in this section do not within themselves directly lead to more efficient implementations and are hence not applied. The fact that these rewrites rules do not directly lead to performance improvements is largely due to the pipelined evaluation model which is utilised in order to evaluate skeleton programs. In the following axioms, $\mathcal{M}_1$ is limited to one of list, bag, or set depending on the category concerned.

C.8.1 Cascading Map Operations

Ax 22 $\mathcal{M}_1\{\ r_1 \mid v_1 \leftarrow \mathcal{M}_1\{\ \ r_2 \mid v_2 \leftarrow \mathcal{M}_1(\alpha)\}\ \}\ \equiv\ \ \mathcal{M}_1\{\ r_1^{r_2/v1} \mid v_2 \leftarrow \mathcal{M}_1(\alpha)\}$

---

where r_2 is either attribute selection.

C.8.2 Cascading Filters

Ax 23 $\mathcal{M}_1\{v_1 \mid v_1 \leftarrow \mathcal{M}_1\{v_2 \mid v_2 \leftarrow \mathcal{M}_1(\alpha),\ P_2^+(v_2)\ \}, P_1^+(v_1)\ \}\ \equiv$
$\mathcal{M}_1\{\ v_1 \mid v_1 \leftarrow \mathcal{M}_1(\alpha),\ P_2^{v_1/v_2}(v_2)^+,\ P_1^+(v_1)\}$

C.8.3 Cascading Map and Filter

Ax 24 $\mathcal{M}_1\{\ r_1 \mid v_1 \leftarrow \mathcal{M}_1\{\ \ r_2 \mid v_2 \leftarrow \mathcal{M}_1(\alpha),\ P_2^+(v_2)\},\ P_1^+(v_1)\}\ \equiv$

$$\mathcal{M}_2\{\ r_1^{r_2/v_1} \mid v_2 \leftarrow \mathcal{M}_1(\alpha),\ P_2^+(v_2),\ P_1{}^{r_2/v_1}(v_2)^+\}$$

C.8.3.1 With Additional Terms

Ax 25 $\mathcal{M}_i\{\ r_1(v_1) \mid v_1 \leftarrow \mathcal{M}_j\{\ \ r_2(v_3) \mid v_2 \leftarrow \mathcal{M}_k(\alpha),\ P^+(v_2),\ v_3 \leftarrow r_3(v_2)\ \}\ \} \equiv$

$$\mathcal{M}_i\{\ r_1(v_1)^{r_2(v_3)/v_1} \mid v_2 \leftarrow \mathcal{M}_k(\alpha),\ P^+(v_2),\ v_3 \leftarrow r_3(v_2)\ \}$$

C.9 Implementation Based Improvements

C.9.1 Use of Indexes

The axioms below transforms the combined generation from an instance of a collection and an attribute selection into an *indexed* selection, should an index exist on the attribute, and indexed generations be supported at the implementation level.

Ax 26 $\mathcal{M}_1\{\ r \mid v \leftarrow \mathcal{M}_i(\alpha),\ CT_1,\ att\_sel\ v = e\ , CT_2\} \equiv$

$$\mathcal{M}_1\{\ r \mid v \leftarrow\ inv\_att\_sel(\mathcal{M}_i(\alpha), att\_sel_i)\ e,\ CT_1\ , CT_2\}$$

$v_2 \notin FV(CT_1)$.

C.10 Discussion

Wong presents a number of rewrite rules on queries over nested collections expressed in the comprehension–based language, CPL [Won94], which is based on the *Nested Relation Calculus* ($\mathcal{NRC}$). Particular attention is paid to the optimisation of queries involving loops for which two well–known optimising techniques involving the combination of two loops into a single loop [Fre86] are presented. The fusion of two loops where the first loop produces the data consumed by the second loop is termed *vertical loop fusion.* The fusion of two independent loops iterating over the same collection termed *horizontal loop fusion.* In this section, the relevance of these optimisations is considered in the context of the skeleton evaluation model presented in this thesis.

C.10.1 Vertical Loop Fusion

Vertical loop fusion in CPL can be specified as the transformation/optimisation [Won94]:

$$\bigcup\{e_1|x \in \bigcup\{e_2|y \in e_3\}\} \rightsquigarrow \bigcup\{\bigcup\{e_1|x \in e_2\}|y \in e_3\}$$

where $\bigcup\{e_1|x \in e_2\}$ is defined to be $sru(\cup, \lambda x.e_1, \{\})(e_2)$ [TBW92, TS91].

The motivation for this optimisation is to avoid calculating a partial result from which elements are subsequently generated. The general form for this optimisation in MQL is:

Def C.1 $set\{e_1'|x \leftarrow set\{e_2'|y \leftarrow e_3\}\} \equiv unnest(\{set\{e_1'|x \leftarrow e_2\}|y \leftarrow e_3\})$

where *unnest* is a function that unnests a collection, and $unit[set](e_i') = e_i$. This is necessary since the expression e_1 in SRU is itself a singleton set. In the MQL to MHL translation scheme, the result of applying the head of the comprehension to each element within the collection is $unit[\mathcal{M}](||a||)$. However, this is not the case in $\mathcal{NRC}$, in which expressions are of the form $\bigcup\{e_1|x \in e_2\}$. In $\mathcal{NRC}$ this denotes the set obtained by first applying the function $\lambda x.e_1$ to each object within the set e_2 and then taking the union (i.e the restricted form of structural recursion, $SRU(\cup, \lambda x.e_1, \{\})(e_2)$). Hence, in an expression such as $\bigcup\{\bigcup\{e_1|x \in e_2\}|y \in e_3\}$, the overall result is an atomic collection, whereas the result of expression $set\{set\{e_1|x \leftarrow e_2\}|y \leftarrow e_3\}$ in MQL is a nested collection $set(set(\alpha))$. The repercussion of this is that in MQL it is necessary to unnest the results of the outer comprehension

Consider the definition of a Cartesian product in $\mathcal{NRC}$:

$$\bigcup\{\bigcup\{\{(x,y)\}|x \in xs\}|y \in ys\}$$

In this case, the union of results for each value of y will yield an atomic set, since the function f which is applied to each element is $\bigcup\{\{(x,y)\}|x \in xs\}$. The set union of these results is then taken. The definition of flatten in $\mathcal{NRC}$ is given by $\bigcup\{x|x \in xs\}$. In the evaluation mechanism used for MQL, the result of the inner comprehension is *not* produced as a partial result.

A consequence of this is that in Def C.1, the evaluation of the inner comprehension on the left–hand side of the rule does not actually produce the complete partial result set. Instead, due to the pipelined evaluation model, the elements which are produced in by inner comprehensions are piped directly through to the outer comprehension.

Since a partial result is not produced in the MHL evaluation model, the application of *vertical loop fusion* does not facilitate a overall performance improvement. However, in cases where loops can be fused, the synchronisation overheads between the two pipe–lined loops may be reduced through the application of this rewrite rule.

C.10.2 Horizontal Loop Fusion

The other form of loop optimisation for $\mathcal{NRC}$ queries presented by Wong (also in [BK90]) is *horizontal loop fusion* which is applicable to queries in which there are two independent loops over the same set. *Horizontal loop fusion* in the $\mathcal{NRC}$ is specified by the transformation:

$$\bigcup\{e_1|x \in e\} \cup \bigcup\{e_2|x \in e\} \leadsto \bigcup\{e_1 \cup e_2|x \in e\}$$

The equivalent formulation for this rule in MQL is given below (also Ax 18):

$$merge[\mathcal{M}_i](\{e_1'|x \leftarrow e\}, \{e_2'|x \leftarrow e\}) \equiv \mathcal{M}_i\{\ merge[\mathcal{M}_i](e_1, e_2) \mid x \leftarrow e\ \}$$

where $unit[\mathcal{M}_i](e_1)$ and $unit[\mathcal{M}_i](e_2)$ represent e_1' and e_2' in $\mathcal{NRC}$.

Appendix D

Compilation Process

D.1 Symbol Table Construction Algorithm

$$\mathcal{T} :: (MHL\_term, (int, int), ST) \rightarrow (ST, VSL\_exp)$$

MHL Form	Preconditions	Action
$\mathcal{T}(c, scope, ST)$	c int,bool,string	(ST, c)
$\mathcal{T}(v_i s, scope, ST)$	$v_i s$ is an instance of a collection	$(ST, v_i s)$
$\mathcal{T}(v_i, current\_scope, ST)$	v_i is a generated variable	**if** $\neg$ $first\_unit$ **then** **if** $not\_null(get\_field(TAIL, st\_record(ST, v_i, current\_scope)))$ **then** $(ST, get\_field(TAIL, st\_record(ST, v_i, current\_scope)))$ **else let** $data\_source = get\_field(COLLECTION, st\_record(ST, v_i, current\_scope))$ **in if** $base\_collection(data\_source)$ **then** **let** $(record, table) = remove\_st\_record(v_i, current\_scope, new\_ST)$; **in** $(add\_record(set\_field(TAIL, TL(\langle v_i, data\_source, get\_field(SCOPE, st\_record(ST, v_i,$ $current\_scope))\rangle_r), record), table), TL(\langle v_i, data\_source, get\_field(SCOPE, st\_record(ST, v_i, current\_scope))\rangle))$ **end** **else let** $(new\_ST, head_{new}) = \mathcal{T}(\langle data\_source, get\_field(SCOPE, st\_record(ST, v_i, current\_scope)), 2\rangle,$ $ST)$; **in let** $(record, table) = remove\_st\_record(v_i, current\_scope, new\_ST)$ **in if** var_gens(data_source) **then** $(add\_record(set\_field(TAIL, \Leftarrow TL((head_{new})), record), table), \Leftarrow TL(head_{new}));$ **else** $(add\_record(set\_field(TAIL, TL(head_{new}), record), table), TL(head_{new}));$ **end end end** **else** (ST, v_i)
$\mathcal{T}(v_i.j, scope, ST)$	-	**let** $(new\_ST, head) = \mathcal{T}(v_i, scope, ST)$ **in** $(new\_ST, head.1)$ **end**
$\mathcal{T}(\langle e_1, \ldots, e_n\rangle, scope)$	-	**let** $(new\_ST_1, head_1) = \mathcal{T}(e_1, scope, ST)$ **in** ... **let** $(new\_ST_n, head_n) = \mathcal{T}(e_n, scope, new\_ST_{n-1})$ **in** $(new\_ST_n, \langle head_1, \ldots, head_n\rangle)$**end** ... **end**
$\mathcal{T}(att\_sel(v_i), scope, ST)$	-	**let** $(new\_ST, head) = \mathcal{T}(v_i, scope, ST)$ **in** $(new\_ST, att\_sel(new\_head))$**end**

$\mathcal{T} :: (MHL\_term, (int, int), ST) \rightarrow (ST, CSL\_exp)$

MHL Form	Preconditions	Action
$\mathcal{T}((e_1)\ relop\ (e_2), scope, ST)$	$relop \in \{=, !=, <, \leq, >, \geq\}$	**let** $(new\_ST_1, head_1) = \mathcal{T}(e_1, scope, ST)$ **in let**$(new\_ST_2, head_2) = \mathcal{T}(e_2, scope, new\_ST_2)$ **in** **if** $base\_filter(e_1\ relop\ e_2)$ **then** $\forall var \in vars(e_1\ relop\ e_2)$. **let** $(new\_table, record) = remove\_st\_record(var, scope, ST)$ **in** $(add\_record(add\_pred(get\_field(pred, st\_record(ST, var, scope)))), NULL\_TAIL)$ **end** **else** $(new\_ST_2, NULL\_TAIL)$ **end**
$\mathcal{T}(e_1/e_2, scope, ST)$	-	**let** $(new\_ST_1, head\_1) = \mathcal{T}(e_1, scope, ST)$ **in let** $(new\_ST_2, head_2) = \mathcal{T}(e_2, scope, new\_ST_1)$ **in** $(new\_ST_2, NULL\_TAIL)$ **end end**
$\mathcal{T}$ (**if** $pred$ **then** e **else** $zero[\mathcal{M}_i], scope, ST)$	-	**let** $(new\_ST_1, head_1) = \mathcal{T}(e, \langle scope, 3\rangle)$ **in let** $(new\_ST_2, head_2) = \mathcal{T}(pred, scope)$ **in** $(new\_ST_2, head_1)$ **end end**
$\mathcal{T}(zero[\mathcal{M}].scope, ST)$	-	$(ST, NULL\_TAIL)$
$\mathcal{T}(unit[\mathcal{M}](e), scope, ST)$	-	**if** $first\_unit$ **then** $first\_unit := false$; **let** $(new\_ST_1, head_1) = \mathcal{T}(e, scope, ST)$ **in** $(new\_ST_1, HD(head_1))$ **end** **else** $(new\_ST_1, \Rightarrow \mathcal{M}(head_1))$
$\mathcal{T}(merge[\mathcal{M}](e_1, e_2), scope, ST)$	-	**let** $(new\_ST_1, head_1) = \mathcal{T}(e_1, scope, ST)$ **in let** $(new\_ST_2, head_2) = \mathcal{T}(e_2, scope, new\_ST_1)$ **in** $(new\_ST_2, merge[\mathcal{M}](head_1, head_2))$ **end end**
$\mathcal{T}(hom[\mathcal{M}_i, \mathcal{M}_j]\lambda\ v_i\ (e_k)\ e_l, scope, ST)$	-	**let** $new\_ST = add\_record(set\_field(COLLECTION, e_l, set\_field(SCOPE, \langle scope, 1\rangle, set\_field(set\_field(VAR, v_i, new\_ST\_record)))))$; $first\_unit = TRUE$ **in let** $(new\_ST_1, HD(head'_1)) = \mathcal{T}(e_k, \langle scope, 1\rangle, new\_ST)$ **in let** $(new\_table, record) = remove\_st\_record(v_i, scope, new\_ST_1)$ **in** **if** $SM(\mathcal{M}_j)$ **then** $(new\_table, S\mathcal{M}_j\{HD(head'_1)\})$ **else** $(new\_table, \mathcal{M}_j\{HD(head'_1)\})$ **end end end**

Table D.1: Algorithm for the Construction of the Symbol Table

Boolean Flag	Significance
$new\_ST\_record()$	= new symbol table record
$add\_record(r, ST)$	$= r :: ST$
$remove\_st\_record(ST, var, scope)$	remove and return the symbol table record applicable to the innermost instance of the variable var, as determined by the $scope$
$st\_record(ST, v_j, scope)$	= returns the symbol table record belonging to the innermost record, by scope, in the ST for which $var(r) = v_i$
$get\_field(f, r)$	=returns the value for field f in the symbol table record r
$set\_field(f, val, r)$	= set field f of symbol table record r to value val return the amended symbol table record
$add\_pred(pred, r)$	= adds the predicate $pred$ to the symbol table record r
$base\_filter(filt)$	= true if $filt$ does not contain any homomorphisms with the range of a Monoid defined over $\mathbb{B}$
$base\_collection(ds)$	= true if ds is a *resident collection*
$from\_var(ds)$	=true if ds is a variable or an attribute selection on a variable
$var\_gens(e)$	=true if e is a previously generated variable or an attribute, att_sel_i.

Table D.2: Symbol Table Key

D.2 VSL Normalisation Algorithm

Form	*Description*
$\mathcal{N}(NULL\_TAIL)$	$NULL\_TAIL$
$\mathcal{N}(zero[\mathcal{M}])$	$NULL\_TAIL$
$\mathcal{N}(c)$	c
$\mathcal{N}(xs)$	xs
$\mathcal{N}(\langle v_i, base\_coll, scope\rangle_r)$	$base\_coll$
$\mathcal{N}(\langle VSL_{exp}, \ldots, VSL_{exp}\rangle)$	$\langle \mathcal{N}(VSL_{exp}), \ldots, \mathcal{N}(VSL_{exp})\rangle$
$\mathcal{N}(att\_sel(VSL_{exp}))$	$att\_sel(\mathcal{N}(VSL_{exp}))$
$\mathcal{N}(VSL_{exp}.i)$	$\mathcal{N}(VSL_{exp}).i$
$\mathcal{N}(HD(VSL_{exp}))$	$\mathcal{N}(VSL_{exp})$
$\mathcal{N}(TL(\mathcal{M}_i(HD(VSL_{exp}))))$	***if*** *collection*$(\mathcal{M}_i)$ ***then*** $\mathcal{N}(VSL_{exp})$ ***else*** $\leftarrow (\mathcal{M}_i(\mathcal{N}(VSL_{exp}))$
$\mathcal{N}(TL(VSL_{exp}))$	$\leftarrow \mathcal{N}(VSL_{exp})$
$\mathcal{N}(\Rightarrow \mathcal{M}(VSL_{exp}))$	$\Rightarrow \mathcal{M}(\mathcal{N}(VSL_{exp}))$
$\mathcal{N}(\Leftarrow (VSL_{exp}))$	$\Leftarrow (\mathcal{N}(VSL_{exp}))$
$\mathcal{N}(merge[\mathcal{M}](VSL_{exp1}, VSL_{exp2}))$	***if*** *collection*$(\mathcal{M})$ ***then*** *split*$(\mathcal{N}(VSL_{exp1}), \mathcal{N}(VSL_{exps2}))$ ***else*** $merge[\mathcal{M}](\mathcal{N}(VSL_{exp1}), \mathcal{N}(VSL_{exp2}))$
$\mathcal{N}(\mathcal{M}_i\{VSL_{exp}\})$ $\mathcal{N}(S\mathcal{M}_i\{VSL_{exp}\})$	$\mathcal{M}_i\{\mathcal{N}(VSL_{exp})\}$ $S\mathcal{M}_i\{\mathcal{N}(VSL_{exp})\}$

Table D.3: VSL Normalisation Algorithm

[0]Note that *collection* in this case refers to the three collections types *list, bag* and *set*. The *grouped*$(\mathcal{M} \times \mathcal{N})$ does not count as a collection for *this* purpose.

D.3 Treatment of Idempotent Predicate Terms

Consider the VSL substituted predicate given below:

$$P_j(TL(\mathcal{M}_i(HD(TL(\langle a, as, scope_a\rangle)))), TL(\langle b, bs, scope_b\rangle))$$

The leftmost VSL term in the filter matches against the rule $\mathcal{N}(TL(VSL\_exp))$. The applicability of the reduction rule depends on the category of the monoids $\mathcal{M}_i$ and as. Suppose that as is a free Monoid in the *CIM* category and that $\mathcal{M}_i$ is a free Monoid in the *CIM* category. In this case, if the filter P_j is a non–idempotent filter, then the reduction rule may *not* be applied since the information regarding the non–idempotent filter is lost in the normalisation process. The *incorrect* normalised form of the filter is given below:

$$P_j(\langle a, as, scope_a\rangle, \langle b, bs, scope_b\rangle)$$

Hence, the application of the normalisation rule that deals with $\mathcal{N}(TL(VSL\_exp))$ has to take into account the properties of the base collections and interim result Monoids. The cases in which the reduction rule may be applied are identified using the following MQL query:

$$\mathcal{M}\{a|a \leftarrow \mathcal{M}_i\{b|b \leftarrow bs, att\_sel_i(b) = constant\}, exps\}$$

where *exps* denotes a sequence of MQL terms.

The forms that *exps* may take, and the treatment with respect to the application of the rule reducing $\mathcal{N}(TL(VSL\_exp))$ in each case, are given below:

- $c \leftarrow cs$, $att\_sel_a(a)$ $relop$ $att\_sel_c(c)$. The reduction may be applied to the VSL expressions for the variables a and c.

- $\mathcal{M}_j\{head_i|v_1 \leftarrow \mathcal{M}_k(\alpha), \dots, P_k(v_1, c)\}$ where $\mathcal{M}_j$ is a Monoid defined over $\mathbb{B}$.

  The application of the normalisation rule is guarded by the following algorithm which determines the cases in which the normalisation may not be applied.

```
If Idempotent_Monoid(M_j)
  then if Idempotent_Monoid(M_i)
        then if Idempotent_Monoid(base_collection(bs))
              then apply_reduction
              else ¬ apply_reduction
        else apply_reduction
  else if Idempotent_Monoid(M_i)
        then ¬ apply_reduction
        else apply_reduction
```

- $\mathcal{M}_j\{\dots|c \leftarrow cs, \dots, P(c, a)\}$ $relop$ $att\_sel(a)$.

```
If Idempotent_Monoid(M_j)
  then if Idempotent_Monoid(M_i)
        then if Idempotent_Monoid(base_collection(bs))
              then apply_reduction
              else ¬ apply_reduction
```

```
else apply_reduction
else if Idempotent_Monoid(M_i)
    then ¬ apply_reduction
    else apply_reduction
```

where the function *base_collection*(MHL_exp) returns the collection associated with the MHL_exp expression given.

D.4 Data Requirements Generation Algorithm

if $(inner\_head(get\_field(TL, str_i))) = NULL\_TAIL) \wedge constant\_head(inner\_head(get\_field(TL, str_i))))$

then $NULL$

else

if $(get\_field(pred\_list, str_i) = [\,]) \vee (\forall P_j \in get\_field(pred\_list, str_i).svp(v_i, P_j))$

then if $appears\_in(var_i, inner\_head(get\_field(TL, str_i))))$

then if $num\_records(ST) = 1$

then if $variables\_in\_head(inner\_head(get\_field(TL, str_i))) = \{var_i\}$

then /*just a single variable in the head*/

$NULL$

else

if $\exists str_j \ st \ str_j \in ST \wedge (str_i \neq str_j) \ st. \ appears\_in(var_i, head(\ get\_field(collection, str_i))))$

then /*generated variable */

$NULL$

else /*subset of Cartesian Product*/

$DUPLICATE(get\_field(collection, str_i))$

else $NULL$

else $NULL$

else /*does not appear in head. Redundant generator or feeder generator*/

$NULL$

else /*the predicate list has some common predicates*/

$(\forall \ P_k \in get\_field(pred\_list, str_i) \wedge multi\_vars(P_k) \wedge \neg marked(P_k).$

$(\forall \ common\_var_l \ st \ \ common\_var_l \in vars(P_k) \wedge common\_var_l \neq var_i.$

$str_{common\_var_l} = st\_record(ST, common\_var_l, same\_scope\_body(ST, get\_field(scope, str_i))).$

$(\forall \ pred_l \in get\_field(pred\_list, str_{common\_var_l}).mark\_predicate(pred_l, str_i) \ \ )))$

where $var_i = get\_field(variable, ST_i)$

Form	Description
$inner\_head(e : VSL\_exp)$	returns the *innermost* head of the VSL expression VSL_exp.
$constant\_head(head\_val : VSL\_exp)$	=true if *head_val* is of the form of a constant
$vars(P_l : VSL\_substituted\_MHL)$	returns a set of variables that are used in the prediate P_l
$num\_records$(symbol_table: ST)	returns the number of records in the symbol table *sym_tab*
$appears\_in(var_i : variable, head_{final} : VSL)$	=true if the variable var_i appears in the VSL expression, $head_{final}$
$multi\_vars(P_l : VSL\_substituted\_MHL)$	returns the set of variables in P_l
$mark\_predicate(P_l, str_i)$	marks the predicate in the symbol table record as having been processed already.
$marked(P_l)$	=true if the predicate has already been processed
$svp(var_i : variable, P : VSL\_substituted\_MHL)$	=true if the predicate only on variable var_i
$same\_scope\_body(s : SCOPE, ST : symbol\_table)$	= returns the scope field in the symbol table for the instance of $common\_var_l$ which *shares* a scope with s. A scope $\langle s_{11}, s_{12}\rangle$ and $\langle s_{21}, s_{22}\rangle$ are determined to *share* when either $s_{11} = \langle\langle \ldots, \langle\langle s_{21}, s_{22}\rangle, 1\rangle, \ldots, 1\rangle$, or $s_{21} = \langle\langle, \ldots_{1s}, \langle\langle s_{11}, s_{12}\rangle, 1\rangle, \ldots_{1s}, 1\rangle$, where $\ldots_{1s}$ means that the numerical scopes in the *dotted* are all 1.

Table D.4: Data Requirements Generation Algorithm

Appendix E

Sample Database — SR–CD

The structure of the individual components in the underlying sets *cds* and *tracks* is outlined in this appendix.

Size of a single element (tuple) for each collection is given below:

- cds = 1060 bytes
- tracks = 1748 bytes

E.1 cd Element

> (Record_type CHAR(4), Date_entered INTEGER, Archive_number_BE CHAR(64), Archive_number_LA CHAR(64), Archive_number_RI CHAR(64), Number_BE INTEGER, Number_LA INTEGER, Number_RI INTEGER, Media_type CHAR(5), Number_of_disks INTEGER, Total_duration CHAR(6), cd_id INTEGER, Distributor CHAR(50), Dist_order_num CHAR(20), Year_of_production INTEGER, Year_circa CHAR(1), Production_type CHAR(1), Entering_comment_D CHAR(150), Entering_comment_F CHAR(150), Entering_comment_I CHAR(150), Title CHAR(200), Person_entered_BE CHAR(20), Person_entered_LA CHAR(20), Person_entered_RI CHAR(20), Date_received_BE INTEGER, Date_received_LA INTEGER, Date_received_RI INTEGER, Date_archived_BE INTEGER, Date_archived_LA INTEGER, Date_archived_RI INTEGER, Disk_status CHAR(2), Sales INTEGER)

Note that the `sales` value has been added from separate sales information and does not form the cds table of the original database.

E.2 distributor element

The *distributors* collection is the

> (Code CHAR(20), Full_name Char(100))

E.3 track Element

Record_type CHAR(4), Date_entered INTEGER, cd_id CHAR(13), Track_id CHAR(17), Track_no INTEGER, Featuring_artist CHAR(128), Job_title CHAR(64), Track_title_D CHAR(200), Track_title_F CHAR(200), Track_title_I CHAR(200), Disk_no INTEGER, Track_duration CHAR(6), Production_people CHAR(128), Prod_peoples_role CHAR(32), Music_type_D CHAR(50), Music_type_F CHAR(50), Music_type_I CHAR(50), More_prod_people CHAR(160), More_prod_peoples_role CHAR(64), Date_recorded CHAR(10), Technical_production CHAR(64), Tech_prod_role CHAR(32), Live_recording CHAR(1));

Appendix F

Evaluation Queries

The *MHL* formulations of the queries used in chapter 9 are presented. The syntax is concrete MHL and is the input to code generator.

F.1 Main Queries

F.1.1 1.112bag*all+atts

```
hom h1 in(bag resident(CDS)) out(bag res)
   hom_lambda hom_evar 1
         hom_condition
            (hom_comp
               (eq,hom_tup_proj (hom_evar 1,hom_att "Dist_order_num"),
                           hom_str "NULL"),
             hom_unit
                (bag,hom_evar 1),
             hom_zero bag)
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.1.2 2.98bag*all+atts

```
hom h1 in(bag resident(CDS)) out(bag res)
      hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (eq,hom_tup_proj (hom_evar 1,hom_att "Number_of_disks"),
                           hom_const 2),
             hom_unit
                (bag,hom_evar 1),
             hom_zero bag)
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.1.3 2.98bag*1+atts

```
hom h1 in(bag resident(CDS)) out(bag res)
      hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (eq,hom_tup_proj (hom_evar 1,hom_att "Number_of_disks"),
                           hom_const 2),
             hom_unit
                (bag,hom_tup_proj (hom_evar 1, hom_att "Person_entered_RI")),
             hom_zero bag)
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.1.4 15.3bag*1+atts

```
hom h1 in(bag resident(CDS)) out(bag res)
    hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (gt,hom_tup_proj (hom_evar 1,hom_att "Date_received_RI"),
                           hom_const 19910300),
             hom_unit
                (bag,hom_tup_proj (hom_evar 1,hom_att "Cd_id")),
             hom_zero bag)
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.1.5 17.3bag*1+atts

```
hom h1 in(bag resident(CDS)) out(bag res)
      hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (neq,hom_tup_proj (hom_evar 1,hom_att "Disk_status"),
                           hom_str "T"),
             hom_unit
                (bag,hom_tup_proj (hom_evar 1,hom_att "Cd_id")),
             hom_zero bag)
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.1.6 100bag*1+atts

```
hom h1 in(bag resident(CDS)) out(bag res)
      hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (gt,hom_tup_proj (hom_evar 1,hom_att "Date_entered"),
                 hom_const 19920000),
             hom_unit
                (bag,hom_tup_proj (hom_evar 1,hom_att "Cd_id")),
             hom_zero bag)
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.1.7 100bag*3+atts

```
hom h1 in(bag resident(cds)) out(bag res)
      hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (gt,hom_tup_proj (hom_evar 1,hom_att "Date_entered"),
                 hom_const 19920000),
             hom_unit
                (bag,<hom_tup_proj (hom_evar 1,hom_att "Cd_id"),
                      hom_tup_proj (hom_evar 1,hom_att "Total_duration"),
                      hom_tup_proj (hom_evar 1,hom_att "Date_entered")>),
             hom_zero bag)
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.1.8 100bag-and-max100*1+atts

```
hom h1 in(bag resident(CDS)) out(<max,bag> res)
       hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (gt,hom_tup_proj (hom_evar 1,hom_att "Date_entered"),
                 hom_const 0),
             <hom_unit(bag,hom_tup_proj (hom_evar 1,hom_att "Cd_id")),
               hom_tup_proj (hom_evar 1, hom_att "Date_entered")>, <0,hom_zero bag>)
end

main in() out(<max,bag> res)
 h1 in() out(<max,bag> res)
end
```

F.1.9 min100*1+atts

```
hom h1 in(bag resident(CDS)) out(min res)
      hom_lambda hom_evar 1
         hom_condition
            (hom_comp
                (gt,<hom_tup_proj (hom_evar 1,hom_att "Date_entered"),
                     hom_const 19920000>),
                 hom_tup_proj (hom_evar 1, hom_att "Date_entered"), 0)
end

main in() out(min res)
 h1 in() out(min res)
end
```

F.2 Join Queries

F.2.1 CDS100*TRACKS100bag-1+att

```
hom h1 in(bag resident(CDS)) out(bag res)
  hom_lambda hom_evar 1
     hom h2 in(cd hom_evar 1, bag resident(TRACKS)) out(bag res)
       hom_lambda hom_evar 2
         hom_condition
                   (hom_comp
                       (eq,hom_tup_proj (hom_evar 1,hom_att "Cd_id"),
                           hom_tup_proj (hom_evar 2,hom_att "Cd_id")),
         hom_unit(bag,hom_tup_proj (hom_evar 1,hom_att "Cd_id")),
         hom_zero bag)
     end
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.2.2 CDS100*TRACKS100bag-1+att

```
hom h1 in(bag resident(CDS)) out(bag res)
  hom_lambda hom_evar 1
     hom h2 in(cd hom_evar 1, bag resident(TRACKS)) out(bag res)
       hom_lambda hom_evar 2
          hom_condition
           (hom_comp
            (eq,hom_tup_proj (hom_evar 1,hom_att "Date_entered"),
                hom_tup_proj (hom_evar 2,hom_att "Date_entered")),
            hom_unit (bag,hom_tup_proj (hom_evar 1,hom_att "Cd_id")),
            hom_zero bag)
     end
end

main in() out(bag res)
 h1 in() out(bag res)
end
```

F.2.3 CDS100*TRACKS100sum

```
hom h1 in(bag resident(CDS)) out(sum res)
   hom_lambda hom_evar 1
      hom h2 in(cd hom_evar 1, bag resident(TRACKS)) out(sum res)
         hom_lambda hom_evar 2
             hom_condition
                (hom_comp
                    (eq,hom_tup_proj (hom_evar 1,hom_att "Cd_id"),
                        hom_tup_proj (hom_evar 2,hom_att "Cd_id")),
             hom_const 1,0)
      end
end

main in() out(sum res)
h1 in() out(sum res)
end
```

F.3 Optimised Queries

This section presents the source optimised and unoptimised skeleton programs that are used to investigate the performance of Ax 9 which is investigated in § 9.6, chapter 9. The optimised and unoptimised query pair for the application of Ax 7 are presented in chapter 9.

F.3.1 Qualification, Ax 9

F.3.1.1 Unoptimised Query

```
hom h1 in(bag resident(CDS)) out(bag res)
 hom_lambda hom_evar 1
   hom h2 in(alpha hom_evar 1, bag resident(TRACKS)) out(bag res)
    hom_lambda hom_evar 2
      hom_condition
        (hom_myand(
           hom_comp(eq,hom_tup_proj (hom_evar 1,hom_att "Date_entered"),
                       hom_tup_proj (hom_evar 2,hom_att "Date_entered")),
           hom_comp(eq,hom_tup_proj (hom_evar 1,hom_att "Disk_status"), hom_str "T")),
      hom_unit (bag,hom_tup_proj (hom_evar 1,hom_att "Cd_id")),
      hom_zero bag)
   end
end

main in() out(min res)
 h1 in() out(min res)
end
```

F.3.1.2 Optimised Query

```
hom h1 in(bag TRACKS) out(bag res)
  hom_lambda hom_evar 1
    hom h2 in(bag processed-CDS) out(bag res)
      hom_lambda hom_evar 2,
        hom_condition
            (hom_comp(eq,hom_tup_proj (hom_evar 1,hom_att "Date_entered"),
                         hom_tup_proj (hom_evar 2,hom_att "Date_entered")),
         hom_unit(bag,hom_tup_proj (hom_evar 1,hom_att  "Cd_id")), hom_zero bag)
    end
end

hom h3 in(in resident(CDS)) out(bag processed-CDS)
  hom_lambda hom_evar 1
   hom_condition
    (hom_comp (eq,hom_tup_proj (hom_evar 1,hom_att "Disk_status"), hom_str "T"),
    hom_unit (bag,hom_evar 1),
    hom_zero bag)
end

pipe p1 in() out(bag res)
 h3 in(bag resident(CDS)) out(bag processed-cds)
 h1 in(bag processed-cds) out(bag res)
end

main in() out(bag res)
 p1 in() out(bag res)
end
```

Bibliography

[ABBM97a] M Afshar, J Bates, G Bierman, and K Moody. A new general purpose parallel database system. In *Proceedings 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)*, pages 2–8. IEEE Press, December 1997. Taipei.

[ABBM97b] M Afshar, J Bates, G Bierman, and K Moody. The transparent parallelisation of declarative database queries: Theory and practice. In *Proceedings of the Seventh Parallel Computing Workshop*, pages P2–M1–P2–M8, 1997. Canberra, Australia, September 25–27.

[AK94] M Afshar and H Khoshnevisan. Mechanical parallelisation of database applications. In Ed Deaton, Dave Oppenheim, Joseph Urban, and Hal Berghel, editors, *Proceedings of The 1994 ACM Symposium on Applied Computing*, pages 436–441. ACM, ACM Press, March 1994.

[Bac81] J W Backus. *The algebra of functional programs: function level reasoning, linear equations and extended definitions*, volume 107, chapter 1, pages 1–43. Springer-Verlag, 1981.

[Ban92] F Bancilhon. Object oriented database systems. In *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, pages 152–162. ACM, 1992.

[BBKV87] F Bancilhon, T Briggs, S Koshafian, and P Valduriez. FAD a powerful and simple database language. In *Proceedings of the 13 VLDB Conference, Brighton*, pages 97–105, 1987.

[BC90] G E Blelloch and S Chatterjee. VCODE: A data-parallel intermediate language. In *Proceedings Frontiers of Massively Parallel Processing Symposium*, pages 471–480, October 1990.

[BCC+90] H Boral, L Clay, G Copeland, S Danforth, M Franklin, M Smith, and P Valduriez. Prototyping Bubba. a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[BCD+97] B Bacci, B Cantalupo, M Danelutto, S Orlando, D Pasetto, S Pelagatti, and M Vanneschi. An environment for structured parallel programming. In L Grandinetti, M Kowalick, and M Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwier, Dordrecht, The Netherlands, 1997.

[BDJ+91] F Baiardi, M Danelutto, M Jazayeri, S Pelagatti, and M Vanneschi. Architectural models and design methodologies for general-purpose highly-parallel computers. In *IEEE CompEuro 91 - Advanced Computer Technolody, Reliable Systems and Applications, Bologna, Italy, May 13-16*. IEEE, 1991.

[BDO+95] B Bacci, M Danelutto, S Orlando, S Pelagatti, and M Vanneschi. P3L: a structured high level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[BDP93] B Bacci, M Danelutto, and S Pelagatti. Automatic balancing of stream-parallel computations in P3L. Technical Report TR_35/93, Department of Computer Science, University of Pisa, Corso Italia, 40 - Pisa, Italy, December 1993.

[BDV95] L Bougamin, B Dageville, and P Valduriez. Adaptive parallel query execution in DBS3. Technical Report 2749, INRIA, December 1995.

[BF79] P Buneman and R Frankel. FQL - a functional query language. In *Proceedings ACM Sigmod, Boston, MA*, pages 52–59, 1979.

[BF97] C Ballinger and R Fryer. Born to be parallel: Why parallel origins give Teradata an enduring performance edge. *Data Engineering Bulletin 20(2): 3-12 (1997)*, 20(2):3–12, 1997.

[BFG+95] C K Baru, G Fecteau, A Goyal, H Hsiao, A Jhingran, S Padmanabhan, G P Copeland, and W G Wilson. Db2 Parallel Edition. *IBM Systems Journal*, 34(2):292–322, 1995.

[BG96] G E Blelloch and J Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96), Philadelphia, Pennsylvania (also SIGPLAN Notices 31(6))*. ACM, ACM Press, May 1996.

[Bir86] R S Bird. An introduction to the theory of lists. Working Material for lectures, 1986. International Summer School on Logic Programming and Calculi of Discrete Design, Marktoberdorf, Germany, July 29 - August 10.

[BK90] C Beeri and Y Korantzky. Algebraic optimisation of object-oriented query languages. In S Abiteboul and P C Kanellakis, editors, *ICDT '90, Proceedings: Third International Conference on Database Theory*, volume 470 of *Lecture Notes in Computer Science*, pages 72–88. Springer-Verlag, December 1990.

[BK96] G H Botorog and H Kuchen. Efficient parallel programming with algorithmic skeletons. In *Proceedings of EuroPar '96*, number 1123 in Lecture Notes in Computer Science, pages 718–731. Springer–Verlag, 1996.

[Ble95] G Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 152213, September 1995.

[BLS+94] P Buneman, L Libikin, D Suciu, V Tannen, and L Wong. Comprehension syntax. *SIGMOD RECORD*, 23(1):87–96, March 1994.

[BNTW95] P Buneman, S Naqvi, V Tannen, and L Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149:3–48, 1995.

[Bra92] T A Bratvold. Determining Useful Parallelism in Higher Order Functions. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, pages 213–226, September 1992. Aachener Informatik-Berichte Nr 92-19.

[Bra93] T A Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–34, September 1993.

[Bra94] T A Bratvold. Parallelising a Functional Program Using a List-Homomorphism Skeleton. In Hoon Hong, editor, *Proceedings of PaSCo94, Linz, Austria*, pages 44–53. World Scientific Publishing Company, September 1994.

[Bre74] R P Brent. The parallel evaluation of atrihmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[BS90] G E Blelloch and G W Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[Bus93] D Busvine. Implementing recursive functions as processor farms. *Parallel Computing*, 19:1141–1153, 1993.

[BV96] N Bassiliades and I Vlahavas. Hierarchical query execution in a parallel object-oriented database system. *Parallel Computing*, 22(7):1017–1048, 1996. Elsevier Science.

[Cam96] D K G Campbell. Towards the classification of algorithmic skeletons. Technical report YCS 276, Department of Computer Science, University of York, 1996.

[Cat96] R Cattell. The Object Database Standard: *ODMG-93*, rel 1.2, 1996. Morgan Kaufmann Inc.

[CC93] L Chambers and D Cracknell. Parallel features of NonStop SQL. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS 1993), Issues, Architectures, and Algorithms. San Diego, CA*, January 1993.

[CDG+93] D Culler, A Dusseau, S C Goldstein, A Krishnamurthy, S Lumetta, T von Eicken, and K Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[CDKK85] H-T Chou, D J DeWitt, R H Katz, and A C Klug. Design and implementation of the Wisconsin Storage System. *Software – Practice and Experience*, 15(10):943–962, October 1985.

[Cea94] M Carey and D DeWitt et al. Shoring up persistent applications. *SIGMOD Record, Proceedings of the 1994 ACM SIGMOD, International Conference on the Management of Data, Minneapolis, Minnesota*, 23(2):383–394, June 1994.

[CG85] S Ceri and G Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *Transactions on Software Engineering*, 11(4):324–345, 1985.

[CK93] K M Chandy and C Kesselman. CC++: A decalartive concurrent object oriented programming notation. California Institute of Technology, March 1993.

[CKP+92] D Culler, R Karp, D Patterson, A Sahay, K E Schauser, E Santos, R Subramonian, and T von Eicken. LogP: Towards a realistic model of parallel computation. Technical Report UCB/CSD 92/713, Computer Science Division, University of California, Berkley, California 94720, December 1992.

[Cod70] E F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Col89] M Cole. *Algorithmic Skeletons: Structure Management of Parallel Computation.* Pitman/MIT, 1989.

[Cor97] Sybase Corporation. Sybase MPP for the IBM RISC System/6000. http://www.sybase.com/, 1997.

[Cro96] D Crooke. *Robustness and Performance in Structured Parallelism*, volume Abstract Machine Models for Parallel and Distributed Computing, pages 135–145. IOS Press, 1996.

[CTW94] D K C Chan, P W Trinder, and R C Welland. Evaluating object-oriented query languages. *The Computer Journal*, 37(10), 1994.

[D+93] J Darlington et al. Parallel programming using skeleton functions. In *Proceedings, Parallel Architectures and Languages, Europe*, number 694 in Lecture Notes in Computer Science, pages 146–160. Springer-Verlag, 1993.

[Dav92] W Davison. Parallel index building in Informix Online 6.0. In *Proceedings of the ACM SIGMOD Conference.* ACM, ACM Press, 1992.

[DGH+88] D J DeWitt, Graefe, M Heytens, K B Kumar Gerber R, and M Muralikrishna. *GAMMA - A Higher Peformance Dataflow Database Machine.* Morgan Kaufmann, 1988.

[DGS+90] D J Dewitt, S Ghandeharizadeh, D A Schneider, A Bricker, H I Hsiao, and R Ramussen. The GAMMA database machine project. *IEEE Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[DGT93] J Darlington, M Ghanem, and H W To. Structured parallel programming. In *Programming Models for Massively Parallel Computers*, pages 160–169. IEEE Computer Society Press, 1993.

[DGTJ95] J Darlington, Y K Guo, H.W. To, and Y Jing. Functional skeletons for parallel coordination. In *Proceedings of Parallel Architectures and Languages Europe 95 (Europar 95)*, 1995.

[DHP97] S B Davidson, C Hara, and L Popa. Querying an object-oriented database using CPL*. Technical Report MS-CIS-97-07, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, May 1997.

[DMB+87] U Dayal, F Manola, A Buchmann, S Heiler, et al. Simplifying complex objects: The probe approach to modelling and querying them. In *Proceedings of Datenbank systeme in Buro, Technik und Wissenschaft*, April 1987.

[DMO+92] M Danelutto, R Di Meglio, S Orlando, S Pelagatti, and M Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8(1-3):205–220, July 1992. Elsevier Science Publisher.

[DPP97] M Danelutto, F Pasqualetti, and S Pelagatti. Skeletons for data parallelism in p3l. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *Lecture Notes in Computer Science*, pages 619–628. Springer, August 1997.

[DSB89] D J DeWitt, M Smith, and H Boral. A single-user performance evaluation of the Teradata Database Machine. In D Gawlick, M N Haynie, and A Reuter, editors, *Proceedings High Performance Transaction Systems, 2nd International Workshop, Asilomar Conference Center, Pacific Grove, California, USA, September 28-30 (HPTS 1987)*, number 359 in Lecture Notes in Computer Science, pages 244–276. Springer–Verlag, 1989.

[DV92] S Danforth and P Valduriez. A FAD for data intensive applications. *IEEE Transactions on Knowledge and Data Engineering*, 4(1):34–51, February 1992.

[EGH95] S Englert, R Glasstone, and W Hasan. A case study of NonStop SQL/MP. *SIGMOD Record*, 24(4), December 1995.

[en92] A Maaßen. Parallel programming with data structures and higher order functions. *Science of Programming*, 18:1–38, 1992.

[Feg93] L Fegaras. *A Transformational Approach to Database System Implementation.* PhD thesis, University of Massachussetts, February 1993.

[Feg94] 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, 20000 N.W. Walker Road, P.O.Box 91000, Portland, OR 97291-1000, 1994.

[FM95a] L Fegaras and D Maier. An algebraic framework for physical OODB design. In *Proceedings of 5th International Workshop on Database Programming Languages, Bubbio, Italy*, volume Proceedings of 5th International Workshop on Database Programming Languages, Bubbio, Italy, 1995.

[FM95b] L Fegaras and D Maier. Towards an effective calculus for object query languages. In *Proceedings 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California*, May 1995.

[Fok91] M M Fokkinga. An exercise in transformational programming: Backtracking and branch-and-bound. *Science of Computer Programming*, 16:19–48, 1991.

[Fre86] J C Freytag. Translating aggregate queries into iterative programs. In W W Chu, G Gardarin, S Ohsuga, and Y Kambayashi, editors, *Proceedings Twelfth International Conference on Very Large Data Bases, Kyoto, Japan*, pages 138–146. Morgan Kaufman, August 1986.

[fSIPS89] International Organisation for Standardisation. Information Processing Systems. Database language SQL with integrity enhancement. ISO/IEC 9075, 1989. (second edition).

[fSIPS92] International Organisation for Standardisation. Information Processing Systems. Information technology — database languages — SQL. ISO/IEC 9075, 1992.

[FW78] S Fortune and J Wyllie. Parallelism in random access machines. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[Gab90] E Gabber. VMMP: A practial tool for the development of portable efficient programs for multiprocessors. *IEEE Transations on Parallel and Distributed Systems*, 1(3):304–317, 1990.

[GCKL93] S Ghandeharizadeh, V Choi, C Ker, and K-M Lin. Omega: A parallel object-based system. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems, (PDIS), Issues, Architectures and Algorithms. San Diego, CA*, pages 182–186, January 1993.

[GJP97] V Gottemukkala, A Jhingran, and S Padmanabhan. Interfacing parallel applications and parallel databases. In Per-ke Larson Alex Gray, editor, *Proceedings of the Thirteenth International Conference on Data Engineering, Birmingham U.K.*, pages 355–364. IEEE Computer Society Press, April 1997.

[GKG+97] T Grust, J Krüger, D Gluche, A Heuer, and M H Scholl. Query evaluation in CROQUE - calculus and algebra coincide. In *Proceedings Fifteenth British National Conference on Databases,7th–9th July, Birkbeck College,University of London*, pages 84–100, July 1997.

[Gra94] G Graefe. Volcano-an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, February 1994.

[Gro96] The Mentat Research Group. Mentat release 3.0 notes. Technical report, Department of Computer Science, University of Virginia, Charlottesville, Virginia 22903, 1996.

[GV92] G Gardarin and P Valduriez. ESQL2, an Extended SQL2 with F-logic semantics. In *Proceedings International Conference on Data Engineering, Phoenix, Arizona*, February 1992.

[Hey92] M L Heytens. *The Design and Implementation of a Parallel Persistent Object System*. PhD thesis, MIT Laboratory of Computer Science, 1992.

[HK93] S F Hummel and R Kelly. A rationale for programming with sets. *Journal of Programming Languages*, 1:187–207, 1993.

[HK96] A Heuer and J Kröger. Query optimisation in the CROQUE project. In *Proceedings 7th International Conference and Workshop on Database and Expert Systems Applications, University of Zurich*, September 1996.

[Hoa78] C A R Hoare. Communicating sequentail processes. *Communications of the ACM*, 21(8):666–677, 1978.

[hpf93] High Performance Fortran Language Specification, version 1.0. Technical Report, May 1993. Rice University.

[hpf97] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, January 1997.

[HTP92] 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.

[IMM93] S Ichikawa, A Manabe, and T Matsuura. Connecting the AP1000 with a mainframe for computations of the experimental high energy physics. In *Fujistu Scientific & Technical Journal, Special Issue on Cellular Array Processor AP1000*, volume 29, pages 97–111. Fujitsu, Spring 1993.

[inf96] Informix Corporation. *INFORMIX—OnLine Extended Parallel Server and Hewlet-Packard Enterprise Parallel Server—Optimal Architectures for Enterprise Class Data Warehousing*, April 1996.

[JK84] M Jarke and J Koch. Query optimisation in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KGP96] C Kim, J-C Gaudiot, and W Proskurowski. Parallel computing with the Sisal applicative language: Programmability and performance issues. *Software - Practice and Experience(SPE),*, 26(9):1025–1051, 1996.

[Kil92] M F Kilian. Can object–oriented aid massively parallel programming. In D B Johnson, F Makedon, and P Metaxas, editors, *Proceedings of the Dartmouth Institute for Advanced Graduate Study in Parallel Computation Symposium*, pages 246–256, June 1992.

[Kim82] W Kim. On optimising SQL-like nested query. *ACM Transactions on Database Systems*, 7(3), September 1982.

[MBBW94] G Michaelson, T Bratvold, D Busvine, and K Waugh. Parallel implementations from functional prototype instrumentation. In Joubert, Trystram, Peters, and Evans, editors, *Parallel Computing: Trends and Applications*, pages 631–634. North-Holland Elsevier Science B.V., 1994.

[Mil80] R Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer-Verlag, 1980.

[mpi95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 1.1*, June 1995. Available from http://www.mcs.anl.gov/mpi.

[mpi97] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[MPTW94] C Mohan, H Pirahesh, W G Tang, and Y Wang. Parallelism in relational database management systems. *IBM Systems Journal*, 33(2), 1994.

[Mur92] M Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In Li-Yan Yuan, editor, *Proceedings 18th International Conference on Very Large Data Bases, August 23-27, Vancouver, Canada*, pages 91–102. Morgan Kaufman, 1992.

[Mur93] P T Murray. *Semantic Correctness in the Specification, Translation, and Parallel Implementation of SQL Queries.* PhD thesis, The University of Sheffield, Department of Computer Science, September 1993.

[NPS91] M Negri, G Pelagatti, and L Sbatella. Formal semantics of SQL queries. *ACM Transactions on Database Systems*, 17(3):513–534, september 1991.

[NS88] P A Nelson and L Snyder. *Programming paradigms for non-shared memory parallel computers*, chapter 1. MIT Press, 1988.

[OBT92] A Ohori, P Buneman, and V B Tannen. Database programming in machiavelli - a polymorphic language with static type inference. Technical report, Department of Computer and Information Science, University of Pennsylvania, 200 South 33rd Street, Philadelphia, PA 19104-6389, November 1992.

[ora86] Oracle magazine interactive. Magazine, October 8, 1986. http://www.oramag.com/press/pr410086.html.

[ora96] Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065, USA. *Oracle7 Parallel Server Concepts & Administration*, January, 8 1996. Part No. A42522-1.

[Par92] D S Parker. SVP - a model capturing sets, streams, and parallelism. In *Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada*, pages 115–126, 1992.

[Pel93] S Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, University of Pisa, Universita di Pisa-Genova-Udine, March 1993.

[PJL92] S L Peyton-Jones and D Lester. *Implementing Functional Languages.* Prentice-Hall International Series in Computer Science, 1992.

[Pou88] A Poulovassilis. FDL : An integration of the functional model and the functional computational model. In W A Gray, editor, *Proceedings of the 6th British National Conference on Databases (BNCOD 6),Cardiff, 11-13 July 1988*, pages 215–236. Cambridge University Press, 1988.

[Pou89] A Poulovassilis. *The Design and Implemenation of FDL, a Functional Database Language.* PhD thesis, Birbeck College, London, 1989.

[Pou92] A Poulovassilis. The implementation of FDL, a functional database language. *The Computer Journal*, 35:119–128, 1992.

[QH90] M J Quinn and P J Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 1990.

[Rab93] F Rabhi. *Exploiting Parallelism in Functional Languages: A "Paradigm-Oriented" Approach.* Oxford University Press, 1993.

[RS96] H Riedel and M H Scholl. The CROQUE–model: Formalisation of the data model and query language. Technical Report Nr. 23, University of Konstanz, http://www.informatik.uni-konstanz.de/Schriften, December 1996. ISSN 1430-3558.

[SAS89] M Stonebraker, P Aoki, and M Seltzer. Parallelism in XPRS. Technical Report ERL-M89-16, EECS Department, University of California, Berkley, 1989.

[SC95] D B Skillicorn and W Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28(1):65–83, July 1995.

[Sch93] W Schreiner. On the analysis of list-based functional programs for automatic parallelization. Technical report, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1993.

[sis97] Sisal - a high performance, portable, parallel programming language. http://www.llnl.gov/sisal/SisalHomePage.html, 1997.

[SK91] M Stonebraker and G Kemnitz. The Postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[Ski90] D B Skillicorn. Architecture-independent parallel computation. *Computer*, 23(12):38–50, December 1990.

[SLR+94] M H Scholl, C Laasch, C Rich, H-J Schek, and M Tresch. The cocoon object model. Technical report, ETH, Zurich, 1994.

[SS89] T Sheard and D Stemple. Automatic verification of database transactions. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[ST98] D Skillicorn and D Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123, June 1998.

[Suc95] D Suciu. *Programming Languages for Collections.* PhD thesis, University of Pennsylvania, 1995.

[Suc96] D Suciu. Implementation and analysis of a parallel collection query language. In *Proceedings of the 22nd VLDB Conference, Mumbai (Bombay), India*, pages 366–377, 1996.

[Sun90] V Sunderam. PVM: A framework for parallel distributed memory computing. *Concurrency: Theory and Practive*, 2(4), December 1990.

[SW95] D Suciu and L Wong. On two forms of structural recursion. In *Proceedings of the 5th International Conference on Database Theory, Prague*, January 1995.

[SY97] J Subhlok and B Yang. A new model for integrated nested task and data parallel programming. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Las Vegas, Nevada*, volume 32 of *SIGPLAN Notices*, pages 1–12, July 1997.

[TBW92] 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 Lecture Notes in Computer Science, pages 140–154. Springer-Verlag, October 1992.

[TC88] CA Teradata Corp. *DBC/1012 Data Base Computer Concepts and Facilities, C02-0001-05 Edition*, 1988.

[TCH91] P Trinder, D K C Chan, and D J Harper. Comprehensions: a Query Notation for DBPLs. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, pages 55–70. Morgan Kaufman, 1991.

[Tri91] P Trinder. Comprehensions, a query notation for DBPLs. In Kanellakis and Schmidt, editors, *Proceedings, Third Interntional Workshop on Database Programming Languages, Nafplion, Greece*, pages 55–68. Morgan Kaufman, 1991.

[TS91] V B Tannen and R Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Proceedings of 18th International Colloquium on Automata, Languages, and Programming, Madrid, Spain*, number 510 in Lecture Notes in Computer Science, pages 60–75. Springer–Verlag, July 1991.

[TW88] P Trinder and P Wadler. List comprehensions and the reltional calculus. In *Proceedings of the 1988 Glasgow Workshop on Functional programming, Rothesay, Scotland*, pages 115–123, August 1988.

[TW89] P Trinder and P Wadler. Improving list comprehension database queries. In *Proceedings EEE TENCON, Bombay, India.* IEEE Press, November 1989.

[vB89] G von Bultzingsloewen. Optimizing SQL queries for parallel execution. *SIGMOD Record*, 18(4):17–22, 1989.

[WC95] P Watson and G Catlow. Architecture of the ICL Goldrush. In C A Goble and J A Keane, editors, *Advances in Databases, 13th British National Conference on Databases, BNCOD 13, Manchester, United Kingdom, July 12-14, 1995,*, number 940 in Lecture Notes in Computer Science, pages 249–262. Springer-Verlag, 1995.

[WG95] E A West and A S Grimshaw. Braid: Integrating task and data parallelism. Technical Report CS-94-45, Universtiy of Virgina Department of Computer Science, 1995.

[WLH90] W K Wilkinson, P Lyngbuk, and W Hasan. The Iris architecture and implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, 1990.

[Won94] L Wong. *Querying Nested Collections.* PhD thesis, University of Pennsylvania, August 1994.

[WT90] P Watson and P Townsend. The EDS parallel relational database system. In *PRISMA Workshop on Parallel Database Systems,Noordwijk, The Netherlands, 24-26 Sep*, number 503 in Lecture Notes in Computer Science. Springer-verlag, 1990.