An Approach to Scalable Parallel Programming

Simon Andrew Dobson

Submitted for the degree of Doctor of Philosophy

University of York

Department of Computer Science

18 May, 2012

Table of Contents

Table of Contents i

Table of Figures v

Acknowledgements vii

Declaration ix

Abstract xi

Introduction 1

Chapter 1. Scalable Parallel Computing 5
1.1. What is Scalability? 5
1.1.1. Scalability in Various Guises 6
1.1.2. Précis: Scalability for Programmers and Users 7
1.2. A Scalable Machine 8
1.2.1. Hardware 8
1.2.2. Operating Systems 10
1.3. Programming for Scalable Systems 11
1.3.1. Issues in Scalable Programming 11
1.3.2. Memory Models 13
1.3.3. Concurrency 22
1.3.4. Configuration 25
1.3.5. Object-oriented Systems 26
1.4. A Scalable Programming System 30
1.5. Résumé 31

Chapter 2. An Abstract Machine View of Scalable Parallel Programming 33
2.1. Abstract Machines 34
2.2. Programming Environments as Abstract Machines 36
2.2.1. Programming Languages as Abstract Machines 36
2.2.2. Toolkits in Scalable Programming 38
2.3. Memory as an Abstract Structure 39
2.3.1. Object-oriented Memory 39
2.3.2. Distributing Abstract Memory 44
2.3.3. Concurrency Regulation and Memory 44

2.4. The Scalable Abstract Machine 45
2.5. Résumé 49

Chapter 3. Implementing Scalable Typed Memory 51
3.1. Requirements 51
3.2. Partitioning: Representing Scalable Memories 52
3.2.1. Overview of Partitioning 52
3.2.2. Managing Data Access 53
3.2.3. Managing Distribution 53
3.2.4. Resolution 54
3.2.5. Parameters Affecting the Distribution Architecture 55
3.2.6. Generic Structure of Partitioned Collections 59
3.3. A Kernel of Partitioned Storage Architectures 61
3.3.1. Arrayed Storage 62
3.3.2. Associative Storage 68
3.3.3. Directed Storage 77
3.3.4. Mathematical Structures 81
3.4. Creating User-level Data Structures 82
3.4.1. Customisation and Refinement 83
3.5. The Semantics of Failure 87
3.6. Résumé 91

Chapter 4. Concurrency in Scalable Systems 93
4.1. Concurrency in Object-oriented Systems 94
4.2. Concurrency Control 95
4.2.1. Concurrency Control in Object-oriented Systems 96
4.2.2. Concurrency Control and Scalability 98
4.3. Concurrency Regulation 104
4.3.1. Approaches to Concurrency Regulation 104
4.3.2. Regulating Concurrency in a Scalable Environment 105
4.3.3. Concurrency Regulation in the Partitioned Model 106
4.4. Résumé 111

Chapter 5. Phœnix: a Prototype Environment 113
5.1. The Structure of Phœnix 114
5.2. The Host Language and Environment 115
5.2.1. Design Issues 115
5.2.2. The Phœnix Pre-processor 116
5.3. The Virtual Machine Layer 119
5.3.1. Design Issues 119
5.3.2. Implementation Overview 119
5.4. The Partitioned Environment 123
5.4.1. Design Issues 123
5.4.2. Basic Classes 125
5.4.3. Collections 127
5.4.4. Partitions 128
5.4.5. Activities 129

5.5. Extensions 130
5.5.1. Issues in Extending Phœnix Classes 130
5.5.2. Example Extensions 130
5.6. Résumé 132

Chapter 6. Evaluation 135
6.1. The Partitioned Object Model 136
6.1.1. Meeting the Aims of Scalable Memory 136
6.1.2. A Comparison of Possible Alternative Implementations 138
6.1.3. Some Problems with the Chosen Implementation 140
6.1.4. The Programming Model and Method 141
6.2. The Phœnix Prototype 142
6.2.1. Sufficiency of the Base System 142
6.2.2. Extensibility 142
6.2.3. Refinement 144
6.2.4. Defects 147
6.3. Performance 151
6.3.1. Theoretical Performance 151
6.3.2. Experimental Performance 154
6.3.3. Discussion 160
6.4. Three Examples 161
6.4.1. Example One: the Booch Components 161
6.4.2. Example Two: a Cellular Automaton 164
6.4.3. Example Three: an Inference System 170
6.5. Scalable Memory, Partitioning and Phœnix: a Judgement 175
6.6. Résumé 176

Chapter 7. Conclusions and Further Work 179
7.1. Réprise 179
7.2. Further Work 182
7.3. Conclusion 184

References 187

Appendix A. A Formal Treatment of Partitioning 197

Appendix B. Wisdom 207
B.1. The Wisdom Nucleus 207
B.2. The Filing Systems 209
B.3. Wisdom in Use 210

Table of Figures

Figure 1: Page faulting in distributed shared virtual memory 16
Figure 2: The workings of object-oriented memory 41
Figure 3: Two paradigms for concurrent processing 46
Figure 4: Collections and concurrency in the scalable abstract machine 47
Figure 5: Flat distribution management 55
Figure 6: Hierarchical distribution management 56
Figure 7: A generic partitioned collection 60
Figure 8: The general case of resolution 61
Figure 9: Arrays as metric spaces 63
Figure 10: Distribution and resolution in dimensional decomposition 64
Figure 11: Distribution and resolution in regional decomposition 66
Figure 12: Dynamic hashing: splitting a bucket 71
Figure 13: Extendible hashing: splitting a bucket 72
Figure 14: A comparison of extensible hashing schemes 73
Figure 15: Distributed extensible hashing: splitting a bucket 74
Figure 16: A strategy for naming graph nodes 80
Figure 17: The effects of a node failure on resolution 88
Figure 18: Generic collection with replicas 91
Figure 19: Different styles of asynchronous method call 95
Figure 20: Attaching activities to a collection 108
Figure 21: The Structure of Phœnix 114
Figure 22: The compilation process 116
Figure 23: Control flow for resolution in Phœnix 124
Figure 24: Analysis of costs involved in accessing data 153
Figure 25: The Wisdom nucleus 208
Figure 26: A Wisdom system in use (showing load balancing) 210

Acknowledgements

No programme of research occurs in a vacuum: the participation of other researchers as both supportive friends and knowledgeable critics is invaluable. For this reason a few acknowledgements are in order.

First and foremost to Andy Wellings, my supervisor over the last three years, for preventing me from going down any of the several "blind alleys" which I encountered and for correcting some of my less precise flights of English.

Secondly, my thanks must go to the members of the Department of Computer Science at York, who have provided a most stimulating and friendly environment in which to conduct research. The other members of the Wisdom group – Kevin Murray and Paul Austin – acted as excellent teachers in the vagaries of Transputer systems, whilst Martin Atkins and Paul Butcher made useful and constructive comments on some of the language ideas.

Finally my thanks go to all my friends both within the department and outside who stopped me spending *all* my time working. Thanks especially to Andy and Karen Coombes, Matt and Jacqui Cuttle, Dave Cattrall and Dave Scholefield.

The use of all trademarks, copyrighted symbols and quotations is gratefully acknowledged.

Declaration

Some parts of this thesis have appeared in previously-published work. The initial ideas for the partitioned object model were presented in:

> S.A. Dobson and A.J. Wellings, *"Programming highly parallel general-purpose applications"*, pp. 49-56 in Collected position papers of the BCS workshop on abstract machine models for highly parallel computers **2**, University of Leeds, 25-27 March 1991.

A more complete discussion of the principles of partitioning, together with implementation details and example program fragments, appeared as:

> S.A. Dobson and A.J. Wellings, *"A system for building scalable parallel applications"*, pp. 215-230 in "Programming environments for parallel computing," ed. N. Topham, R. Ibbett and T. Bemmerl, North Holland Elsevier, 1992.

Appendix B is adapted in part from Austin[11], taking into account improvements in, and experiences with, Wisdom since his survey was produced. It also includes a brief survey of Austin's work.

Abstract

Current parallel programming environments are inadequate in dealing with the problems introduced by highly parallel, highly scalable, general-purpose computing systems. They fail to tackle problems of conceptual modelling, distribution management and concurrency regulation which are central to the effective use of such systems.

It is argued that sophisticated models of memory and processing hold the key to scalable programming, by allowing applications to be written in an abstract framework which may then be mapped transparently onto an underlying scalable machine. A model of memory based around scalable typed abstract memory modules is developed, which allows applications to create and manipulate arbitrarily large collections of data, independently of the collection's distribution, and to use these collections as an infrastructure for creating and regulating concurrent activity.

A set of implementation techniques for representing scalable memories is developed, covering the commonly-encountered forms of memory. Concurrency control and regulation in such architectures are considered. A prototype programming environment based on these techniques is presented and discussed. The abstract model, implementation architectures and prototype are then evaluated

The evaluation shows that scalable memory is indeed a viable programming solution for scalable systems, simplifying the construction and configuration of a range of parallel applications. However, the prototype environment is shown to be deficient in several key respects. Future work is proposed to rectify these faults, thereby creating a realistic environment for scalable programming.

If I'd the knack
I'd sing like
Cherry flakes falling.

Basho

Introduction

If we offend, it is with our good will.
That you should think, we come not to offend,
But with good will. To show our simple skill,
That is the true beginning of our end.

Quince in *A Midsummer Night's Dream*

The course of computer science, over its short history, might be described as a conflict between two sometimes contradictory aims: increasing the power of computers in order to tackle harder problems, whilst finding better ways to express solutions to these problems.

It is a truism that computer users have an insatiable demand for more computing power, both to address more complex problems and to increase the sophistication of user interfaces. However, there are limits – both physical and fiscal – to the performance which may be obtained from a single processor. Most modern computers make use of dedicated support processors for input/output and floating-point arithmetic; recently there has been much interest in systems where a number of general-purpose processing elements are connected together. In particular, systems of processor-memory pairs connected by a sparse network of point-to-point links have become popular.

Multicomputers have several highly attractive features. Firstly, each processing element may be built from affordable off-the-shelf hardware rather than from expensive custom units. Secondly, processors may be added to a system as required by simply connecting them to the existing system with additional communications links. The architecture is thus both highly scalable and massively parallel, and allows extremely high-performance machines to be built at a reasonable cost – which in turn allows highly computationally intensive problems to be tackled.

Programming language evolution is essentially concerned with finding new models within which programmers may express applications. In general, the trend has been towards greater levels of abstraction in programming languages, taking programmers away from machine-oriented concerns and allowing applications to be

expressed in terms which are close to the programmer's conceptual model. The price for this simplification of programming is generally a decrease in a program's run-time efficiency, as more demands are placed on the compiler and run-time system.

Hence there is a conflict: on the one hand, applications programmers wish to exploit the power of the new multicomputer machines in order to create more complex applications; on the other, the languages which would best allow these applications to be expressed incur (sometimes unacceptable) performance penalties. In the quest for obtaining the best possible efficiency, programmers have often been forced to return to the low-level practices which high-level language evolution has sought to banish, but multicomputer architectures – especially their distributed memory and massive parallelism – make high-quality programming difficult for any but the most trivial and regular applications.

Programming for scalable multicomputers thus presents a classic paradox: the features which make the architecture attractive are precisely those features which make the creation of applications most difficult.

Intentions of this Thesis

The goal of the research described in this thesis is to develop a new method of programming by which the power of the multicomputer architecture can be harnessed. It is intended to explore the notion that a high-level, application-oriented view of programming – concentrating on the structures which are of most use to the application developer, rather than simply on those which are most attractive from a theoretical standpoint – would considerably simplify the programming task. The overall aim is the production of the theoretical framework for programming scalable applications, together with a prototype programming system. Five related problems need to be satisfactorily tackled:

- managing and co-ordinating large quantities of structured data in a distributed-memory environment;
- regulating and controlling of massive amounts of concurrent activity;
- hiding architectural details from programmers through the use of an abstract programming model;
- providing a supportive programming framework with scope for re-use, to avoid unnecessary re-invention; and
- ensuring scalability by ensuring that applications can take advantage dynamically of whatever resources are available at run-time.

The intention is to produce a programming environment to run on top of a suitable scalable parallel operating system: the work does *not* address operating system issues such as process management or load balancing, although these are crucial to the success of the system.

Thesis Structure

Any new course of work must first begin with a thorough study of what has gone before. Chapter one, therefore, contains a survey of the literature of parallel computing. It covers three broad fields – parallel architectures, operating systems for parallel computers, and parallel programming systems – but approaches each by considering from the start the scalability of each element and its relationship to other elements. As a result of this survey, some shortcomings in the programming models used on previous highly parallel systems are observed.

Chapter two addresses these shortcomings by describing a programming model specifically aimed at scalable parallel programming. The model's emphasis is on the transparent and scalable use of resources, and on the ease with which applications may be created and re-used. The central theme of the model is the use of scalable memory as a basis for programming. Memory is seen as being typed, being similar to the data structures commonly found in applications.

Chapter three presents a set of techniques which aim to implement scalable memory efficiently. Several alternative implementation strategies are considered: the one chosen makes extensive use of fully distributed algorithms to avoid performance bottlenecks. The effects of various parameters for distribution are considered. Three variations on the main theme are described to implement three common forms of storage – arrayed, associative and directed – which may form the kernel of a programming environment.

Chapter four considers the problems and consequences associated with the introduction of large amounts of concurrency into applications. Scalability demands a high degree of system involvement in the creation and location of processes, and these problems are addressed *via* the scalable memory implementations described earlier. Concurrency control is also discussed.

Chapter five presents *Phœnix*, a prototype programming system based around the abstract model. Phœnix realises the implementation techniques discussed in chapter three as a set of classes for use in constructing object-oriented applications. The use of object-oriented techniques allows a substantial amount of code and design re-use to occur both between classes and across applications – a feature considered vital for the successful programming of complex parallel systems.

Chapter six contains an evaluation of the Phœnix prototype. The evaluation approaches Phœnix from three directions: in terms of efficiency, programmability and abstraction. The weak points of the system are highlighted and analysed.

Chapter seven concludes the thesis with a résumé of the work described. It comments upon the decisions taken during the work, discusses the design of programming languages and operating systems for an "ideal" scalable parallel system, and points to some directions for future research.

Some Conventions

There is currently a healthy debate within the programming language community as to the exact meanings of the common terms *concurrent*, *parallel* and *distributed*. These words have been used in contexts so disparate as to destroy their

usefulness. Without wishing to contribute to this debate, we shall henceforth adopt the following convention:

- a *concurrent* system is a system in which several loci of control may be active simultaneously, at least from the programmer's conceptual viewpoint;
- a *parallel* system is a concurrent system in which the number of simultaneously active processes is very large – of the order of hundreds or thousands; while
- a *distributed* system is a system built from a number of largely independent computers connected by a network, so that nodes do not share memory.

A similar debate exists around the term *scalable*. As the meaning of this term is central to this thesis, it is discussed in chapter one as the basis for the literature review.

Within this thesis, all fragments of code, class definitions and the like appear in `Courier` typeface.

Chapter 1.

Scalable Parallel Computing

Half of the people can be part right all of the time,
Some of the people can be all right part of the time,
But all of the people can't be all right all of the time.
I think Abraham Lincoln said that.

Bob Dylan, *Talkin' World War III Blues*

There exist many surveys of the field of parallel processing (for example[95][106][109]), but few have considered the scalability of systems as their primary organisation. This is our aim here.

The concept of scalability is very important in understanding the aims of the current work, so we shall begin by defining the term and analysing the ways in which it may be applied to different computing tasks. We shall then use this definition to examine the literature appertaining to the creation of parallel applications, concentrating on the available operating systems and programming languages. We shall investigate the advantages and failings of different systems when considered for scalable programming, and shall derive from this investigation some factors which are characteristic of programming system suitable for scalable computing.

1.1. What is Scalability?

The term “scalable” has different meanings to different people, and we shall first define it more precisely.

The idea of scale derives from notions of measurement. We may speak of things being “on different scale,” such as a mountain and a mole-hill, where the same

characteristics are apparent at two different orders of magnitude; and of "changes of scale" when a phenomenon occurring in small objects manifests itself in larger domains. We may also speak of things which are constant across a range of scales, either qualitatively or quantitatively.

Essentially, something is *scalable* if, without altering its gross characteristics, its size may be increased or (less commonly) decreased. That is, the phenomenon can deal with changes of scale without apparent change. The key word here is "apparent": scalability does not imply that no changes occur internally but simply that, to the outsider's view, the system being studied remains qualitatively the same across a range of problem sizes.

Scalability offers a number of advantages. It allows underlying factors to change without these changes propagating beyond the scalable system's boundaries; allows systems to be compared across changes in size; and allows a system to be applied to a range of different scales, which in turn implies that the scale of problem is not relevant to its solution. Thus scalability is an important form of abstraction.

Another important facet of scalability is that it tends to be an emergent property: when a group of smaller items are collected together, scalability arises from their interactions. The scalability of a system is *not* latent within the components of the system , but arises solely from their interconnection and interaction.

1.1.1. Scalability in Various Guises

The most commonly-mentioned form of scalable system is represented by a class of algorithms which may be applied to problems of radically different sizes. Coming as it does from computational complexity theory, this form of scalability is pervasive. A common statement is that, for an algorithm or system to be truly scalable, it must have a computational complexity of no more than $\log n$ (or sometimes $n \log n$) for a problem of size n. This implies that, for a system to be scalable according to this definition, *no important algorithm or structure must have a complexity greater than this*, otherwise this item will become a brake on the system as it grows – a "bottleneck." A good example might be a system which used a bubble-sort in one of its fundamental algorithms: since bubble-sort has a complexity of order n^2 [70], this algorithm will bottleneck as the number of elements rises.

Another frequently-encountered scalable system is the interconnection network of a multicomputer. The scalability in this case applies to the complexity of routing messages between nodes: since communication is so fundamental to multicomputer systems, it is vital that the hardware which controls message routing is scalable – otherwise there is a maximum number of nodes representing the point of diminishing returns, beyond which the addition of more nodes will be counter-productive due to communication delays. The hypercube architecture[55], with a communication complexity of $\log n$ for an n-node system, thus has "perfect" scalable communications characteristics.

A third example of scalability is found in the creation of operating systems for scalable computers such as multicomputers. Scalability here refers to the ability of the operating system, seen as a whole, to support any number of processors without alteration. This, however, differs from the above in that it is scalability from an

external viewpoint only: the operating system can scale across a range of hardware configurations without changing qualitatively to users, programmers and applications, although quantitatively it will have changed to provide more processing power, more memory *et cetera*.

This is a good illustration of the emergent nature of scalability: such an operating system is only scalable because all its components are themselves scalable, in isolation and in conjunction.

There is clearly an incremental dimension to all these forms of scalability. It is important to be able to vary the system at an arbitrary rate, without any "jumps." A scalable system must support the range of possible scales continuously and gradually.

1.1.2. Précis: Scalability for Programmers and Users

If a system may transparently accommodate changes in problems sizes, the size of problem to which the system is applied has effectively been abstracted away from.

For the programmer, having a scalable system means that the same software system may be used on problems of arbitrary size, since the system will scale to deal with the problem without programmer intervention. For users a similar benefit accrues: a scalable application may deal with any problem which the user sets, regardless of its complexity.

A good example of a scalable application would be a "sort" command. There are a great many sorting algorithms, but their efficacy often depends upon the problem size being tacked: bubble sort is more efficient than quick sort on small problems, but less so on large data sets. Therefore a scalable sort command would choose the most appropriate algorithm for the job being requested, and would appear to users as a perfectly scalable system able to respond to the demands being placed upon it. The alternative is to offer either a command with unpredictable (and possibly unacceptable) performance or a selection of commands from which the user must choose the correct one.

Another example would be a parallel system in which the degree of parallelism varies depending upon run-time conditions. The selection of how much parallelism to use on a problem, and how much distribution, is a complex one, and is further complicated by systems whose capabilities may be scaled. A scalable application must be able to take advantage of whatever resources are available at run-time.

However, for the programmer this view of scalability also introduces problems, since it implies that all applications software must be scalable. The familiar view of software components as "black boxes" exporting a functional interface must be extended with the notion of scalable black boxes which may be used on problems of any size.

1.2. A Scalable Machine

The purpose of this chapter is to review the current state of the art in constructing highly parallel systems, with a view towards the scalability of existing

techniques. We shall begin our examination of scalability by looking at the hardware platform being used.

1.2.1. Hardware

The construction of a scalable machine involves building a computer system which is capable of extension to provide additional resources and processing power. Ideally one would like to be able to scale a machine's capabilities incrementally, so that it may be expanded gradually to meet changing requirements. Indeed, this might be seen to be *the* major advantage which scalable machines have over more traditional centralised systems: they may be extended as requirements – and finances – allow.

The classification of machine architectures due to Flynn[46] divides machines into four categories:

- single instruction, single data (SISD), where a single instruction stream is applied to a single data stream;
- single instruction, multiple data (SIMD), where instructions are applied to many data items in parallel;
- multiple instruction, single data (MISD), encountered only in some advanced digital signal processing chips; and
- multiple instruction, multiple data (MIMD), where separate sets of instructions act on separate data streams.

The SISD category thus encompasses all the traditional single-processor systems; SIMD and MIMD are both (potentially) highly parallel, and are both candidates consideration as scalable machines.

SIMD systems such as the Connection Machine[58] usually exhibit a massive amount of parallelism – often over eight thousand simple processors are used, far more than in any current MIMD machines. It is not usually possible to add either memory or processing elements to such machines – although there is no theoretical reason preventing this – and they will not be considered further.

Processors and Memory

Prior to the introduction of systems with multiple processors the only way to increase the performance of a system was to purchase a new central processor: an expensive and often frustrating business.

Multiple-processor systems allow an easier upgrade path: processors and memory may be added incrementally to the system, without altering its essential characteristics. For example, a multiprocessor having eight processors is qualitatively the same as a system having four processors – but faster, with more memory, and able to support more users, more complex problems, or both. Such systems are therefore, by the preceding definitions, scalable with respect to their processing capacity; they are not, however, scalable indefinitely, as contention for

the shared memory and shared buses place an upper-bound on the number of processors which may be connected to the system (typically around twenty[1]).

Multicomputers tend to offer better processor scalability in this respect, since they have no shared memory over which contention can occur. It is thus possible to add processor-memory pairs without affecting the access behaviour of other nodes: in many respects the nodes act as independent computers (hence the term "multi(ple)computer") with peripheral inter-node communications capabilities.

Interconnection

In a multiprocessor, nodes communicate *via* the shared memory; multicomputers require some extra communications mechanism to connect nodes together. Adding more processors introduces more communications traffic into the system, and it is now well-recognised that it is communications bandwidth – and *not* processing capacity – which is the limiting factor in building massively parallel systems.

The processor interconnection may take the form of a shared communications medium (such as a broadcast network), but such an architecture acts as a brake on scalability: as the number of nodes on the network increases, so does the network traffic, but the amount of communications bandwidth remains the same: hence eventually the network will become saturated[87].

The alternative is to provide a "sparse" network and (possibly) software support for complete connectivity using channels, capabilities *et cetera*. The usual approach is to associate communications capabilities intimately with processing nodes, with each node being a processor-memory-communications triple. This allows the amount of communications in the system to scale alongside the increase in processing elements.

Hypercubic networks are often presented as the epitome of scalable networks: nodes are arranged to form the vertices of a hypercube (typically between a three- and a nine-dimensional hypercube), with network links forming the edges. An n-dimensional hypercubic network has the property that exactly n links are incident on each node, and that the average distance between any two nodes (the *communication distance*) is $\lceil \log_2 n \rceil$ links. By comparison, a mesh-based network has a communication distance of $\sqrt{n}$, which is considerably more complex (and hence less scalable).

However, this idealised view of routing is less than perfect when considering the other aspect of scalability, incremental development. In a hypercube, the dimensions of the network is directly visible at each node in terms of the number of links: increasing the number of dimensions impacts upon every node in the system, since more links will need to be added to maintain the topology. By contrast, a mesh-based system is more scalable in this respect. Nodes may be added at the edges of the mesh without affecting, in hardware terms, nodes in other parts of the network.

[1]Multiprocessors such as the Monarch[102] allow thousands of processors to be connected, but only at the cost of greatly increased memory access times.

1.2.2. Operating Systems

Many parallel computers are run "naked," without any operating system support. The rationale behind this decision is that use of an operating system inevitably introduces overheads which may be avoided by direct-access by applications to the hardware. Frequently the only supporting software provided is a communications harness such as Tiny[38].

Such approaches are acceptable only in limited circumstances: when only a single application is running on the machine, and when the programmer is competent to deal with low-level details. If wider access to parallel computers is to be achieved, it is necessary that the parallel system presents a similar interface to that of a "standard" machine: this implies allowing applications to co-exist on the platform, the existence a filing system, a shell – in short, an operating system.

A distinction may be drawn here between *distributed* and *parallel* operating systems, echoing the conventions of terminology from the introduction. A distributed system tends to have the goal of increased reliability, increased availability and the like; a parallel system's main rationale is to increase performance. A *scalable* operating system is in many respects a fusion of the two: it may use the availability of multiple processors to increase its reliability and to improve performance for applications. Many of the techniques of distributed systems – especially those involving failure management, filing systems and name spaces – are also applicable to scalable parallel systems (see especially the work on Plan 9[15][97][99]). A good overview may be found in [92][110].

Helios

Helios is a commercially-available operating system for networks of Transputers. It is POSIX[2]-conformant, and supports load-balancing through the use of specialised tools.

Helios is composed of six elements: a kernel, several libraries, a processor manager and a loader. The system supports several advanced Unix ideas such as shared libraries, mountable file systems *et cetera*.

Programming in Helios closely follows the Occam model of communication: processes may declare a number of channels which are then attached to the channels of other processes using a configuration script. The channel abstraction is fused with the Unix notion of file streams, so applications may use the standard Unix notations to access other processes *via* their channels.

Wisdom

Wisdom[8][10][90][91][92] is a micro-kernel operating system nucleus for mesh-based systems, currently implemented on Transputers.

The Wisdom nucleus is composed of three modules: a load balancer, a namer and a router. The same nucleus runs on every node in a Wisdom system.

Parallelism comes in two forms: tasks and processes. A task is the smallest unit of true parallelism, and may be composed of several processes executing in a shared

[2]POSIX is the international standard definition of a Unix programming interface (IEEE 1003.1-1990 or ISO 9945-1).

address space. The load balancer allows tasks to be moved at their creation onto any less-loaded neighbour of the process which creates the new task. This forms a load balancing "ink blot" of tasks spreading out from the user's initial log-in task.

The router allows any pair of tasks to communicate using capabilities[88]. A capability is a user-space object which may be passed freely between tasks: any task holding a copy of a capability may use it to communicate with the task which created the capability. Message routing is unreliable, with zero-or-once delivery.

The namer is analogous to the Unix file name space, but maps textual, user-friendly names onto capabilities rather than onto files. Any task may "register" itself with the namer by supplying a capability: any other task may then obtain this capability by making a request to the namer.

The modules of all the nuclei in the system co-operate to present the illusion of a single computer, but additional nodes may be added to the mesh to increase the system's capabilities without changing any software.

A more complete description of Wisdom may be found in Appendix B.

1.3. Programming for Scalable Systems

We shall begin by defining the issues which set scalable programming systems apart from other parallel or distributed environments, before addressing the issues raised – memory, concurrency and configuration – by a survey of the literature. Inevitably some systems fall into several categories: in this case, a system has been placed in the category to which it makes the biggest contribution. Object-oriented systems span categories to such an extent that discussion of them is contained to its own section.

1.3.1. Issues in Scalable Programming

A scalable application is an application which is able to take advantage of whatever computational resources are available at run-time, and is able to tailor its resource utilisation and organisation according to the problem in hand. In other words, a scalable application is extremely responsive to its environment when dealing with a problem.

Such responsiveness implies that a very flexible approach is taken towards all those features of the system which may be changed by scalability. In particular, a scalable applications must deal with the fact that the amount and distribution of memory and the amount of usable parallelism may change between runs.

Memory

Memory in a multicomputer is divided between the processing nodes. A memory is private to the node to which it is connected, and only processes executing on that node may access its memory directly.

This partitioning of the system's address space has several effects. The most beneficial effect is that bottlenecks caused by several processors competing to access memory are eliminated. The processing nodes of a multicomputer each behave like

independent computers – with the addition of communication links and the removal of most or all peripheral devices.

The disadvantages, however, are severe. An application must constantly consider the *location* of data items and processes, in terms of the node on which they reside. This need for location-management adds considerable complexity to the programming task and introduces overheads: since a process may only manipulate data held in its node's memory, any data not residing there must be acquired from the remote node where they reside.

Scalability means that an application cannot know, *a priori*, exactly what memory resources will be available: additional nodes may be added, local memory sizes may be increased, other applications may be running *et cetera*.

Concurrency

Parallelism is the *sine qua non* of scalable computing. Without parallel execution, an application is limited in its performance and memory to what can be accomplished on a single processor.

Concurrency may be seen in two lights: one the one hand, it may be used to improve performance by allowing several parts of a computation in parallel; on the other, it may be used to provide a degree of redundancy in processing, so as better to tolerate partial failures. Both these aspects are important in a scalable system.

The fact that processors may be added – that is, the processing resources scaled – means that applications must take a very flexible view of these resources if they are to be scalable. It is unacceptable to define *a priori* the number of processes which will be created by an application, as this limits the scalability of the system: less processes than processors may result in less-than-optimal performance, whilst too many processes may cause contention and time-slicing problems. There is a difficult line to be walked between the programmer's involvement in generating concurrency and the potential scalability of the system.

Configuration

Configuration is the arrangement of elements of a program in memory, and their interconnection. In a system in which the number of processors and their topology may vary, this is obviously a difficult task.

There are essentially three approaches to configuration:

- "hard-wiring" an application's configuration into it at compile-time;
- separating an application's functionality from its configuration, and specifying configuration using a separate tool; and
- allowing configuration to occur automatically at run-time.

The first approach means that changing an application's configuration requires a complete re-compilation – not a viable option in a scalable system. The second allows configurations to be changed with only minimal changes (possibly re-writing the configuration description, but not the application itself). The third allows an application to decide itself upon its configuration, relegating the programmer's

involvement to the specification of "hints" to help the mapping. This last is the most appropriate to scalable systems, since the programmer is never called upon to specify directly the resource utilisation of the application.

1.3.2. Memory Models

There are several ways in which memory may be presented to the programmer. These range from the visible partitioning of the machine's address space to the complete abstraction of all notions of data location.

Explicit Data Mapping

The most primitive form of memory model for a multicomputer system is one which allows the partitioning of the system's address space to be seen by applications. Another way of looking at this is that any system address is composed of a pair *(node, addr)* which defines location *addr* on machine *node* (which may itself be a structured name). This is rather reminiscent of the segmented address space found in some processors, and has the same disadvantage: there is an arbitrary upper limit to the size of the largest contiguous block of memory which may be allocated. A further disadvantage is that the system's memory is not random access: addresses on a different node are more expensive to access than local addresses.

The single advantage of this form of memory model is that it allows the most precise control possible over the placement of data and code. In order to achieve the best possible performance, some programmers are willing to accept the penalty of increased application development and debugging times as a trade-off against better speed of execution in the final application, which will not be incurring any penalties from supporting a more high-level memory model.

The best-known explicit data mapping system is undoubtedly Occam[86], which is covered in more detail in §1.3.3. Other systems, such as Concurrent C[51] and Joyce[27], also use the same model.

Data Structure Mapping

As an alternative to such explicit mapping, some systems allow data structures to be mapped automatically onto processors. This has the great advantage that many applications – particularly in the engineering community – are based around manipulating large arrays or other highly structured data collections, so the mapping concerns the central constructs of an application.

Different systems have differing degrees to the transparency with which they perform their mappings: some perform it completely automatically, or with the addition of annotations, whilst others allow the structure of the mapping to be visible to (and exploited by) applications.

Parallel Fortran

Even after forty years, Fortran is still the *lingua franca* of the scientific community. There is a vast amount of Fortran software in existence – both as

applications and as libraries of sub-routines for common calculations – much of which is in the form of "dusty decks" of code which it is impossible (or impractical) to translate into another language. For all these reasons, there is a great deal of interest in running Fortran code without modification (or, more realistically, with only trivial annotation) on parallel systems. Many of the common sub-routine libraries have been ported to the new dialects[3].

There have been a great many attempts to create a parallel Fortran, mostly concerned with introducing parallelism into nested loops. The usual mechanism is to perform a data dependence analysis on the Fortran source code to determine how the values of one loop iteration affect other iterations, and then to use this information to construct a suitable parallel version of the loop and a suitable distribution for data and processes.

The recent High Performance Fortran (HPF) proposal[57] contains directives to specify application-defined topologies of processors coupled with mappings of arrays (the only applicable data structure in Fortran) onto these processors. Parallelism comes from "forall" statements and directives specifying that certain loop segments are independent and may proceed in parallel. There is also the notion of a "pure" function having no local state dependence, which may safely be applied in parallel.

Topologies

Topologies[104] are an operating system construct which allow a single object to be distributed across the nodes of a network. Each topology is an object having a well-defined interface, a list of "vertices" at which portions of the object (both code and data) may reside, and a connection topology for the vertices.

A topology is instantiated by specifying a mapping of its vertices onto processors. When communicating with the object, processes may specify a particular vertex to which they wish to bind, and all their communication will be directed directly to this vertex.

Each vertex must be equipped with a procedure which can determine whether the portion of the object being accessed is located at that vertex, and must forward the request if this is not the case. The internal organisation of the topology is visible to its clients.

Kali

Kali[71] separates the description of a program into three parts: a description of the processor topology, some data structures which may be distributed, and a collection of parallel loops over the elements of those data structures. Programs are written under a shared-memory model, but the distribution of slices of data structures onto the processor topology is specified explicitly. The compiler than transforms the program into a collection of message-passing processes. In essence, the system is a forerunner of HPF.

Parallelism comes from "forall" constructs which allow a command to be executed over all (or a sub-set of all) the elements of an array. The site at which execution occurs may also be given, and may change with each iteration.

Distributed Shared Virtual Memory

Distributed shared virtual memory (DSVM) is an increasingly popular model of memory[3]. It attempts to provide the abstraction of a single globally-shared memory on a distributed-memory machine by using the techniques developed for virtual memory[110], extended into a wider domain. In the following discussion, we shall use the DSVM system described by Li[76] as representative of a wider class of such systems: a comparison of various DSVM systems, with each other and with other models of memory, may be found in [114].

Memory is represented by fixed-length *virtual pages*. Each page may be resident in memory or may be temporarily paged-out onto disc. The physical memories of the processing nodes are divided into *page frames*, each of which may hold a single virtual page. Address translation is used to map the *virtual addresses* generated by processes into physical addresses: different processes may have independent virtual address spaces, just as with standard virtual memory systems.

The chief characteristic of DSVM is its ability to migrate and replicate pages in different physical address spaces. Since there is no fixed mapping between virtual pages and physical addresses or locations, it is possible to move a page between page frames on different processors in response to requests for addresses on that page – a *page fault*. If a process generates a page fault, the DSVM manager determines where the required page is located and either moves or copies it into a page frame on the processor which generated the fault. In order to do so a page frame may need to be freed: this may be accomplished by (for example) discarding the least-recently-used replica of any page.

There is a problem of memory consistency, however. The definition of DSVM states that it will be *strongly consistent*: the value which a process reads from a memory location will be the value *last* written by *any* process into that location. If two processors were to be allowed write access to a single page, consistency problems would arise.

DSVM solves this by associating an access permission with all pages. A page may be marked as read-write or read-only. There may be many read-only replicas of a page, but at most one read-write replica (known as the *master copy*). If any replica of a page is written to, all the read-only replicas of that page are *invalidated*: nodes holding these replicas throw them away, so the next attempt to read data from that page causes a page fault which will acquire the updated data on that page from the master copy. A typical sequence of accesses is shown in figure 1.

[3]Unfortunately known by a variety of names, another common one being virtual shared memory (VSM).

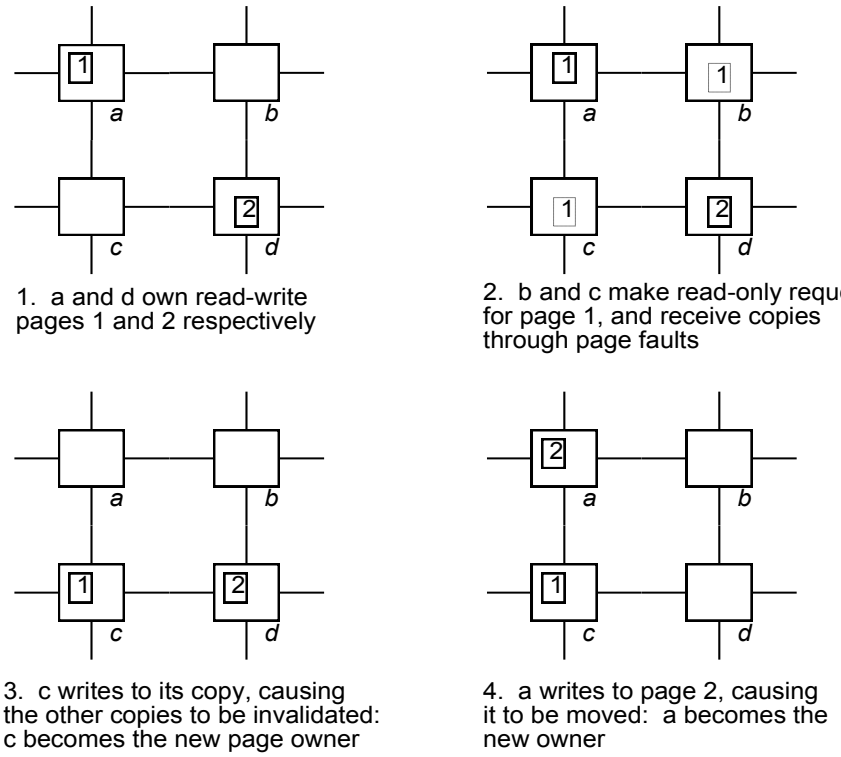


Figure 1: Page faulting in distributed shared virtual memory

Li investigated several strategies for managing pages and replication, and derived a fully distributed scheme which ensured consistency between replicas of pages[77]. He also investigated the overheads involved in implementing a DSVM system practically on a hypercube architecture[78], drawing the conclusion that the system was practical on networks with a high dimensionality: it is doubtful that his methods would function as well in systems with a low-dimensional interconnection.

DSVM seems to be an ideal candidate for a scalable system: it successfully allows applications to abstract-away from details of data location, since all pages are accessible from all nodes. It has a number of shortcomings when examined more closely, however.

Firstly, there is the question of allocation of data to pages. Consider a pair of processes A and B which are communicating using a shared variable: A is a producer and B a consumer. A writes values into the variable which B reads and processes. The processes reside on different processors, with the variable being mapped into a virtual page. Communication then occurs as follows. A owns a read-write copy of the page, while B owns a read-only copy. When A writes a value to the variable, B's replica of the page is destroyed; when B next attempts to read the variable, it will cause a page fault and acquire the value from A's master copy.

Now consider another pair of processes, C and D, which are interacting with each other in exactly the same way as A and B using another variable, and which are located on another (different) pair of processors. C and D do not interact with A and

B in any way, as long as the shared variable used by A and B is located on a different virtual page to that used by C and D: otherwise, when A writes to its copy of its variable it will not only invalidate B's copy (as above) but also C's and D's as well; the same applies when C writes to its variable. Each write by *either* pair causes the *other* pair to cause a page fault at the next access to its variable – even though the processes are unrelated!

This phenomenon, known as *false sharing*, mandates that the variables used by unrelated processes are mapped onto different pages, using a very sparse allocation of data to pages. This is quite acceptable using the assumptions of DSVM, which (tacitly) assume that there is always a sufficient amount of memory available. This assumption may be challenged in a scalable system.

If a DSVM system is to be scalable, it must be possible to increase the number of virtual pages available along with the number of processing nodes, otherwise the number of processors will eventually be left without enough virtual memory. This implies that it must be possible to add new paging discs, and this in turn implies that a page which is paged-out onto disc may be placed onto one of several paging discs. There is an additional decision to be made as to exactly which disc a page is swapped-out onto: it may always be the same disc, or the nearest, or the least loaded *et cetera*. This introduces more complexity into an already complex algorithm.

This problem may be avoided by doing away with paging discs and always holding all pages in memory. At least one copy of every page must always exist at some node in the system (for example the master copy is always be preserved), and discarding a page to free a page frame moves its data to some other node. This may result in a page fault giving rise to a cascade of page movement, which is almost certainly unacceptable. It may, however, be practical if the number of pages virtual pages in the system is much smaller than the number of available page frames.

Munin

An interesting variation on DSVM is the Munin system[16]. Munin offers an object-oriented form of virtual memory, performed on a per-object basis.

The basic observation is that the general case of access to objects – reads and writes occurring with equal frequency – is only rarely encountered. Munin identifies nine categories of access pattern:

- write-once – the object is written to only during initialisation, and is read frequently thereafter;
- private – the object is accessed only by a single thread;
- write-many – writes occur frequently between reads;
- result – all writes are completed before any reads occur;
- migratory – the object is accessed by only one thread at a time, although the thread changes with time;
- producer-consumer – the object is used as a channel between two processes;
- read-mostly – the object is read frequently between writes; and
- the general case of read-write.

No single virtual memory scheme can support all these patterns efficiently (Li and Hudak recognise two of the above cases – write-many and producer-consumer – which are inefficient under their DSVM[78]).

Munin addresses this problem by implementing virtual memory in software on a per-object basis. The system uses a different memory management scheme for each category, and can change the category of an object dynamically as patterns of access change.

Munin applications are written using Presto (see later, §1.3.5), which is usually a shared-memory system but which can function in a distributed-memory environment when coupled with Munin.

The Kendall Square Research KSR1

A recent entrant into the DSVM arena is the KSR1[82]. The machine uses a caching system called ALLCACHE[49], coupled with a very high-speed optical bus to offer massive parallel performance using virtual shared memory.

The ALLCACHE scheme is closely related to that described above, but makes good use of the speed of the machine's communications to update copies of data items cached on other processors as changes occur. This reduces the number of page faults. The bus system is hierarchical to avoid contention.

In programming terms, the KSR1 deliberately separates programming from the machine's physical construction: as with other DSVM system, it is necessary for the programmer to understand the sizes of caches *et cetera* in order to maximise locality of reference and performance, but these vital figures are hidden. An approach of "incremental parallelisation" is advocated, where parallelism is gradually added to existing sequential code.

The major problem with the existing KSR1 is its susceptibility to failure. The crash of a single memory board in the system is enough to bring the entire machine down, since the operating system cannot sustain the loss of a piece of itself, and mean times between failures of 24 hours have been reported[82]. This may be contrasted with a more distributed multicomputer architecture where the operating system nuclei on each node are separate, so the failure of one node – whilst it may render data unobtainable – is not catastrophic to the other nodes. The machine is also based on custom processor and communications technology, which makes it vastly more expensive than systems constructed from "off the shelf" components.

Linda

Linda, also known as the *generative communication* paradigm[2][34][52], is another shared-memory abstraction running on distributed-memory hardware. Rather than use the traditional model of memory as a collection of untyped words, Linda implements its shared memory in a novel, strongly-typed way.

The fundamental element of Linda is the *tuple space*, which is a bag of *tuples*. Each tuple is a typed, ordered sequence of values: for example,

```
{ 1 }
{ "Hello", "World" }
{ "Hello", "World", 22 }
{ "World", "Hello" }
( "Hello", 1.0 }
{ "Hello", 1 }
```

are all valid, distinct tuples. There may be many copies of a single tuple in tuple space at any time.

Tuples may be inserted into tuple space using the `out` primitive, which places a tuple into tuple space atomically. The statement

```
out( "Hello", "World" )
```

places a tuple composed of two strings into tuple space.

Once placed into tuple space, a tuple may be accessed by *any* other process in the applications (or the system) using the `in` statement[4], which removes a tuple from tuple space. Tuples are retrieved using associative matching: a pattern, or template, is provided which is unified against all tuples in tuple space. A template may contain values and *formal* parameters, which are variables which will take on values from the selected tuple. For example, the statement

```
string s
in ( "Hello", ?s )
```

would match the tuple inserted above: after the statement, `s` would have the value `World`. The associative matching algorithm respects the types and arities of tuples, so the statement

```
int x
in ( "Value", ?x )
```

would match the tuple `( "Value", 10 )` but not the tuples `( "Value", 10.1 )` or `( "Value", 10, 1 )`.

Linda provides two other operations. The `rd` statement acts exactly like `in` but does not remove the tuple from tuple space. It also defines an `eval` statement which generates "active" tuples – tuples which have functions as values. An active tuple is the Linda abstraction for a process: when inserted into tuple space, the functions are evaluated to turn the tuple into an "ordinary" passive tuple containing the results of the functions. The functions may access tuple space in the course of their evaluation.

Linda is a sometimes described as a *co-ordination* language. It is *not* a language in its own right, but shares something in common with DSVM or communication models like CSP[59]. Linda is intended to be inserted into another, "host" language,

[4]Some people contend that this use of `in` and `out` is rather paradoxical, since `in` takes tuples out of tuple space. The convention is to view all tuple activity from the process' point of view, so `in` brings a tuple into the process.

which performs all computation but uses the Linda primitives for all shared data and inter-process communication. This allows Linda to be implemented as a library and linked into a program: indeed, the most common Linda system is an embedding of the primitive operations into C[19][33].

In many respects, Linda suffers from the same deficiencies as DSVM: by hiding distribution of data so completely, it makes it impossible to optimise applications when a "good" data distribution is known.

There is a dual abstraction between the way in which data is structured internally and the way in which it is communicated to other processes. process may hold data as (for example) a tree internally, but must "flatten" this tree into tuples in order to communicate it with another process – and the receiving process must reverse this operation in order to manipulate the tree. This may be a complex procedure for realistic data structures.

The semantics of the `eval` statement are stated only very vaguely, and it is difficult to find a really satisfactory answer to the question: in an active tuple such as

```
eval ( "squaring", f(10), g(20) )
```

what are the semantics of the execution of `f` and `g`? One might interpret the statement as either evaluating the functions in parallel, or sequentially. If the answer is the former, then there may be strange and unpredictable interactions between the functions if they both access tuple space in the course of evaluation; if the latter, they may deadlock, and it becomes difficult to determine how much concurrent activity occurs in a program.

There is also the question of matching an active tuple: is this possible, and if so, what are the semantics? There are at least three possible semantic meanings for removing an active tuple from tuple space – delete the process, suspend the process or reverse all its actions up to the point at which it is removed – and without a proper definition it is impossible to reason about a Linda system without knowing the details of its implementation.

Some more recent Linda implementations include more comprehensive features, such as multiple tuple spaces and languages well-integrated with Linda[30].

Strand

Strand[48] is a language based around the ideas found in Prolog[39] and the concurrent logic languages[121].

The shared memory in a Strand application is a database of logical assertions. An assertion may be a simple statement of fact or a rule of inference for processing other assertions, in much the same way as in Prolog. A Strand clause may be *guarded* by an expression which must be true for the clause to be evaluated.

A typical Strand program is the following set of clauses:

```
twice ( X, Y, Status ) :-
   integer(X), X>0 |
      Y is 2*X, Status := [].

twice ( X, Y, Status ) :-
   integer(X), X=<0 |
      Y := 0, Status := [].

twice ( X, Y, Status ) :-
   otherwise |
      Y := 0, Status := error.
```

(This and other examples are taken from [5].) These clauses evaluate `Y` to be `2*X` when `X` is an integer greater than zero; 0 if `X` is less than zero; and raise an error otherwise. The guards on the clauses ensure that only the appropriate clause fires.

Parallelism comes from the evaluation of clauses concurrently in the satisfaction of goals. For example consider the function:

```
quad ( X, Y ) :-
   twice(X, W, S1), twice(W, Y, S2).
```

This definition is read as two processes communicating through the shared unit buffer `W`. Initially this variable is undefined. Both clauses begin execution concurrently, but the second blocks on attempting to acquire the value of `W` (which will be unified with `X` in the definition of `twice`). The first process will evaluate `2*X` and place the result into `W`, unblocking the second process and allowing it to proceed to evaluate `2*W` and place the result into `Y`.

From this example, it may be seen that Strand's view of shared memory is as a collection of clauses coupled with *single-assignment* variables. Two processes accessing the same variable are essentially communicating using a unit buffer: for more advanced communications, Strand provides lists: the clauses

```
par_twice ( [N | LX1], LY ) :-
   Temp is 2*N, LY := [Temp | LY1],
   par_twice(LX1, LY1).

par_twice ( [], LY ) :-
   LY := [].
```

map the function `2*X` across a list in parallel.

Strand also allows multiple languages to be used in writing the functions which appear in clauses. Thus functions may be written in any suitable language, but must communicate using the Strand database. The Strand system handles the unification of shared variables, the scheduling of clauses which may execute, and the assignment of evaluating clauses to processors. Strand itself is designed to be highly portable across a range of architectures and languages[31].

It is impossible to avoid comparisons between Strand and Linda. Both implement shared memory using a novel memory model (Strand's being less novel than Linda's) and deal with the problems of access, matching (using different associative matching algorithms) and process creation and location. Strand also handles process scheduling, in the manner of an OR-parallel logic language (*e.g.* [83]), using single-assignment variables for communication, whereas the Linda programmer defines the execution order; Linda is (potentially) strongly statically typed, whilst Strand is weakly dynamically typed.

As a memory model for a scalable system, Strand suffers the same benefits – and the same deficiencies – as Linda. The abstraction of execution order, however, gives Strand a slight advantage in the management of concurrency, since there is no explicit parallelism in a Strand program.

1.3.3. Concurrency

The concurrency found in applications is often very tightly tied to the intended hardware architecture, although this is not always the case. For scalable systems, we obviously require a scalable model of concurrency, in which the concurrency used in an application may scale according to the number of processors which are available.

We shall consider concurrency in terms of whether it is extracted automatically from a program's text or whether the program must express its concurrency explicitly (a third method, using object interactions, will be deferred until later).

Implicit Parallelism

"Implicit" parallelism refers to the situation where the language compiler extracts automatically any parallelism latent in a problem. The programmer writes the program without any concern for its execution structure, and the compiler automatically generates the necessary concurrency generation and synchronisation primitives. There have been two main threads in automatic parallelisation: extraction of parallelism from sequential languages, and the use of non-procedural languages.

The most popular choice for automatic parallelisation is undoubtedly Fortran. As mentioned above, §1.3.2., parallel Fortran dialects frequently use data structure mapping to distributed elements of arrays and then perform operations on these arrays in parallel. The Fortran program is still essentially a sequential one, but the compiler is free to optimise array operations using parallelism: an SIMD approach which may be implemented on either SIMD or MIMD hardware.

Parallelism extraction in imperative languages is constrained, however, by the implicit flow-of-control information embedded into applications: the order of statements, sub-routine calls and the like has a strong semantic meaning which cannot be altered by program transformations. Non-procedural languages, which (largely) remove the semantic importance of statement ordering, offer another, more tractable route to automatic parallelism. The compiler is free to evaluate functions in parallel at the granularity appropriate to the architecture being used. Hudak makes the point that

> "An often heralded advantage of functional languages is that parallelism is implicit; it is manifested solely through data dependencies and the semantics of operators[61]"

but harnessing this advantage has proved to be more difficult that might have been expected: the same is true of logic languages.

At the risk of generalisation, it might be said that automatic extraction of parallelism – both in imperative and declarative languages – without programmer-supplied hints and annotations has largely been unsuccessful in the general case, although certain important special cases (such as some parallel Prolog systems) have exhibited interesting potential for the future.

Explicit Parallelism

Explicit parallelism is undoubtedly the most common form of code written for parallel machines, and is also the easiest for the language implementor.

We may further sub-divide explicitly parallel systems into those in which parallel structures exist within the language and those in which parallelism is added to a purely sequential language.

In the first category fall languages such as Concurrent Pascal[26], Joyce[27], Concurrent C[51] *et cetera*. All these languages have (for example) a `cobegin...coend` construct, which allows statements to be specified for concurrent execution, coupled with the ability to perform synchronisation. There are, of course, variations in syntax and type security between these languages, but they are overwhelming similar semantically.

Occasionally one encounters explicit statements of parallelism within functional languages – a recognition of the difficulties encountered in extracting parallelism from even the most tractable frameworks. A good example is Concurrent ML[101], in which the `spawn` function (which is not referentially transparent) generates new threads of control.

Embedding parallelism into a purely sequential language is accomplished by creating libraries of functions which interact with the underlying operating system to create and control parallel activity. The best-known examples are the `fork` and `wait` system calls found within the Unix standard library[12], where each process executes in its own address space. Systems such as Mach also provide "thread" calls, where processes may share an address space[65]. Problems may occur with this approach, however, as the introduction of concurrency subtly alters the semantics of the host language, introducing the need for shared variables, protected regions *et alia*. This is especially true with languages allowing global variables.

Occam

The Occam language[86] is derived closely from Hoare's CSP[59]. It models concurrent systems as a collection of sequential processes which communicate *via* bi-directional channels. It has had a major effect on thinking regarding parallel languages, and deserves to be treated in detail.

Occam offers very fine-grained parallelism – processes may be created from arbitrary statements or blocks of statements, so it is possible to express problems in a maximally concurrent fashion. Processes do not share memory, at any level, so all communication proceeds by passing data values along channels – there is no pass-by-reference mechanism, and no pointers. The lack of shared memory in Occam means that it encourages a pipelined approach to parallel processing[64].

Occam does suffer from some fundamental disadvantages, however. The language has a very weak type system and almost no encapsulation facilities. The weakness in the type system means that there are type- and syntactically-correct programs which nevertheless cannot be compiled – or, worse, compile by generating incorrect code. The language forces an application to have a completely static structure in terms of the number of processes it contains and their interconnections. Current implementations are also very flimsy: the most common use of Occam is in programming Inmos Transputers, but the Transputer does not support some of the Occam constructs and makes the configuration of Occam programs very difficult. (For example, a single channel must be mapped onto a single hardware link, which is a significant constraint: it is necessary for programmers to perform their own link multiplexing in software.

From the point of view of scalability, the fixed number of processes and the fixed channel network means that an Occam application cannot re-configure itself to different hardware arrangements.

The Occam model of processing is also found in other systems, notably those such as Concurrent C which are really embeddings of Occam-style constructs into a sequential language. Occam's communications model is also found in some rudimentary parallel processing toolkits, as procedure calls for sending data down channels.

1.3.4. Configuration

Changing the processing elements in a system inevitably affects the optimal configuration of applications running upon that system. The search for an “optimal” configuration for particular programs is thus complicated by scalability..

Helios CDL

For the current purposes, the most important feature of Helios is its method of load balancing and configuration of multi-process programs. The system implements concurrency at the granularity of the Unix process (which is identified with a Transputer process), and allows processes to be composed using a Component Description Language (CDL). A CDL script defines the links between processes: when executed, a compiled CDL program is passed to the Task Force Manager on the node at which it is started, which attempts to map each process onto the system's nodes in a near-optimal manner using various heuristics.

The Helios shell has been modified so that, for example, a request to connect processes using “pipes” will result in a CDL object file to achieve the appropriate load balance.

Conic and Darwin

Both Conic and Darwin share the view of CDL, that processes are described using a sequential language augmented with the provision of ports which may be connected together. They differ from CDL in the level at which configuration is performed.

Darwin (the successor to Conic) is intended for use alongside the MP language[84]. A Darwin program defines a set of *components* which in turn define a collection of *ports*. The ports of each component may be linked together to construct an application.

Darwin allows its components to be either executable modules or collections of modules: a collection of modules may be made to behave as a single component for configuration purposes.

The current MP/Darwin combination is targeted at Transputer systems. A component may be assigned to a particular node; it is also possible to define the connection topologies of the nodes by treating each Transputer as a Darwin component having four channels (its links). This allows Darwin to configure both the multicomputer network and the application running upon it.

Performing Configuration Automatically

It is perfectly possible, in principle at least, for an application to be written without any information being supplied about its configuration. It is then the system's responsibility to determine the best locations for elements of the application, and to ensure that they can communicate.

Any system which completely removes notions of data and process locality (such as Linda) is effectively performing automatic configuration. The evidence suggests, however, that such systems are considerably less efficient than a skilled programmer.

A compromise is to allow the programmer to provide “hints” to the automatic algorithms. With care, the programmer may provide enough information for the system to determine a near-optimal configuration. Such an approach is ideal for a scalable system, since no hard-wiring of components occurs but the programmer still retains a residue of control over configuration.

1.3.5. Object-oriented Systems

Object-oriented systems are currently very popular in computer science, offering good encapsulation, typing and re-use. They offer a good abstraction for all the facets of programming systems considered above, and may be used to implement a distributed memory model, a concurrency model and a method of configuration, all rolled into one.

Fundamentals of Object-oriented Programming

An object is a named, persistent instance of an abstract data type[6] which may be manipulated only through an exported interface. The operations (called

methods[54] or *member functions*[108]) may manipulate the state of the object on which they are called, and may initiate method calls to other objects.

The basic intention in creating a distributed object-oriented system is to utilise the encapsulation properties of objects as a basis for controlling the location of data. The underlying system is constructed so as to make object names valid throughout the network, so that two objects may interact *via* method calls, no matter where they are located. An object-management system deals with marshalling and transmitting a method call to the appropriate processor and with returning any reply. Such systems are often called *virtual object spaces*[81].

Objects may also be used as an encapsulation mechanism for concurrency, introducing concurrency at either the object or the method level. Some systems provide special "active" objects which have a thread of control, acting like a process in a more traditional language. Others allow some (or all) methods to be executed without blocking the caller, or to unblock the caller before they complete. If only a single method can be executing within an object at any time, the objects present a *single-threaded* concurrency model; if many methods may be executing concurrently, then the object model is *multi-threaded* but requires some additional concurrency control mechanism.

Configuration is a matter of deciding which objects reside on which processors. This may be performed explicitly, by creating objects at a particular location, or implicitly by allowing load balancing or migration.

Some Sample Distributed Object-oriented Systems

At the risk of generalisation, it might be said that most systems concentrate either on managing concurrency or on managing distribution, but seldom both. We shall examine some of the more important distributed object-oriented systems below.

A Gossip of Smalltalks

Smalltalk[54] is the archetypal object-oriented programming environment. Everything in Smalltalk – integers, data items, data structures, class definitions, method definitions, blocks of code, files *et alia* – is an object with the same privileges and properties. This makes Smalltalk an exceptionally clean language.

A major criticism of Smalltalk in a concurrent environment is that processes are not objects, but are created from blocks using the `fork` method. This leads to programming occurring on two levels – objects and processes – which is a little at odds with the Smalltalk philosophy[72]. ConcurrentSmalltalk[122] is an attempt to solve this dual standard. It provides a small set of language extensions for creating asynchronous methods (which do not block the caller), `CBox` objects (which behave rather like futures[56][79]), acknowledgement replies (which reply to and unblock the caller without terminating the method) and atomic classes (on which all method calls are serialised).

A further extension to ConcurrentSmalltalk is provide by DistributedConcurrent-Smalltalk[93], which provides "secretary" objects to manage concurrency constraints more complex than the total seriality of atomic classes. Secretary objects allow methods to be related so that they will only execute in mutual exclusion, and allow

guards to be placed on methods which must be true for the method to proceed. The secretary is accessed through the meta-class hierarchy. The system also allows objects to be distributed. Objects are collected into name spaces, and users may select a hierarchy of name spaces (rather like the "path" variable[25] used to find executable files in Unix).

These Smalltalk variants all suffer from the disadvantage that they scrupulously maintain Smalltalk-80's semantics. Smalltalk was never designed as a parallel or distributed language, and certain aspects of its semantics and built-in classes (which must perforce be considered part of the language) are not amenable to extension into the parallel domain.

Emerald

The Emerald language[20][21][63] is an object-based language (to use Wegner's terminology[116]) which is chiefly concerned with the implementation of a virtual object space in which objects may migrate between nodes.

An Emerald object is created using a prototype, rather than a class. An object resides on a single node, but this node may change with time – a process called *migration*[67][68].

The run-time system tracks the patterns of activity occurring to every object in the system. If it notices that two objects are interacting heavily, then it will migrate one of them towards the site of the other (usually it is the receiving object which moves). This migration reduces the communications distance between the objects and hence improves performance. Various heuristic methods may be used to decide when to move an object.

Single method calls may also result in object migration. Usually values are passed either by reference or by value: Emerald supports a third parameter type, *by move*, which suggests that the object named be migrated to the site of the target of the method call.

It is possible to get into pathological situations with migration, such as when two widely-separated objects share the use of a third. The shared object may then be migrated from one side of the network to the other, causing a decrease in performance.

Object names must be maintained when objects are migrated, and Emerald performs this by using forwarding. An object name contains the node on which the object resides: if that object subsequently moves, the name will become invalid. Emerald stores a forwarding address at the node from which an object moves, so that any messages targeted to its old location may be forwarded. This may result in a cascade of forwarded messages if the object migrates often, which can degrade performance.

"Shared Objects"

The notion of "shared" objects comes from Tanenbaum *et alia*'s work on the Orca language[14][112], but is also found in other projects such as the work of Mallon and Dew[85].

Shared object systems emphasise the use of objects for communication, at the expense of their use as processes. In Orca, for example objects are completely

passive, and are complemented by active processes which perform computation. An object cannot have an independent thread of control: processes provide the active parts of the system, and may make method calls onto objects. All method calls are completely synchronous, and objects are single-threaded.

Mallon uses a similar system, where passive data objects reside in an address space common to all processors – *not* a DSVM despite the similarity in addressing. Processes may use the objects to communicate, and a migration mechanism similar to that in Emerald is used to reduce communication by migrating objects without changing their addresses.

One could argue, with a certain degree of truth, that shared objects are *not* object-oriented systems in the usual sense, as they separate processing from data encapsulation. Processes and process descriptions are not first-class, and require separate mechanisms for their implementation (processes are not first-class in Smalltalk, either, but process *descriptions* – blocks of code – are).

Actor Languages

Actors[1] are a model of computation based loosely on object-oriented systems – indeed, they have been described as a formal description of object-oriented programming – but which also bear a resemblance to functional and other state-free languages.

An actor is composed of a *mailbox* and a *behaviour*. Actors communicate with each other using asynchronous messages sent to mailboxes and queued (the model makes no guarantees of arrival times of messages, but does guarantee reliable delivery). Each mailbox has an associated behaviour, which may change with time.

A behaviour is a procedure which accepts a single message from the mailbox It processes this message and then dies: in dying, it nominates a *replacement* behaviour which is to process the next message in the mailbox. Behaviours must have a finite execution time, and no state information is maintained between behaviours.

A little refection will show that actor systems are *massively* parallel. Since behaviours must be finite, they cannot contain loops: all loops must be implemented by sets of actors communicating using messages. Since there is no state information, and no explicit ordering in the delivery of messages, it is possible for an actor to nominate its replacement *before* it processes its own message, thus overlaying the processing of the next message with its own actions. In general, a behaviour will give rise to a number of messages to other actors and the creation of new actors – all these actions are asynchronous and may occur in parallel.

This is both a strength and a weakness of the actor model. Parallelism is so widespread and so fine-grained that it is can be difficult to contain, and this often leads to actor systems being extremely slow on available processors.

Concurrent Aggregates

Concurrent Aggregates[36] (CA) is a language based loosely on Lisp, but which also bears a significant resemblance to the actor formalism.

An aggregate is a collection of objects which may be manipulated using a single name. The aggregate acts as a concurrent front-end to incoming messages, allowing

the aggregate to be internally concurrent. The aggregate delegates messages to the appropriate objects as they are received.

Messages (which are first-class in CA) are targeted at aggregates, rather than objects. The run-time system directs such messages to a particular member of the aggregate (a *representative*) selected non-deterministically. The representative may process the message itself or may delegate it to another object or aggregate.

The main claim for CA is that it allows hierarchies of abstractions to be created without introducing unnecessary synchronisation – a problem with some other object-oriented languages, in which the object is the unit of synchronisation.

Presto

The Presto tool kit[17], described as "a system for building custom concurrent programming environments," is a library of C++ classes targeted at shared-memory multiprocessors.

Presto's basic mechanisms are built around objects providing threads and synchronisation. The semantics of these classes are deliberately left quite weak: the intention is that they be used as base classes which are then specialised to provide application-specific objects. The justification for this approach is that Presto can be used to implement *any* concurrent processing paradigm, and so can be used to construct environments which are targeted closely at a particular domain.

The "openness" of the system means that adding new objects need not compromise efficiency. The scheduler, processor, thread and synchronisation objects may all be customised and may replace the default system objects (such as the system scheduler) dynamically.

Arjuna

Arjuna[43] is a tool kit for creating reliable applications in a distributed environment. However, a sizeable amount of work in the project concerned the management of concurrency[96], and reliability is a major concern for scalable systems.

The basic construct in Arjuna is the *transaction*: an atomic action performed on an object which either completes successfully or fails completely – there is no possibility of a partial failure and consequent inconsistencies. A successful transaction *commits* itself, whereupon the state of the object is written to stable storage[42]; a failed transaction causes the previous state of the object to be restored from storage. Transactions may be *nested* when one transaction starts another as part of its function, and Arjuna guarantees that such nested transactions also behave correctly.

1.4. A Scalable Programming System

Scalability introduces into applications the need to be able to cope gracefully with changes in the computing resources which underlie the application's definition. This implies that the actual available resources used by an application are only determined at run-time: to do otherwise, fixing the amount of parallelism and

distribution of data at compile-time, means that unnecessary constraints are placed on the ability of the application to cope with resource changes.

The "ideal" scalable parallel application will make optimal use of the resources which it finds available when it is executed, and will evolve its resource demands to fit changing system conditions. Depending on the nature of the overall computing environment, applications could optimise in such a way as to maximise their own performance, or to maximise the throughput of the system as a whole. Hence adding resources to the system will have a direct impact on the performance of all applications.

Given this, we may see the form of a programming environment for creating scalable applications. Such an environment has aims similar to those of the environments encountered traditionally in the construction of highly parallel applications, as surveyed in the preceding section. The needs of scalable systems are, in many ways, identical to those of other distributed-memory parallel systems, involving the regulation of concurrency and the distribution and control of data.

The main difference in a scalable system, however, is that there is no *a priori* information available to guide the programmer in distributing data or regulating concurrency: this information only becomes available at run-time. Therefore the identifying characteristic of a scalable application is that *it must defer until run-time all decisions relating to the exact distribution of data and its concurrent processing*. That is not to say that a programmer cannot indicate *which* parts of an application should execute in parallel, or how data should be distributed: merely that such descriptions are "slack."

The systems surveyed tend to take too rigid a view of their resource utilisation to be completely suitable for scalable programming. Linda, Strand and the DSVM systems offer an appropriate memory model, but at significant cost to performance – there is often a need to involve the programmer, however peripherally, in the assignment of elements to nodes. The model of parallelism presented by non-procedural languages is similarly suitable, but again the programmer must become involved if an application is to run efficiently. Both these factors reflect on the approach taken to configuration, which essentially concerns the way in which processors are assigned work in terms of storage and processing: none of the systems offers a suitably flexible model.

1.5. Résumé

Any new course of study must first review and analyse the work which has gone before. In this chapter, we have presented a review of the extant literature appertaining to the construction of highly parallel machines and applications, with a view to identifying important factors for scalable systems.

A definition and analysis of the term *scalability* was first presented. Scalability was seen to be a phenomenon emerging from the interconnection of groups of components in such a way that the power of the system thus created may grow in a useful way. A scalable computing system was thus seen to be a computer whose capabilities may grow incrementally and (effectively) without limit, to accommodate the needs of changing application and user communities. The essential feature of a

scalable system is its ability to address problems, regardless of their size, in an externally uniform way.

The creation of scalable machines was then discussed, from the point of view of their hardware and operating system support. The "best" architecture was argued to be an extensible, low-dimensional network of processor-memory pairs, coupled with an operating system which abstracts-away (at the application and user levels) from concerns about exactly what computing resources are available.

Systems for creating applications to make best use of scalable machines were then analysed. The consensus was that applications need to take a very abstract view of their computational requirements – the distribution of data onto processors and the regulation of the concurrency used to process that data – in order to be free to take advantage of scaling in the underlying machine. A form of "ideal" scalable application was identified, along with the form of a programming environment through which such applications may be created.

Chapter 2.

An Abstract Machine View of Scalable Parallel Programming

> When anything really new begins to germinate around us, we cannot distinguish it – for the very good reason that it could only be recognised in the light of what it is going to be. Yet if, when it has reached full growth, we look back to find its starting point, we only find that the starting point itself is now hidden from our view, destroyed or forgotten ... Beginnings have an irritating but essential fragiliy.
>
> Pierre Teilhard de Chardin, *The phenomenon of Man*

We shall begin our search for a scalable parallel programming system by developing an abstract model of scalable programming as a whole, to form the basis of a programming environment.

Firstly we shall describe the notion of an abstract machine, and draw a generalised picture of computer systems as layers of abstraction, with each layer being an abstract machine. We shall then focus on one particular aspect of abstraction – that of memory representation – and consider the ways in which the various layers of abstraction treat memory. From this we shall conclude that conventional data structuring may be seen as an abstract memory model.

We shall then introduce the concepts of distribution and parallelism, and argue that the data structuring notions are perfectly suited as a programming framework for scalable parallel distributed systems. We shall draw all these ideas together into a

uniform model for a scalable abstract machine, to be used as the basis for a programming environment for scalable systems.

2.1. Abstract Machines

Abstraction and abstract machine models are frequently seen as something of a panacea within computing: if the view taken of a problem is sufficiently abstract (so the argument runs) then its expression will be elegant and its solution simple. The reverse of this argument is that abstraction is an expensive luxury in terms of machine resources.

Viewed in its proper perspective abstraction is undoubtedly an extremely powerful tool, allowing the programmer to focus on the task in hand whilst avoiding uninteresting details. The use of abstraction often incurs a performance penalty, however, as the "uninteresting" details may in fact have a major effect on a program's run-time performance unless they are dealt with correctly.

There is thus a tension between the desire to abstract (and hence gain elegance) and the desire to achieve maximum performance (by tweaking low-level details). Nowhere is this tension more evident than in parallel programming, whose very *raison d'être* is to increase the speed of execution of programs made complex by parallelism and distribution.

Characteristics of an Abstract Machine

A *physical* machine is composed of three broad elements:

a) a set of *processing elements* to perform operations upon data;
b) a set of *memory elements* to store (and possibly manipulate) programs and data; and
c) an *interconnection network* connecting (a) and (b) in some manner.

The typical "Von Neumann" computer is an extremely pathological case: a single processor connected to a single memory *via* two buses[5]. A more modern case is the MIMD multicomputer, using processor-memory pairs connected by point-to-point links.

An *abstract* machine may be seen to have a similar structure:

a) a set of *processing structures* defining data transformations;
b) a set of *data constructs* to hold data; and
c) a notion of *communication* between structures.

[5]The existence of modern computers, following this pattern in the main but having dedicated peripheral buses, instruction and data caches and the like, does not invalidate the argument which follows.

Thus each physical structure has an abstract analogue, although the latter are likely to be more powerful than the former.

An abstract machine may hence be described as being an interesting computational model having capabilities not necessarily reflected directly in any particular physical realisation.

There are sequential and parallel abstract machine models, just as there are sequential and parallel computers. At the risk of generalisation, whilst current sequential models have sought to simplify tasks such as constructing type-correct programs, parallel models have been concerned with increasing efficiency and co-ordinating the actions of parallel threads of control[6].

Some Parallel Abstract Machine Models

The most famous model of parallel computation is undoubtedly Hoare's Communicating Sequential Processes (CSP)[59], which served as the basis for the Occam language. This views computation as being performed by a (large) number of simple sequential processes which communicate with each other using named uni-directional channels. Processes and channels form a fixed topology when the program is defined – it is not possible to reconfigure a set of processes while the system is running, since channel names are not first-class and cannot be passed between processes.

Functional programming has often been hailed as the perfect candidate for parallel computing, as functional languages may be seen to be inherently parallel. Church's λ-calculus[37] is the theoretical basis for functional languages, and may be viewed as an abstract computational model holding the same position as does CSP with respect to Occam. The chief distinguishing feature of λ-calculus is its lack of an explicit memory system. Programs are composed from collections of (side-effect free) functions which return a value based solely on their parameters, and the values within functions (and indeed the functions themselves) are identified by bindings of names to values rather than by named storage locations. The "memory" for such a system is provided implicitly by the nesting of function evaluations rather than by variables.

Closely related to functional programming is logic programming, where programs are expressed as logical predicates working on a database of known logical assertions. The most common example of this style is Prolog[39]. A program unifies assertions and sub-goals within the clause database, which may grow to be extremely large as the program progresses. The abstract model for this style of programming is the first-order predicate calculus.

A final category of abstract machine are the object-oriented and actor models. Both regard programs as being built from collections of objects, which are named entities encapsulating state information and a set of operations which may manipulate that state. Such systems are often seen as being a hybrid between message-passing and shared-memory models: although objects may be seen as the

[6]There is evidence that this difference is being eliminated – witness parts of the current work, and that of Bruce[28].

unit of (visible) memory within a system, all operations are invoked by sending messages to target objects. Parallelism in such models comes by allowing several objects to have active threads of control (sometimes abetted by allowing multiple threads concurrently within a single object). Actor models generalise even further by making all communication asynchronous and encouraging the construction of extremely parallel systems of communicating agents.

2.2. Programming Environments as Abstract Machines

Consider the case of the simple Von Neumann computer, such as a typical personal computer. The Von Neumann model specifies a single processor accessing a single block of memory composed of a number of named locations of equal size. The processor itself implements a small set of instructions which may be used to perform computation by accessing memory and control registers. Call this the level 0 or *physical* machine layer.

The level 0 machine is hardly ever encountered by programmers: it is a "bare" machine in every sense. The vast majority of programmers will work with some form of operating system, which will provide extra services over and above those provided by the bare machine. A typical example would be Unix, which provides notions of processes and file storage. Indeed, Unix' process abstraction seeks to provide the illusion of a number of independent processors dedicated to particular tasks, and implements this abstraction using a single real processor. Such an operating system also aids portability across hardware platforms. Call this the level 1 or *operating system* layer.

2.2.1. Programming Languages as Abstract Machines

Portability and re-usability are aided if programs are written in a high-level language rather than in the machine code of a particular platform. Some high-level languages also support generalised views of services such as filing systems so that the language view may be ported between several different operating system views – take as an example the `FILE` construct of Pascal. However a high-level language will also usually define its own data and processing models. This may be neatly summed-up as "algorithms + data structures = programs"[120] – the language provides a set of computational structures together with a set of data structures, and the two interact to perform processing. Neither element need correspond too closely with the related elements at levels 0 or 1, although a close correspondence may lead to superior performance.

This leads to the following question: if the programming language is seen as providing structures (both computational and storage) which are superior to those of the naked and operating system machines, is the programming language also an abstract machine model in its own right? The answer to this is a tentative "yes."

It is easy to see that a very high-level language – such as Prolog or Haskell – is in some sense "more abstract" than an operating system; it may be more difficult to

justify a systems language such as Ada or C as being so. However there are grounds for this supposition.

Pascal, for example, views memory in a manner completely different to that provided by the bare hardware – as variably-sized and strongly-typed. Similarly Pascal allows procedures to be written at a high level and to manipulate these large structures *en bloc* – a clear departure from the simple instructions and addressing of the processor. The "machine" being programmed is clearly different from that at level 0. What is more, Pascal's (and more especially C's) interface to operating system services is through a standardised procedural interface which may be ported between operating systems, so these languages are also different to the level 1 machines. It seems quite appropriate, therefore, to suggest that *any* programming language is *ipso facto* providing an abstract machine model: call this the level 2 or *language* layer.

Beyond Programming Languages

An abstract machine allows interesting computational structures to be easily expressed, without overburdening the user with too much detail. By their very nature, abstract machines are minimalist creations – they provide the "bare bones" functionality needed for their purposes. Good programming languages are no exception to this rule: they allow programmers to write a wide variety of applications without providing syntactic support for every possible contingency. On the other hand, there is sometimes a need for syntactic support to avoid an uncomfortable proliferation of unstructured procedure calls – try supporting exception handling without additional syntax!

Minimalism is undoubtedly a virtue, in that a simple system is far easier to learn and reason about than a larger one. However, in writing realistically complex applications, a simple language inevitably requires a great deal of functionality to be layered on top of the its basic structures in order to solve the problem. An ideal example of this problem is the Turing machine: powerful, but so minimal as to be unusable for any but the most trivial of tasks.

Our discussion so far has illustrated the important point that abstractions may be layered: each new layer of abstraction uses the layer(s) below it to present a new abstraction to the user. The advantage of layering is that layers may be done away with if necessary – the additional layers are optional, and the programmer may still use the lower layers – and different layers may be adopted as appropriate for different applications.

Taking Occam as an example: Occam allows users to write a wide variety of parallel applications, and does not mandate any particular coding style. This has not stopped the emergence of a number of programming "idioms", such as multiple-worker, process farm, pipelines *et cetera*, and several libraries and support environments exist to provide support for structuring applications around one of these idioms.

The layers added to languages are frequently termed *toolkits*, as they provide programming tools beyond those provided by the base language. A toolkit may be used, for example, to provide graphics capabilities within a language. It is usually

presented to the programmer as a collection of data structures and procedures, together with a run-time library to be linked into the finished program; it may also include auxiliary tools such as pre-processors, code generators *et alia*. By using the features of procedural and data abstraction provided in the host language, the toolkit can produce a more-or-less seamless abstraction over its chosen domain.

A well-thought-out toolkit shares many features in common with a programming language. It provides computation and data structures not found in the basic language, together with compositional mechanisms. It should be fairly minimalist, so that the programmer can easily assimilate its basic ideas and use them productively. In many ways, a toolkit should also be closely married to the language in which it is implemented, to reduce the intellectual effort needed to learn it. The creation of toolkits also places some demands upon the host language, in terms of its abstraction and encapsulation features.

Returning to the definition of an abstract machine, it should be clear that a toolkit is also an abstract machine: a level 3 *toolkit* abstract machine layered onto the level 2 programming language and making use of all the layers below. Toolkits may also communicate with other toolkits, or with applications running independently. Using the graphics toolkit as an example: it adds structures for the creation and display of graphic images which are not present in the base language. Many toolkits are extremely sophisticated, providing large amounts of extra functionality without in any way restricting the use of the host language. In a similar vein, many parts of the C standard library are level 3 entities: the `FILE` structure and its associated functions allow files to manipulated at a far higher level than is possible using the Unix system calls.

2.2.2. Toolkits in Scalable Programming

There are clear advantages to be had by taking the ideas of toolkit design and using them to construct a toolkit for parallel applications. The host language may be kept minimal whilst common idioms may be supported by the layered toolkit abstract machine.

The major complexity in parallel programming, over and above sequential programming, comes from the introduction of large amounts of concurrent activity. The task of the language designer, therefore, is to help programmers to master this additional complexity. As mentioned above, there are a number of programming "idioms" which have been developed to address this task. These idioms tend to focus on the large-scale structuring of processes – into pipelines, farms, collections of workers *et cetera* – allowing them to be viewed abstractly as a single entity.

In scalable systems, of course, the concerns of "normal" parallel processing are still present: in addition to them, there are issues of regulating concurrency in the face of uncertainty about the processing resources available, and of managing data location in the face of uncertain distribution. It is this uncertainty – not present in other fully-configured, ready-to-run parallel systems – which sits at the crux of the problem of managing and exploiting scalability.

2.3. Memory as an Abstract Structure

We shall now focus our attentions more closely on a single facet of scalable programming – the way in which memory is represented – and explore the ways of dealing with its challenges using layered abstraction. In doing so, we shall develop a framework for constructing scalable applications using memory as the central structuring theme.

Memory in General

The most pervasive programming tool is the chosen programming language. As a rule, imperative languages enforce a strict delineation between program and data; in other languages this separation blurs to the point of invisibility. In general a language defines a model of memory which it both presents to the programmer and (more or less) transparently maps onto the underlying storage architecture. Thus it is vital to realise that *memory* in a programming language sense differs markedly from *memory* as seen in hardware. This is a result of the view of programming languages as abstract machines, and it has important consequences.

Bruce[28] has argued that *all* elements of a program – its data, code, working storage, source code, execution information *et alia* – are conceptually stored in *some* memory unit. Different categories of program element are stored in different sorts of memory: procedures, for example, are accessed from an associative memory keyed on their name (at least at source code level); data is stored in variables as real numbers, records, objects, lists, enumerated types and the like, which are accessed associatively but manipulated using their own interface. All these categories are a far cry from the traditional (hardware-supplied) Von Neumann view of memory: elements are variably-sized, can be arbitrarily large, are scoped, and are accessed using different protocols. This last is the key feature.

The use of different access protocols to access different elements essentially means that memory is *typed*, since the memory defines both its contents' arrangements and the operations through which they may be accessed. From the programmer's viewpoint it is normally irrelevant that all these types are mapped onto the same physical architecture: it is the application-level abstraction which is important.

2.3.1. Object-oriented Memory

Of particular interest are object-oriented systems, since objects encapsulate both data and procedural interfaces. Distributed object systems are particularly fascinating, as they allow the features of object-oriented memory to be deployed against the problems of distributed parallel computing.

Systems in which objects form the sole unit of programming are sometimes referred to as *object spaces*. Objects form the unit of storage, with object names being equated to storage addresses. These names are then the unit of data naming, as communicated between objects. Objects may maintain a small amount of local state, consisting of name-value pairs: these may be communicated *by value* but not *by*

name, since such a pair is a binding, not an object, and hence has no name to communicate. An application's shared storage is then represented solely by objects.

An Overview of Object-oriented Memory

A basic object-oriented computer consists of some physical memory and processing elements: it may be a simple personal workstation, a shared-memory multiprocessor or a distributed multicomputer. The operating system on this machine implements a distributed object space in which all operations take the form of method calls to objects using a capability to name and access the object. There is no *semantic* notion of an object's location – all objects may freely interact through method calls, providing the caller holds a capability to identify the callee.

All elements of a program are notionally represented as objects. Hence all storage is represented by the creation and deletion of objects, whilst computation occurs through communication between objects using method calls. Possession of an object's capability may be equated with possession of the object, so that the acquisition of a capability effectively retrieves the object from object space

Variations

Consider the case where a fixed number of objects reside in the object space, none of them possessing a capability to any other object. Each object is then essentially independent of all the others. This situation cannot be called memory. Although the objects are retained within a single object space, and may be added and removed, they cannot communicate: moreover, they cannot retrieve data from the space as they do not have the necessary capabilities *and have no mechanism by which they can acquire them* since they can have no communication with any other object.

Now consider the case where objects do possess capabilities to other objects, either fixedly or *via* the creation of new objects. They can now communicate using method calls, as defined by their interfaces.

If we impose the restriction that capabilities cannot be communicated in method calls, we have essentially defined a special form of CSP – for *object* read *process* and for *capability* read *channel*. The objects in the system are in a fixed topology, defined by the possession of capabilities: since capabilities cannot be exchanged, the communication topology cannot change. Objects can only exchange data values. What we have effectively done is introduce the notion of data streams, which brings with it the idea of stream-parallel processing as found in Occam.

If we relax the restriction on communicating capabilities, we allow objects to pass data (represented in other objects) *by reference*. Doing so involves the caller passing the capability to the data object in a method call to the callee: the callee can then interact with the named shared object. A piece of storage known to the caller is made known to the callee *via* an explicit exchange of names, in the manner of Milner's π-calculus[115].

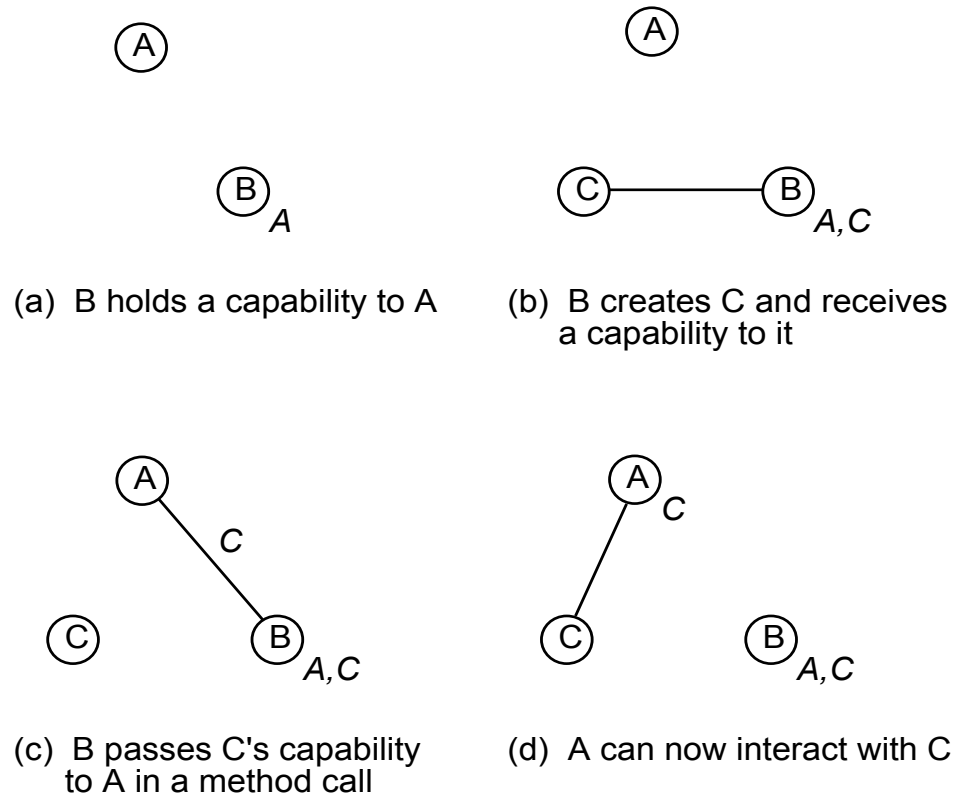


Figure 2: The workings of object-oriented memory

This scheme could be regarded as object-oriented memory: objects share data by exchanging the names of storage locations. This is the type of memory found in, for instance, the Sloop[81] and Orca[14] languages. Objects are created private, essentially as new pieces of storage known only to their creator: permission to share them must be given explicitly by passing the shared object's capability. Retrieval of information is represented by the passing of a capability, either as a parameter to a method call or as the result of one.

This may be shown more clearly in figure 2. Initially, object B may interact with object A by making method calls, but nor *vice versa* as A does not hold a capability to B. If B now creates an object C, B can interact with both C and A as it holds capabilities to both: A, however, can interact with neither B nor C. If, however, B passes to A the capability of C through a method call, then A can interact freely with C independently of B: in effect, the storage represented by C is shared between A and B.

Data Structuring and Memory

Although this scheme is workable, it seems a little too restrictive. There is a single level of naming – capabilities – with all exchange of data being explicit. This

means that if an object wishes to create and then share many objects it must explicitly pass the capabilities to the shared objects to *all* objects with whom it would share them.

In Pascal – to choose an example language – there is the notion of *structured data*, where a number of primitive data items are coalesced to form a larger item which may then be manipulated as a unit. Although Pascal is not object-oriented, this idea is an important one. In one sense, objects provide exactly this form of data aggregation, as an object collects together its local state under a single name; in another sense, there is still something missing.

Pascal allows arrays of data items to be built. These arrays may then be passed *en bloc* to a procedure. In our current object-oriented memory there is no equivalent of the array.

The common solution to this problem is to view an array as abstract data type and encapsulate its behaviour into a class, which may then be instantiated as required to build arrays of objects. An array object's internal state is then the elements composing the array, and as an object it may be communicated and shared by passing its capability. An array may be accessed using whatever (programmer-defined) interface is seen to be desirable.

There are two important ramifications of this approach. Firstly, the `array` keyword in Pascal acts as a (privileged) type constructor, generating array types from other, more basic types: in the current system, the array class is simply a generic or polymorphic class defined within the host language's framework, which thus has the same flexibility as any other class – specifically, it may be sub-classed to provide specialised interfaces and encapsulation. Secondly, the approach can be used to generate many different forms of data aggregate which may not be present in the host language – if the host were ML, for example, there would be no in-built notion of an array. Hence aggregation sits firmly at the application level.

The same argument holds for any language-defined structuring mechanism: pairs, lists, records *et cetera*. This is a great benefit of object-oriented programming, as it allows many features which were previously hard-wired into a language to be moved to the application level, with a corresponding increase in their flexibility.

This solution is completely workable as far as it goes, and raises an interesting point about the nature of memory. To recap: we have argued that in object-oriented systems memory is represented solely by the object population, and data sharing occurs by the exchange of capabilities to shared objects. Thus the only way to access memory is to be passed their capability explicitly. However, in creating an array object *another form of memory has been introduced*.

The sole purpose of the array is to act as a single name for an aggregate of other objects. The component objects may be accessed *via* method calls to the array, which will return (or assign to, or whatever) the various elements. We thus have a new mechanism by which objects can be addressed: they may be placed into an array and accessed by *element name* (presumably a small tuple of numbers). The object has acquired another name: it still has its capability, through which all interaction must eventually occur; but it also has another name reflecting its position in some programmer-defined space. Given that another object knows the name of the abstract space (the array) it may access the object using a meaningful protocol.

We have thus introduced a new notion of what it means to retrieve an item from memory. The array object acts purely as a "namer," in the sense that it maps meaningful names onto capabilities. Although acquisition of a capability still occurs *via* method calls, there is now a special form of object whose *sole task* is to supply these capabilities.

In order to share a new object, it has only to be placed into a data structure in order to be accessed by all other objects who share that structure. Thus the array is a form of shared memory: and since it is simply an object, it is a memory which is typed and instantiable.

The essential difference between this approach and the encapsulation of arrays *within* objects (rather than *as* objects) is one of information hiding. An object may use an array internally and not present an array-like interface; conversely an array interface need not actually be stored as an array. Using the explicit approach makes the storage architecture being used more explicit: by weakening the level of abstraction, we gain the ability to reason about the storage structure being used.

The abstract nature of such structures is obvious – they present very high-level characteristics to the programmer, being (in principle) of infinite size and having tailorable interfaces. On the other hand, they are evidently realisable as they are simply generalised versions of well-understood data structures, albeit in an uncommon guise.

We have introduced nothing essentially new in postulating collections of data as the aggregates of memory, but we have certainly made explicit some features of memory and data structuring which were hidden beforehand. In doing so we have altered the regular Von Neumann notion of what memory is, replacing it by a more flexible and larger-scale abstraction[7].

Typed memory allows the direct expression of patterns which may be hidden by a less flexible memory model. By regarding such structures as memory themselves, rather than as entities built on top of memory, it is possible to simplify a programmer's conceptual view of the machine being programmed. Since the memory interface is typed, it is possible to build *intelligent* memories tailored to specific applications.

2.3.2. Distributing Abstract Memory

The basic object-oriented computer described above was discussed without reference to its hardware architecture, and the view of memory just propounded similarly makes no reference to hardware issues. We shall now discuss a more restricted target machine – and object-oriented multicomputer – and consider the effects which distribution has on the memories discussed. Having illustrated the

[7]This whole argument is rather reminiscent of part of John Backus' Turing Award lecture from 1977[13], in which he argues that the "Von Neumann bottleneck" comes largely from the treatment of data as small units – a view prescribed more by the underlying architecture than by any high-level goal. In using large structures as memory, we are effectively advocating the processing of data in large blocks and the utilisation of certain application-level scoping constraints – the effects of which will become more apparent when we come to consider the effects of distribution and parallelism.

principles, we may now show how these ideas may be used to provide a programming environment for a highly parallel distributed system.

The main effect of distribution is to introduce data partitioning and parallelism. Since nodes do not share a common address space, an object in one address space is not directly visible to objects in other spaces. The virtual object space technique effectively implements a virtual shared memory, using complex network-valid object names and a parameter marshalling system in order to allow remote objects to communicate.

Having more that one processor allows one to obtain true parallelism, providing an application is written with this in mind. Distributing data onto a number of processors allows these processors to access the data in parallel, providing no global bottlenecks to access exist, and this in turn introduces the possibility for large performance gains.

The partitioning of a system's address space between component processors is probably the most noticeable effect of distribution. This disadvantage is an immediate departure from the more familiar flat shared memory model.

The advantage of our proposal comes from the fact that such a memory has no direct correspondence to any architectural feature in a system. There is no notion of object location and objects may be co-ordinated into meaningful structures of any size. In advocating the use of abstract typed memory modules (AMM's), we must address the effects which distribution has on these memories and *vice versa*.

The first case presents an obvious problem: we place no size constraints on our memories, so how can a memory larger than a single address space be implemented? Stated differently: if objects are the unit of memory, and abstract memories are simply special-purpose objects, is it not the case that the largest memory is limited to the size of the largest physical memory in the system in which a single object might be represented? The second case is a juxtaposition of the normal problems: can we use our new memory model to harness distribution in order to achieve a benefit?

2.3.3. Concurrency Regulation and Memory

Ideally it should be possible to solve both the distribution and concurrency regulation problems within the same framework, using the distribution management ideas to solve the concurrency problems.

Concurrency paradigms essentially fall into two categories: data-based and stream-based. In the former, a set of elements are accessed and processed in parallel; in the latter, a set of values is passed between processes. The difference is evident from figure 3: processes in a data-based system access a shared pool of elements, whilst processes in a stream-based system pass values between themselves.

Let us consider again the basic structure of our abstract memory modules. A collection of data, structured according to application- rather than machine-level concerns, is composed of a number of component objects which act as a single resource from the programmer's view. Each component holds a small amount of the collection's data locally, but has some mechanism for accessing transparently any elements which are held in other components of the collection.

The population of the AMM – in terms of elements and components – may vary with time. Presumably adding elements will produce more components: an AMM with more elements will have more components. Therefore the number of components is, to a large extent, a measure of the number of elements within the collection.

Furthermore, each component is a single object – the unit of memory – and so may be distributed. The components of an AMM may reside on different nodes in the multicomputer, co-ordinated through the virtual object space. Therefore a large collection, having more components, can potentially be more distributed than a smaller collection.

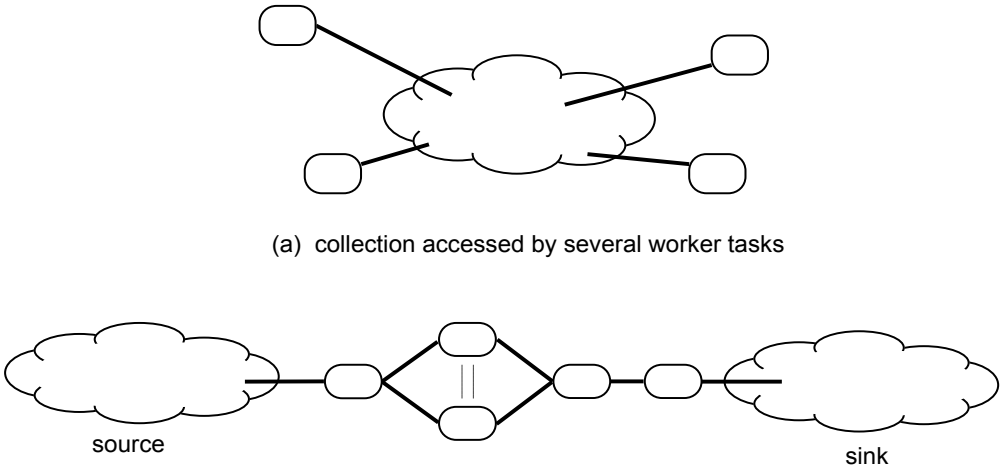
Let us now consider what is meant by processing. In general, a "process" accesses "memory" in order to obtain and store values upon which it works, transforming values according to its function. In our model, memory is represented by AMM's, so a process is simply an entity which accesses a data structure: a data- rather than a stream-based view of structuring concurrency.

For the data-based case, the concurrency regulation problem now reduces to how many processes should be deployed to access a memory, and where should they be located. Our model gives us a way to answer these questions. One may create one process per component of an AMM, and co-locate the process with the component which it is to access. The number of components is related to the size of the collection, so a larger collection will generate more concurrency; but equally a large collection is more widely distributed, occupying more processing nodes and providing scope for more true concurrency.

The stream-based case may use memory modules as sources and sinks for data (as in figure 3) but cannot use the size of the source as a guide to the number of stages in the pipeline: this is determined by functional decomposition, not by the amount of data to be processed. Although the model discussed above cannot directly aid this decomposition process, it can be used to construct such process structures: since processes are object, they may be placed into an AMM. It may also be used where replicated pipelines are applicable: each "process" accessing the shared data set may be a pipeline, with the number of replicas being governed by the size of the source collection.

2.4. The Scalable Abstract Machine

We have now arrived at a point from which we can see the shape of our scalable abstract machine. We shall attempt to draw together the threads sketched above – abstract machines and memory architectures – to create an abstract description of a programming environment for scalable computing.



(a) collection accessed by several worker tasks

(b) pipeline with replicated stage

Figure 3: Two paradigms for concurrent processing

Scalable Memory

The central features of a scalable system is its resource utilisation: scalability implies that processing, memory and communications resources may be added to a system as required.

Our abstract model is built on top of the object-oriented approach to computing. A system is composed of objects, each of which resides at exactly one node. Objects are the unit of memory, with all shared storage being represented as one or more objects. An object is a "black box" whose internal state may only be accessed through its exported procedural interface. Objects exist independently, and an object may only communicate with those objects whose name (capabilities) it knows. This scheme is essentially that found in the common object-oriented languages.

We now introduce the notion of AMM's, which are objects which represent what are normally described as data structures. A single AMM is composed of a number of objects acting so as to present the abstraction of a single large resource. The objects forming an AMM may reside on different nodes in the system. Each object holds a part of the AMM's elements, and can access any other element: it is irrelevant *where* an element of a collection is stored, as it may be accessed from any component object.

The scalability of this solution is obvious: there is no explicit statement of the number or distribution of the components which form an AMM, and so an application cannot be written to rely upon any particular arrangement. This leaves "the system" (or, more precisely, some underlying theorem of the abstract machine) to determine the exact configuration of an AMM – fixed or variable – according to run-time conditions. For any AMM there will exist some policy for deciding which elements are assigned to which component of the collection, and this mapping may change with time as conditions warrant.

Scalable Processing

Concurrency in the scalable abstract machine comes in two flavours. The first comes from the basic definition of the object model: any object may give rise to a thread of control, and there may be several threads running concurrently within a single object. This makes *task parallelism* – where threads perform logically separate tasks – easy to express. Moreover, since activities are simply objects, they may be placed into memory: this implies that the structuring features provided by scalable memory may be used to build and interact with "active" objects.

The second form of concurrency comes from the attachment of activity objects to components of an AMM. Since the AMM is distributed, there is scope for true parallelism when processing it's elements using a multiple worker, data-based and -regulated style. Each activity is responsible for processing the elements held by the component to which it is attached. This relationship is shown in figure 4.

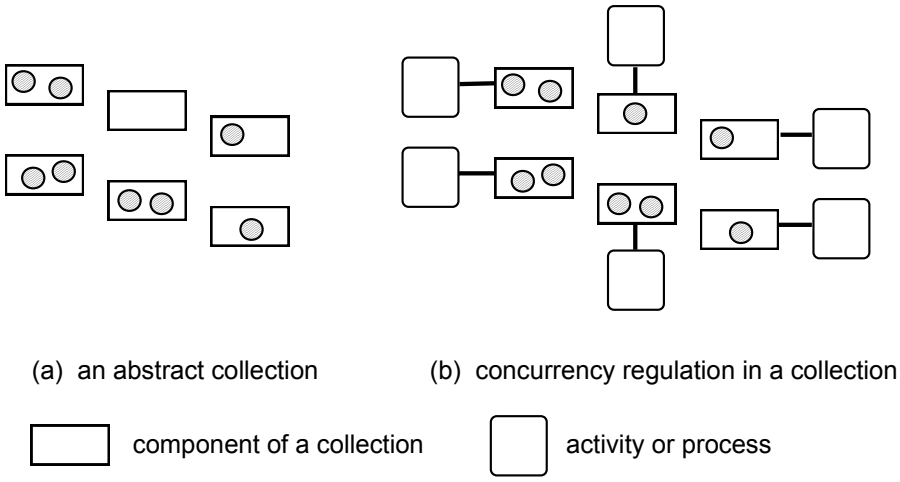


Figure 4: Collections and concurrency in the scalable abstract machine

Therefore the policy by which AMM's are created – especially the policy by which the number, size and location of components is decided – is used as a metric for concurrency regulation. There is no notion of exactly how many components will exist, or where, and hence no notion of exactly how many processes will be deployed to process a particular AMM. From the point of view of scalability, this architecture has the desirable feature that applications must be written so as to be able to use an unknown number of activities in the processing of a memory. The processes themselves may be simple objects or more complex structures of processes, themselves constructed using the AMM tools.

Programming Practice

An abstract machine inevitably has an impact on programming practice, and it is natural to wonder how applications written under the scalable abstract machine would be written.

The basic architecture of such an application is as a set of AMM's to which are attached activities running in parallel. The distribution and internal structure of each AMM is managed by the underlying compiler and/or run-time system, and it is this distribution which is used to regulate and locate concurrent activities. The model might be described as one of *indeterminate parallelism*: an application determines *when* parallel activities are to be created, and *which* AMM's they are to access, but has no control over exactly *how many* activities are created. This indeterminism allows the system to scale itself according to the available resources.

In order to see the form of an application built around scalable memory[8], consider the case of a logic programming interpreter using a dictionary-associative AMM. The clause database itself may be created and clauses added, with the exact distribution of clauses and the database being hidden:

```
type database = ... ;;
val newDatabase : unit -> database ;;
val assert : database -> string -> unit ;;
val retract : database -> string -> unit ;;

let cd = newDatabase () ;;
assert(cd, "man(simon)") ;;
...
retract(cd, "lives(simon, York)") ;;
...
```

The distribution of the database may change as clauses are added, and might be affected by external control factors such as suggestions about the decomposition used. Each component of the database is a `database` object in its own right, related to the other components of the AMM in a manner not apparent to clients. The population of components may be fixed, or may vary with time.

A query of the database consists of generating a set of unifications of variables against clauses in the database.

```
type 'a binding = {   var : string ;
                      val : 'a } ;;
```

Queries may occur in parallel by creating a “query processor” activity and attaching it to the database. Each query processor performs the actions involved in the query on a part of the database to which it is attached, with the results of the complete query being the amalgamation of the results of all the query processors.

[8]In this example we shall use a pseudo-code notation modelled on ML. More concrete examples are deferred until we introduce the Phœnix prototype, chapter 5 *et sequitur*.

Functions are needed to attach activities, synchronise on them and retrieve their results:

```
type 'a activity = Activity of
                     (database -> 'a) ;;
val attach : database -> activity ->
                  activity list ;;
val waitFor : activity list -> unit ;;
val resultsOf : activity list -> 'a list ;;
```

The query may run in parallel, with the creating thread either continuing execution or blocking until the results of the query are obtained

```
val unify : string -> database ->
                binding list list ;;

let s = "man(?x)" in
   let q = Activity ((fn c cd = unify cd c) s) in
      let al = attach cd q in
         waitFor al ;    (* block *)
         flatten (resultsOf al) ;;
```

resulting in a list of all the possible sets of bindings which satsfy the query.

Notice that nowhere is there any reference to the number of activities created (although it might be obtained by the length of the list returned by `attach`): the application has no need of this information. Similarly any activity (or any other piece of code) may access any element of the database: the application has no need to be aware of the element's location within the distributed structure.

We shall return to this example in §6.4.3.

2.5. Résumé

We began this chapter by considering the nature of abstract machine models as applied to programming systems. We developed a view of a programming environment as a layered abstract machine, with each layer building upon the layer below. This approach led to the contention that both programming languages and toolkits implemented within them constitute abstract machines, as they provide important, well-defined computational and storage structures which are not encountered in lower layers.

We considered one important feature of languages and toolkits: the way in which they allow data to be stored and manipulated. We argued that any system which provides data aggregates – data structures such as arrays, lists and the like – is essentially providing a memory model, as these structures alter the way in which programs treat their data. The analogy was drawn between accesses to memory and accesses to data structures, with the result that a data structure was seen to be an abstract typed memory whose elements could be accessed using meaningful names.

We introduced the notion of distribution, and considered the effects which a data-structure-oriented view of memory would have on scalable programming for multicomputers. We showed that the use of abstract memory, if it can be implemented, hides certain important facets of the underlying machine – notably its local memory sizes and parallelism. It is possible to create applications which can utilise the scalable platform – rather than be hindered by it – by writing programs around the framework of scalable memory modules. This allows the application to defer until run-time – when the memory modules are actually instantiated – those features of its execution which are affected by scalability: its distribution and use of parallelism being the two main variables.

Chapter 3.

Implementing Scalable Typed Memory

> Knowledge is simply a kind of fuel; it needs the motor of understanding to convert it into power.
>
> John Wyndham, *The Midwich Cuckoos*

Having derived an abstract model of scalable memory, it is now necessary to consider the ways in which it might be implemented. In this chapter we shall consider some methods for implementing scalable storage architectures in an object-oriented fashion, while deferring the exact details of such an implementation until a later chapter.

We shall first determine the requirements of any implementation, derived from the abstract model, and discuss some possible implementations. We shall then present the *partitioned object model* as a particularly suitable architecture for representing memory in scalable systems. Using this model, we shall present some storage architectures representing a kernel of basic memory types. The architectures derived are the most general for the structures being considered, and alternatives are described which may be useful in particular cases. From these architectures, the process of deriving user-level data structures will be illustrated.

3.1. Requirements

The previous chapter presented a form of scalable abstract memory modules composed of object communities, with each member of the AMM residing in a single address space. The AMM implements the abstraction of a single, scalable resource: the membership of the community, its size and distribution may change

across its lifetime, but each component may at any time be used to access any element of the AMM to which it belongs. The task of implementing such a model of memory may therefore be summarised as follows: an implementation must provide

- a *single-object abstraction* so that collections appear to clients as a single entity whilst being implemented as a community;
- *transparent distribution of elements*, so elements may be accessed from any member of the collection;
- *unbounded size* to allow collections up to the size of the globally-available memory to be represented;
- *variable size*, so a collection only occupies the storage necessary at any time;
- *concurrent access* avoiding bottlenecks which would mitigate against highly parallel access;
- *strong typing* to avoid the abuse of data and structures; and finally
- *extensibility*, to allow application-specific intelligent memories and distributions to be constructed.

The implementation of transparently-distributed, unbounded and variably-sized strongly-typed structures is the subject of this chapter; concurrency will be deferred until chapter 4, whilst extensibility will be addressed in chapter 5.

3.2. Partitioning: Representing Scalable Memories

Implementing AMM's requires the development of a technique for co-ordinating all components of the AMM into behaving as a single resource. The technique has been termed *partitioning*, as it involves the (largely) automatic partitioning of an AMM's elements amongst a number of nodes.

3.2.1. Overview of Partitioning

Scalable memory is represented by *partitioned collections*, which presents AMM's as high-level, strongly-typed data structures. These structures may be treated as single resources, regardless of any internal distribution. They are implemented as communities of objects which co-operate to support the single-resource abstraction.

Given that distribution of elements in a partitioned collection is essentially invisible to clients, control of data manipulation and data distribution within the collection may be separated. This separation allows these tasks to be specialised independently of one another, with a corresponding (beneficial) effect on re-use. The partitioned object model provides this separation in the form of two parallel class hierarchies: one provides an access protocol through which client tasks access the collection; the other controls the manner in which elements are distributed and new storage created. The interface between these two hierarchies is well-defined, allowing easy modification of either party.

The details of the strategy for implementing collections in this way will now be discussed, together with the trade-offs and important parameters which affect the method's efficiency.

3.2.2. Managing Data Access

A data access protocol must allow clients to access a collection's elements in the appropriate manner – using a key for associative memories, for example. The exact details of the protocol are obviously specific to the particular class of collection being implemented – this is especially true of the user-level protocol – but a few features are common across categories.

The partitioned model calls classes providing data manipulation *collection* or *component* classes. A component class has three duties within the model: it must

- provide local storage for some elements of the partitioned collection to which it belongs;
- provide an interface through which elements of the collection may be accessed; and
- implement functions to perform data manipulation on any elements which it holds locally.

The management of local storage means that it is the components which control all local memory accesses and allocations, hiding "real" memory from client tasks. The provision of an interface means that the components define exactly what operations may be performed upon data within the collection. The interface will (for the abstract base classes) be quite minimal, but it may be specialised in sub-classes to provide arbitrary functionality. The manipulations need only be defined on locally-held elements, however, as remote elements will require the intervention of the partitioned collection's distribution management objects: this implies that there exists some way of determining whether a particular request may be serviced locally.

3.2.3. Managing Distribution

Since partitioned collections are composed of object communities, some way must be found to co-ordinate components of the collection into behaving as a single resource. This is the function of distribution management objects, which the partitioned model terms *partition* objects.

Like the component classes, partition classes serve a well-defined function. In fact they have better external independence, as they do not have to deal with user-defined access protocols. A partition serves three functions: it

- defines what elements of the collection are held by which component;
- creates and destroys components as required; and
- resolves requests for data onto the component which holds that data locally.

Although components create and manage local storage, the contents of that storage, relative to the complete collection, is defined by the partition objects. Components may have their storage responsibilities re-defined as new components are created or old components destroyed. The extent to which a partition is involved in data manipulation is restricted to its ability to locate components which hold data: *what* is performed to that data is irrelevant; the partition is only concerned with *where* the action takes place.

Component and partition classes interact, then, through a very narrow interface, limited to the following:

- the ability of partitions to assign and re-assign local storage to components; and
- the ability of components to forward requests referring to remote data *via* the partition to the appropriate component.

This narrowness means that it is relatively straightforward to use a novel distribution manager with an existing data class. However, the exact interface used varies depending upon the type of collection which is being distributed, and this also affects the way in which elements are retrieved from the components of the collection.

3.2.4. Resolution

Resolution is the name given to the process of locating an arbitrary element of a partitioned structure from a particular component. Since, as mentioned above, all components of the structure allow all elements to be accessed – they all act as first-class *pseudonyms* for the entire collection – resolution occurs whenever a component receives a request which it cannot service locally.

The basic process is as follows. A component (the *receiving* component) receives a request from a *client* object for some data. If this data is held locally, then the request is serviced locally; otherwise a request for resolution is made to the receiving component's partition. The request for resolution will contain the name of the element(s) required and the service to be performed, and the partition maps the element name onto the component which holds the requested element (the *servicing* component). It then forwards the request which was made of the receiving component *in toto* to the servicing component, which will perform the request and return the result to the client.

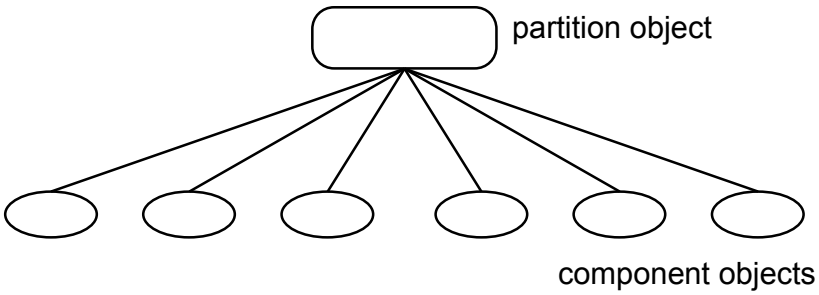


Figure 5: Flat distribution management

The method by which resolution is performed is defined internally by the partition classes. It may involve intermediate steps through other partitions as the request percolates through the partition tree.

3.2.5. Parameters Affecting the Distribution Architecture

The partitioned model effectively hides from clients the exact inner workings of scalable memory, and implementors of new (or derivative) memory structures and distributions may use the architecture provided for their new, application-specific tasks. However, there are several important trade-offs and parameters which must be understood if the partitioned model is to function effectively:

- the arrangement of the partition tree;
- the sizes of components;
- the degree of automation in the management of a collection's distribution; and
- the manner in which applications are configured for execution.

It is important to realise that the parameters to be discussed affect the *efficiency*, not the *semantics*, of applications. An application is shielded from the details of distribution: although some distributions are more efficient than others in particular cases, *all* distributions are equal in terms of correctly accessing elements of a memory.

The Partition Tree

Several trade-offs occur in the organisation of the partition tree. Its major parameters are its degree – how many child components and partitions a particular partition has – and depth.

In a flat distribution, all components of a collection are managed by a single partition. This is shown in figure 5.

This architecture has the advantage of simplicity. Knowledge of the structure resides in the partition object. Resolution of any element from any component can

occur in a single step – from the receiving component, through the partition, to the servicing component.

However there are disadvantages, as might be expected from so simple a solution. The prime problem is exactly the centralisation mentioned above: since all resolution requests pass through a single partition, that object is a hot-spot in computational and communication terms. Distribution of components may also be a problem, depending upon the load balancing scheme used: in some schemes objects may tend to cluster around the partition which created them. If this problem is overcome, and components are more widely distributed, then the communications costs implied by making resolution requests will increase due to the distance. The single partition is also a single point of failure – a point to which we shall return in §3.5.

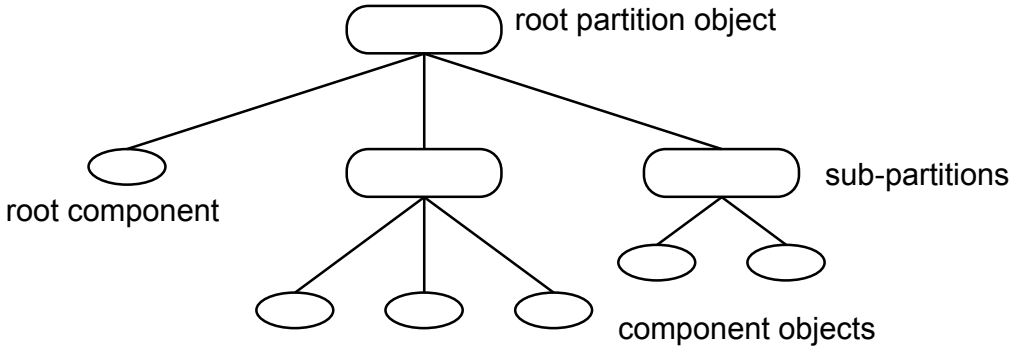


Figure 6: Hierarchical distribution management

In a hierarchical distribution, management of components is performed using a tree of partitions, as shown in figure 6. The major difference apparent between this figure and figure 5 is the existence of partition objects as intermediate nodes of the tree – the tree has a depth greater than one.

Since no single partition object has a complete view of the structure, resolution is complicated. Several steps through several intermediate partition objects may be needed to resolve a request, and the complexity of a request varies depending upon the relative positions of the requesting and servicing components.

However, this very complexity allows us to introduce desirable features. Although resolution is now a distributed algorithm, several concurrent resolution requests may progress without interference and without generation of hot-spots: the computation and communication are distributed throughout the structure. Furthermore the architecture lends itself to better control of distribution of components, since the partitions (which perform the creation of new components) are themselves distributed. The single point of failure in the flat partitioning scheme has also been removed, which may allow better tolerance of faults in the network.

Routing through the partition tree has logarithmic complexity. On average, one might expect 50% of all requests for data to be resolved through the root of the tree: however, one goal of the partitioned model is to exploit locality of reference, which should significantly reduce the possibility of such pathological circumstances arising.

Component Sizes

The unit of distribution for a partitioned collection is the component. The size of components – and therefore the number of components, and hence their distribution – is the major factor controlling the manner in which partitioned collections are represented in memory.

The size of components is not, *a priori*, important to a collection, in that exactly which component holds a particular element is semantically irrelevant. However, there are a number of lower-level pragmatic issues which *do* constrain the size of a component.

The first constraint is local memory size: a component cannot be larger than the physical memory in which it resides. This places a hard upper-bound on *component* size: in a non-partitioned object-oriented system, this limit fixes the maximum size of a *collection*.

It may not be desirable, however, to make use of all the available physical memory: there may be advantages to making components smaller than they actually have to be. Using the entire memory of a node means that no other objects may reside on that node. If the component contains (the names of) other objects these objects must be placed on different nodes, which introduces a communications overhead. Equally importantly, the component's partition object may be farther away than is desirable, and this means that extra communications delays may be incurred at every resolution step. Indeed, it may be advantageous to have more than a single component per node, for reasons discussed in the next chapter.

Load Balancing

Load balancing is a well-studied field (a good overview may be found in [92]), and the current work does not attempt to expand upon it. What *is* assumed is the existence of a load balancing component within the underlying operating system which chooses the site for a new object at its creation.

Load balancing has several aims, amongst which are to achieve:

- low overhead, to avoid the load balancing process impacting on performance;
- low remote communications, so that the gains in improved processing speeds are not offset by increased remote communications overheads;
- high processor utilisation, to avoid wasting available processing capacity; and
- even load distribution, so that all processors have roughly the same amount of "work" to perform.

We may examine the partitioned model to determine how it interacts with these aims. The first, third and fourth goals are essentially operating system concerns of general impact to any programming system; the second, however, is a major concern. Object-oriented systems by their very nature are communications-intensive, so it is vital that objects are not placed too far away from the other objects with which they interact.

It may be noted that there is a location structure implicit within the partitioning process: components are assumed to reside close to their partitions, and sub-partitions to their parents. Notionally there is a "one hop" communication between these pairs of objects, and it is desirable to make this logical distance the physical distance also.

If this assumption is not respected – if, for example, a component is placed a long way from its partition – then a resolution request may involve far more communications that is apparent from the logical structure of the collection. In the worst case, such a request might be sent great distance only to be resolved back to a node close to that which originated the request!

The partitioned model will function best in situations in which the logical distance between two objects is related closely to their communications distance in the network. A good example of this is the "ink blot" style of load balancing, where objects are load-balanced onto nodes neighbouring that of their creator. A collection occupies a number of neighbouring nodes (initially a single node); as it grows, new components and partitions load balance onto nodes along the periphery of this neighbourhood. A larger collection thus occupies a larger neighbourhood, with components which are farthest from the root logically being distributed furthest physically.

A topic closely related to load balancing is object migration (or adaptive load balancing), which can result in substantial gains in performance over statically load-balanced systems[68]. It would be difficult to respect the assumptions given above in a system providing general object migration; moreover, resolution may exacerbate the pathological cases possible under migration (§1.3.5).

The other aims of load balancing may be aided by the partitioned model's fin-grained decomposition of large structures. Since objects are the unit of distribution (and hence of load balancing), ensuring high utilisation and even load is much simplified A consequence of this decomposition is that the load balancing system must have an extremely low overhead, since more entities will need load balancing than in coarser-grained systems.

Automatic versus *Manual Distribution*

It must be recognised, however, that in many cases a knowledgeable programmer may achieve better load balancing *for his application* than can an automatic system. Thus there is often a need – or at least a desire – to circumvent load balancing and place elements of an application manually. Scalability makes this aim almost an impossible one, however.

On the one hand, a programmer would like to be able to specify the distribution of program items according to knowledge which are not readily available from the program's code (often termed *meta-knowledge*, although Wilson[119] prefers the term *clichés*) in order to improve its performance; on the other, a highly parallel application, unless it has a very regular structure, may prove beyond a programmer's capacity to manage effectively, and even regular applications may change their computational requirements dynamically. The partitioned model takes the view that the actual distribution management policy should be encapsulated into a class, which may then be changed and customised if required. This has several advantages, the

primary one being that the method used for distribution is completely separated from the methods used to access data.

In the general case, it is desirable to allow placement to occur through the load balancer, since this module may make use of information to optimise the load across the entire system rather than across a single application.

Configuration in the Partitioned Model

It should be obvious by now that determining the optimal configuration for a partitioned collection involves balancing several, possibly contradictory factors, and this may only be performed at run-time. This implies that factors such as component size, tree depth and other parameters which have no real semantic importance should be set only at run-time: they should not be hard-wired into the compiler or into an application. It should be possible for the programmer (or indeed the end-user) to "play" with the values to achieve best performance. This has the additional advantage that tools might be employed to modify the exact configurations of partitioned-model applications.

Fortunately an obvious method exists for supplying these values: the use of a mechanism such as Unix' shell variables, which may be set per-user at run-time and accessed by running applications. Even more flexible are the property sheets encountered in systems such as X Windows[103] which provide a more structured name space.

The exact properties used may vary per structure, and indeed may be varied according to what distribution policy – *e.g.* which partition class – a collection is using. At the risk of getting ahead of ourselves, the arrayed collections of the Phœnix prototype (as described in chapter 5) allow properties such as the maximum and minimum number of elements in a component, the minimum dimension of a component, the depth and composition of the partition tree *et cetera* to be set at run-time from the property sheet. There is no reason why these values should be provided directly by the user: they may be generated by some form of automatic configuration tool whose results are stored in a suitable format. This means that the partitioned model has potentially the same levels of efficiency as other, less flexible and lower-level systems.

3.2.6. Generic Structure of Partitioned Collections

For clarity, we shall summarise the structure and features of the partitioned collection architecture. The generic form of collections is shown in figure 7[9].

[9]The symbols for component, partition, activity *et cetera* in the key will be used consistently for the rest of this thesis, and will not appear in future diagrams.

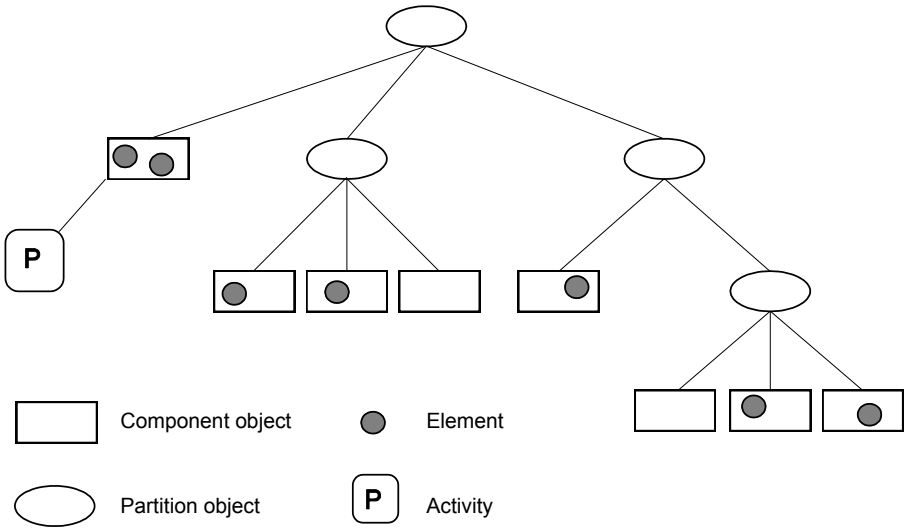


Figure 7: A generic partitioned collection

The community of objects form a tree structure, the branches being formed by partition objects and the leaves by component objects. Note that, in this diagram, there is no indication of the location of objects: the objects may be created and located by the system, not necessarily by the programmer.

The process (or *activity*) creating the collection (marked **P** in the diagram) holds a handle to one component (called its *root* component), through which it may access all elements of the collection, held in any component, without exact knowledge of the component with which it interacts. Other activities may be passed either the handle of the root component, or some other component, and may similarly access any element without knowledge of distribution. The resolution process ensures that such accesses occur correctly.

There are two general cases for resolution. In the first (shown occurring from activity **P** in figure 8), the item being requested is held locally by the receiving component and may be returned directly. In the second case (initiated by activity **Q**) the data is held remotely and must be resolved by one of more resolution steps.

All algorithms used within the partitioned collection must be distributed, using only local knowledge. This prevents bottlenecks occurring in the face of highly parallel access: only those activities which access the same part of the partition tree will interfere with each other. Similarly, concurrency control should – as far as possible – occur on a per-component basis.

The generic architecture, and the properties required of the algorithms used, is presented formally in Appendix A.

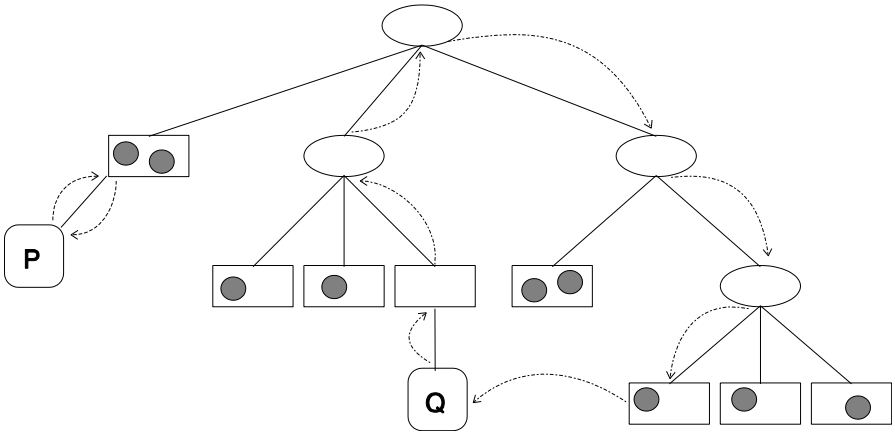


Figure 8: The general case of resolution

3.3. A Kernel of Partitioned Storage Architectures

No programming environment can hope – or, indeed, should hope – to provide all the computational objects which might possibly be required in applications. The essence of programming language design is to select a small number of common, powerful structures together with mechanisms for composing them into new structures.

In proposing a programming environment based around the partitioned model, this principle manifests itself in the following question: what memory structures should be provided "as standard" within the environment, and how should programmers be able to extend these structures?

A survey of the literature – especially Knuth's seminal work[70] – indicates that three basic storage architectures predominate:

- *arrayed* storage, where elements are stored at points in a discrete high-dimensional space;
- *associative* storage, where elements are accessed using complex keys or partial matching; and
- *directed* storage, where elements are stored relative to one another.

These three architectures may be used to form the kernel of a partitioned object programming environment. Other architectures are also possible, as described in §3.3.4.

As an aside, it should be mentioned that in this section we shall only deal with the architecture of memory, not the programmer's interface. It is perfectly possible, for example, to implement an array using an associative memory architecture rather

than an arrayed architecture. From the programmer's viewpoint, it is the *interface*, not the *implementation*, which determines the properties of a structure; internally, the high-level programming interface is largely irrelevant for the efficient implementation of the memory. We defer discussion of the programmer's interface and the ways in which it may be developed until §3.4.

For each architecture, we shall first describe the properties of the architecture and a basic approach to its implementation. A prime factor in assessing any such approach is the manner in which *locality of reference* may be exploited within the structure to reduce communications. We shall then discuss the manner in which the elements of the architecture may be decomposed for partitioning purposes, and finally give an overview of the structure and algorithms used in its implementation.

3.3.1. Arrayed Storage

Arrays are one of the most common structures in parallel computing, although this is largely attributable to the current engineering bias in parallel applications. Consequently it is important to provide a usable arrayed storage architecture.

An array is a dense collection of elements, each being uniquely identified by an indexing *n*-tuple (usually of integers or some other discrete type). The degree of the tuples also defines the number of dimensions in the array, and reflects the fact that an array may be viewed as a discrete *n*-space: in fact, representing areas of space (such as wind tunnels) is a common application of arrays.

The space represented by an array is a *metric* space, as there is a relationship between the various possible index tuples. This relationship defines a distance between two tuples, and allows one to speak of two elements of an array as being "close together" or "far apart." This may be best seen by considering an array with a small number of dimensions, say two: such an array defines a part of a plane, and the distance function may then be seen as the usual straight-line distance between two points on the plane (or may be taken to be the Manhattan distance). This notion of distance is shown in figure 9.

Basic Approach

The basic approach to decomposing arrays is to observe that any single array may be seen as a collection of smaller arrays whose origins are displaced. By suitable computation it is possible to map any element in the original array onto an element of *one* of the smaller component arrays.

This observation allows us to suggest how arrays should be distributed within the partitioned model: first determine the origins and bounds of the small arrays and then distribute them, performing the appropriate mapping whenever an element is accessed.

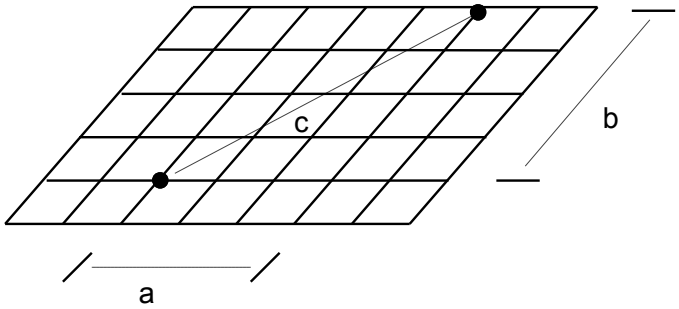
It is a matter of policy, determined mostly by application-specific considerations, how the origins and extents of the small arrays are determined. Many policies might be used, some if which are explored further below. There are two important aspects to any chosen policy:

- how does the policy chosen interact with the principle of locality? and
- can the policy deal with arrays of arbitrary size and dimensions in a scalable manner?

A third, lower-level concern must also be addressed:

- how does the policy lend itself to distribution on a large scale?

Since we are considering scalable structures, this is a very real concern: one may reasonably expect a scalable system to make *good* use of its available resources.



Straight-line distance = c = 5 units

Manhattan distance = a + b = 7 units

Figure 9: Arrays as metric spaces

Resolution must occur through the indexing tuple of the element being sought: there must exist some means of mapping an arbitrary tuple onto the component which stores it. Different decomposition strategies will require different resolution strategies, having different timing properties.

Locality of Reference

The principle of locality, as it applies to arrayed memories, may be stated as follows: given that a task accesses a point *(x, y)* at time *t*, it is likely that the same task will access a point $(x + \delta x, y + \delta y)$ at some time $(t + \delta t)$. In other words: if a point is accessed, other points which are close to it in space will be likely to be accessed in the near future. Exploitation of the principle requires that accesses following this pattern are made as cheap as possible, on the assumption that they will predominate over accesses which do not conform.

Within arrays it seems to be relatively easy to define "close together:" we may use the metric space view of arrays described above. By storing elements in clusters, or *locales*, the principle of locality may be exploited: if any point in a locale is accessed, most of the other points close to it are available directly from the same

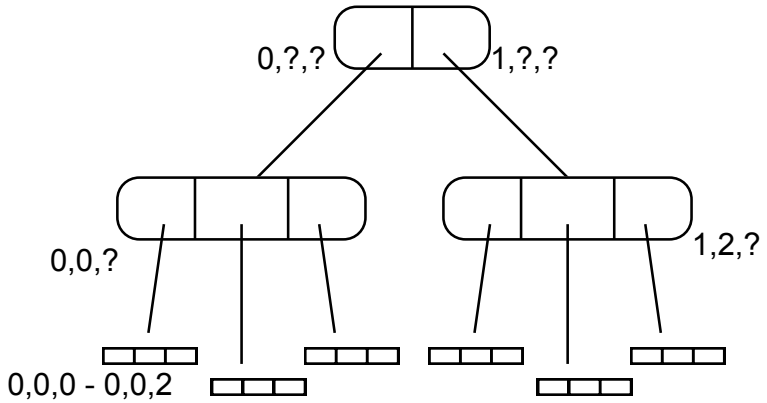
locale. Although edge effects will occur – when neighbouring points lie in different locales – these effects may be minimised by careful choice of locale size. This optimal size is very much an application-dependent quantity, so it is vital that it be easily alterable to achieve best results.

Decomposition Strategies

The key factor affecting the representation of an array is the way in which its elements are decomposed into sub-regions, which will then be used to form components. The strategy chosen will be used internally by the partition class in order to create and organise the components: the code of the components themselves is independent of the strategy chosen.

Dimensional Decomposition

Dimensional decomposition divides an array according to its dimensions, acting so as to reduce the dimensionality of locales. An *n*-dimensional array may be seen as a one-dimensional array of *(n-1)*-dimensional arrays, which may in turn be seen as a one-dimensional array of one-dimensional arrays of *(n-2)*-dimensional arrays, and so forth. Alternatively, the same array might be seen as a two-dimensional array of *(n-2)*-dimensional arrays: there are many possibilities.



a = array (0..2, 0..2, 0..1) of T
each component = array (0..2) of T

Figure 10: Distribution and resolution in dimensional decomposition

The great advantage of dimensional decomposition is that it essentially reduces all arrays to one of a number of simple forms having a low dimensionality. This means that the base-level storage management and access routines may be written for simple cases, as more complex cases are automatically built up from these simpler units.

Furthermore, a structure thus decomposed has a simple resolution strategy associated with it. The first level of storage is simply a set of pointers to other arrays, and form the first level of distribution. This is shown, for example, in figure 10, where a two-dimensional array is represented in exactly this way. Resolution can occur simply by stripping-off the first element of the addressing tuple and descending to the corresponding sub-array, until an array containing data elements (rather than further pointers) is encountered. For a d-dimensional array, such an approach has a complexity $o\left(\frac{d}{d_{step}}\right)$, where d_{step} represents the number of dimensions stripped at each step until a component is reached.

There are three disadvantages, however. First and foremost, reducing an array in this manner alters its essential spatial properties: two points which are close together in the original array may not be close together in the decomposed structure if the decomposition takes place across the dimension in which they are close. This makes the principle of locality break down in the general case, as a locale of points may be separated.

The second problem is one of the size of the intermediate storage. It may be that an intermediate array is generated which is too large for a single node memory – or, at least, is too large for comfort. There is no recourse within the strategy to deal with this case.

The final disadvantage is that this strategy does not make good use of the available resources. The intermediate arrays are simply pointers to "real" storage. All the storage will occur on the fringes of the structure's distribution, with the effect that remote requests will be very expensive.

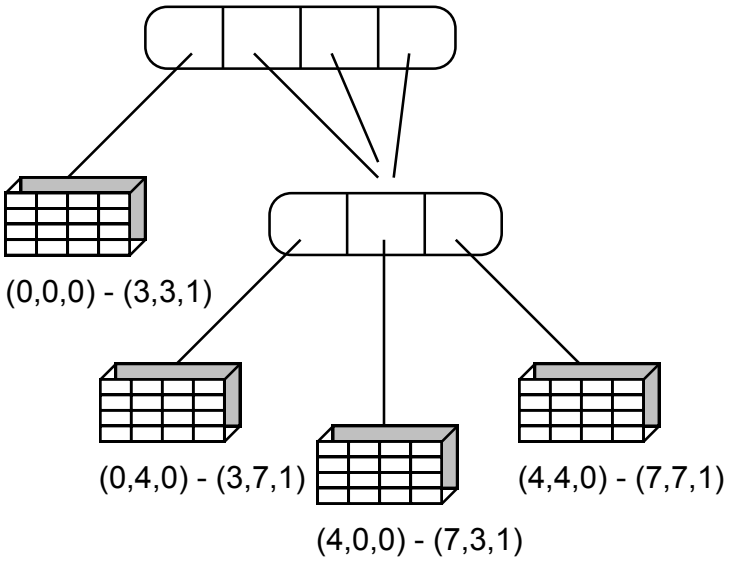
Regional Decomposition

To counter these disadvantages, we may take the view that an array should be decomposed *regionally*: that is to say, each component should hold a small part of the array having the same dimensionality as the original. This preserves locality of reference within the metric space of the original array.

Two points may be made about this approach. Firstly the dimensionality of the original array is preserved by the decomposition, so any locality properties are also preserved. Secondly, the distribution is "flat" rather than progressive as in the dimensional decomposition. This has the advantage that all the resulting component are available from the first and may be distributed evenly throughout the resources used by the structure.

The disadvantage is that, as the decomposed components of the structure may have arbitrary dimensionality, the component classes must be able to store data in this form.

Resolution takes the form of locality tests performed in each partition. Every partition must contain references to all components held below it – with the result that the root partition holds a reference to *all* the components of the structure, which may make it very large. From each stage a single resolution step is performed according to which region the requested point lies. The step will result in either a sub-partition or the holding component. This resolution strategy is evidently *o(n)* for an array with *n* elements.



a = array (0..7, 0..7, 0..1) of T

each component = array (0..3, 0..3, 0..1) of T

Figure 11: Distribution and resolution in regional decomposition

Hierarchical Decomposition

Hierarchical decomposition is the most "geometric" of the resolution strategies, taking its cue from the idea of oct-trees[47]. The space of the array is divided into a number of smaller arrays – usually eight in a three-space, hence the name oct-tree – which are then recursively sub-divided until a suitably-sized region is created.

Such a strategy has much to recommend it: in particular, the access complexity of a structure distributed in this way in $o(\log n)$, which is perfect for a scalable system. A disadvantage, however, is that correctly-sized regions only appear at the leaves of the distribution process. If implemented naïvely this would mean that data is only stored at the fringes of the data structure, which would be rather unacceptable from the point of view of load balancing. This problem may be solved by using a slightly more complex creation algorithm, re-broadcasting some of the final regions back up the tree.

N-fold Decomposition

One further important decomposition method should be mentioned: dividing the array into a particular number of sub-arrays. In fact, this method may be successfully applied to both hierarchical and regional decompositions, rather than being a completely new distribution method, but is particularly suited to regional decomposition when the programmer wishes to control accurately the distribution of data onto processors.

Consider the case of an application wishing to create an array, where the programmer knows that the applications exhibits a certain pattern of access which makes it especially suited to being distributed onto a rectangular grid of processors. The programmer may wish to control very precisely the way in which the elements are mapped onto the processor mesh. A partition may be created which divides the array into a number of smaller regions – where the number is the same as the number of processors onto which the array is to be mapped – and map them onto the processors.

It may be difficult to divide an arbitrary array into exactly a given number of sub-regions, to the number required may be taken as a hint, a lower bound or some other suggestion rather than as a "hard" number.

Structure

The creation of an arrayed storage module involves three stages: the decomposition of the array space into manageable units, the distribution of these regions, and the creation of storage for their elements.

Decomposition may proceed according to any of the strategies mentioned above. For convenience we shall use regional decomposition.

The root component creates a partition of the appropriate type and passes it the region representing the entire array's space. The partition decomposes this into smaller regions. One of these regions will be assigned back to the root component; others will have components created for them; others will be passed to additional partitions which are created for them. The exact numbers used to determine how many components and how many sub-partitions are created may be varied to alter the exact structure of the collection. The process of creating components and sub-partitions proceeds until a stage is reached when all the sub-regions have been assigned to components. At this point the collection is ready for use.

(Usually, arrays are created "eagerly:" storage is allocated for them at their creation. However, this may be very expensive in the case of a large array, and especially so if the array is destined to be sparse. Rather than perform eager creation, the partitioned model allows storage to be created "lazily," as it is demanded. The advantage is that any region which is never accessed will never consume storage; the disadvantage is that accesses to element may cause storage allocation, which will introduce an overhead.)

The basic action of the array components in use is to accept requests from client objects for an element and return this point's value (or assign it, or some other operation). Since each component knows its own bounds – the elements which it is responsible for holding – it may quickly determine whether an element requested is held locally and, of so, may perform the requested operation. If the element is not local, the request is forwarded to the partition for resolution. The partition will accept the request, resolve it onto the component holding the requested point, and forward the request to that component. The component can then return the result (if any) to the originating client. Hence a component only performs significant processing on requests which it can service: its involvement in remote-request processing is limited to a locality test and a forwarding operation.

Within the partition tree, resolution proceeds as follows. A partition examines the regions which it knows about: depending on the decomposition strategy used, a different matching mechanism must be used. For regional decomposition, the regions will map either to components or to sub-partitions. In the former case, the resolution process has succeeded, and it may be passed the processing request; in the latter, a request for further resolution is passed to the selected sub-partition. If a suitably-matching region cannot be found, a request for further resolution is passed to the parent of the partition, on the assumption that eventually a partition will be encountered (the root in the worst case) which will be able to perform a resolution downwards.

3.3.2. Associative Storage

Associative memories, though less common in practice than arrays, are in many respects more powerful. Such a memory holds a collection of objects having no explicit relationships between them. An object is simply held within the store, where it may be accessed or removed: there is no notion that any object is related to any other in any way. This may be contrasted with the metric space conception of arrays.

Unlike arrays, associative memories have no fixed size: elements may be added and removed as required.

The principle use of associative memories is in storing objects which must be accessed through partial matching. An application may construct a template of the object which it wishes to retrieve and present this to the memory, which will then match the template against its elements to find one or more matching objects.

Basic Approach

The basic approach to constructing an associative memory is through the use of a *hashing* algorithm.

Hashing is a technique whereby an indexing *key* is used to access a large set of records. The key need not be unique, so a given key value will be shared by many of the structure's records. The keys are used as indexes into a table, each entry of which contains those records sharing the given key. Thus a simple look-up operation may be used to reduce quickly and dramatically the number of records which need to be searched.

"Standard" hashing algorithms require a set of keys $C = \{c_1, c_2, \ldots, c_n\}$ which are mapped onto a set $S = \{s_1, s_2, \ldots, s_m\}$ of *buckets* holding records. The records in each bucket each share a common key, and a *hash function* $h: C \rightarrow S$ maps each key onto a single bucket. In general, $m << n$, with S and h being fixed before the algorithm begins executing.

This means that a hashing algorithm can, given a key for a record, locate the bucket containing the record in *o(1)* time by applying the hashing function. The records in the bucket thus selected may then be searched using linear search, binary split *et cetera*, depending on the internal structure of a bucket, so that the exact record being sought is located. The complexity of these intra-bucket searches may be *o(b)*, $o(\log b)$, or some other function of bucket size b: if this intra-bucket

complexity is represented by the function *j(b)*, then the overall search complexity of the hashing is *o(1+j)*. For best results, the hashing system should attempt to make the *j* component negligible so that the system tends towards *o(1)* searching.

Since buckets are usually of a finite size, there may come a time when a bucket overflows. In this case some overflow method is required to re-hash the record which caused the overflow onto a new bucket, together with a corresponding modification of the search algorithm to search for overflowed records if necessary. In order to avoid these complications, buckets are sometimes implemented as linked lists or some other dynamic structure.

The main problems with hashing, for the current purposes, is that it is non-scalable. The set *S* is fixed at the creation of *h*, so it is not possible to vary the number of buckets to accommodate dynamic changes. The use of unbounded buckets is unacceptable: for one thing, it can require a large amount of local storage and may thus cause distribution problems; for another, the time taken to search such a bucket degenerates towards the complexity of the intra-bucket search, as the *j* term become dominant.

What is required is a structured, scalable hashing system which retains the desirable $\approx o(1)$ search complexity. Fortunately a number of methods have been proposed for *extensible* hashing system[45][73][80], and these have been examined for suitability in the current context. The "ideal" scalable hashing algorithm would have the following characteristics:

- unbounded size;
- a simple distribution strategy
- regularity, in the sense of using the same algorithm under all circumstances;
- decentralised control; and
- scalability, so the distribution of a structure reflects its size.

It was found that no completely suitable algorithm was described in the literature, but the existing systems were found to be suitable as a rubric for creating a distributed extensible hashing algorithm.

A Guide to Extensible Hashings

All the extensible hashing considered here were developed in the 1970's to manage secondary storage in large database systems. A particularly pertinent comment is made by Fagin *et alia* in discussing their hashing:

> "Over the past two decades, schemes for structuring large files have evolved by merging concepts and techniques from two areas that were initially perceived as requiring distinct approaches: data structures appropriate for central memory, and access methods appropriate for slow, high-capacity secondary storage. The distinction is becoming more and more blurred[45]."

We have examined three of the most promising extensible hashing methods: the *virtual*, *dynamic* and *extendible* hashings.

Virtual Hashing

Virtual hashing[80] works by altering the hash function whenever an overflow occurs. The definition given for virtual hashing – as any hashing which may change its hash function over its lifetime – is very vague, but a concrete example may be found by considering hash-by-division.

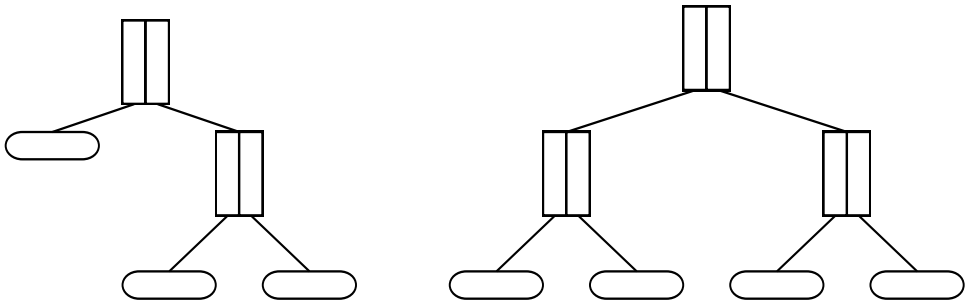
The hash function in this case may be defined as $h_0(c) = c \bmod |S|$, which defines a "classical" hashing with a fixed number of buckets each of which is identified by the remainder of dividing the key c by $|S|$. When a bucket overflows (assuming buckets of a fixed size, rather than chained buckets) the collision is usually resolved by calculating a new key value *for that record* and re-inserting it into another bucket. In virtual hashing, a new hash function is generated which re-maps *all* the records in the full bucket, leaving them in the existing bucket or mapping them into newly-created buckets (rather than onto existing buckets as was the case before). A definition of the virtual hashing is given by $h_j(c) = c \bmod 2^j |S|$ where h_j is the function applied to resolve the j'th collision. Initially j will be zero. When a bucket is full and is split, any hash keys which, when transformed under h_0 target that bucket will be re-hashed using $h_{\#1}$. If further buckets become full they will be re-hashed using h_1, and so forth.

This particular hashing doubles the number of (potential) buckets at each split, and has the additional property that the "new" buckets are always empty (which avoids a possible cascade of splits). Its disadvantages are that there are constraints on the forms of h_{i+1} compared to h_i which must be guaranteed, and a new function (satisfying these constraints) must be supplied at each level. This leads to a lot of functions.

Dynamic Hashing

Dynamic hashing[73] uses a hash function h to identify a tree which is used to contain records. Instead of a single structure, a collection is represented by a forest of binary trees $T = \{t_1, t_2, \ldots, t_q\}$ with the hash function being defined as $h: C \rightarrow T$. Once the correct tree has been identified, it is searched. Navigation of the tree uses a secondary function $b: C \rightarrow \{(0,1)^+\}$ from this key to an infinite series of bits. This sequence is used to traverse the tree by using each bit to indicate a left/right branch. Traversal continues until a leaf node is encountered, which contains records.

If a bucket overflows, it is split and re-arranged according to the values of b. This is illustrated in figure 12.



(a) tree from the initial forest (b) splitting a bucket

Figure 12: Dynamic hashing: splitting a bucket

Dynamic hashing may be especially efficient if indexing data is held locally The secondary hash function *b* may seem complicated, but may be implemented very simply using a pseudo-random number generator seeded from the key. The initial hashing – from key to tree – suffers from the usual disadvantage of hashings that the size of the forest is fixed at creation-time, so essentially the scheme is a mechanism for structuring buckets internally in an efficient manner. The algorithm is distributed in the sense that intermediate nodes are used for navigation.

Extendible Hashing

The most promising candidate is extendible hashing[45], which builds on Fredkin's work on *tries*[50]. In trie[10] memory, a search tree is constructed from a set of records whose keys are sentences in some alphabet Σ. For some (possibly complete) prefix of a key, a tree is constructed: from the root node, a branch exists for every $\sigma \in \Sigma$, and so on recursively until the key has been completely parsed. The result is a tree having a path corresponding to every possible key prefix within the alphabet.

For each leaf node a bucket is constructed to contain records whose keys have the leaf's identifying path as prefix. Such a system is naturally scalable: if a bucket overflows, the leaf node is split into a new branch node which consumes the next symbol of the key (extending the prefix by a single letter of Σ), with new buckets being created at the end of each new edge and the values within the original bucket being re-distributed according to the new trie.

Special cases are the binary trie using the alphabet $\Sigma = \{(0,1)^+\}$and the *n*-dimensional binary trie using the alphabet $\Sigma = \{((0,1)^n)^+\}$.

A trie may easily be seen to be wasteful of memory, as a path is created for every possible prefix of a given length. To improve this, branches may be created "lazily" when a symbol is encountered. Levels may also be compressed so that several symbols are consumed at each branch in the manner of the *n*-dimensional binary trie. It is this latter observation which is used to create extendible hashing.

[10]The name rhymes with *try*.

Several alternative extendible hashing schemes may be considered in which an additional level of dictionary look-up is provided between the hash function and the buckets – the hash function maps onto a set of dictionaries, which themselves map onto a set of buckets (if $D = \{d_1, d_2, \ldots, d_r\}$ is the set of dictionaries, then h is defined by $h: C \rightarrow D \rightarrow S$). A specific extendible hashing may be defined using a *pseudo-key* $c' = h(r)$ where c' is large and of fixed length (say sixty-four bits). A certain prefix of c' (of, say, three bits) is used in the first hash routine, with each combination mapping onto a bucket. Thus all records whose pseudo-keys have the same prefix will hash into the same bucket. The buckets themselves may have an internal structure as required.

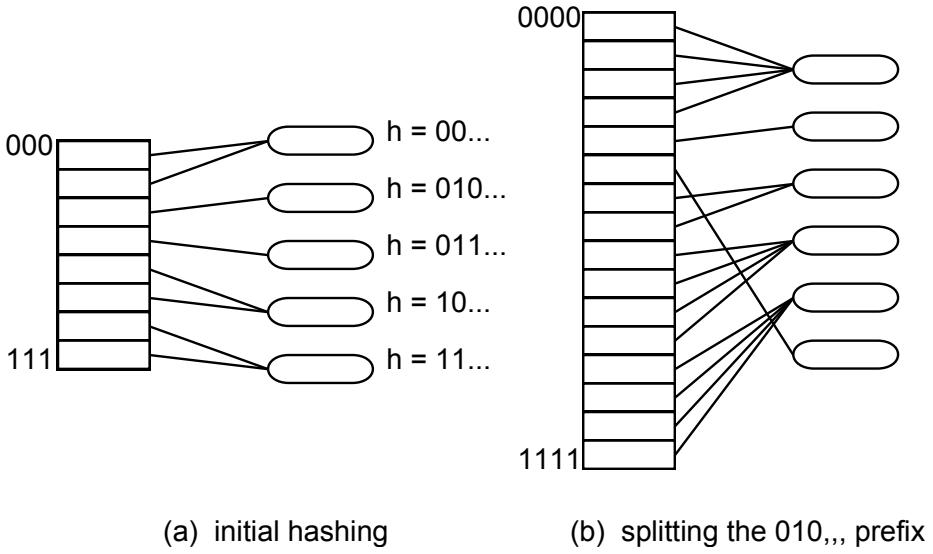


Figure 13: Extendible hashing: splitting a bucket

When a bucket overflows, another bit is added to the pseudo-key in the primary table – splitting a prefix causes *all* prefixes to be split. Most of the new prefixes will be redundant, mapping onto "overloaded" buckets: the bucket which caused the overflow, however, will be re-arranged so that a new bucket is created. This is shown in figure 13.

Extendible hashing hence requires that the splitting of a bucket is recorded in the master hash table: it is not a distributed algorithm, in the sense that all information is concentrated within a single table. However, it guarantees that the bucket containing a record may be acquired using a single probe, as with standard hashing.

Distributed Extensible Hashing

None of the schemes discussed was designed with distribution in mind, so they have an essentially centralised nature. It is common practice, for example, to assume that the entire index is held centrally – not a viable proposition in the current context,

where it would constitute a bottleneck, a limit on the maximum size of the structure and a single point of failure all rolled into one.

The table in figure 14 summarises the properties of the methods reviewed. All the systems satisfy the constraint that the structure be able to vary its size in response to changing membership. Regularity is a feature of extendible hashing, so the code used to implement the structure need not change for varying depth. Distribution control is aided by the use of intermediate points, such as those of dynamic hashing, rather than the use of a logically centralised table.

Thus, while none of the systems exhibit all the desired characteristics, together they satisfy the needs of a scalable parallel implementation. It would be attractive, therefore, to synthesise the required algorithm from a combination of the three systems.

	"Standard" hashing	Virtual hashing	Dynamic hashing	Extendible hashing
number of buckets	fixed	variable	variable	variable
keys	fixed	fixed	fixed	fixed (prefix)
secondary keys	none	none	infinite binary sequence	none
additional functions	none	one per level	tree traversal	none
bucket access method	single-step	recursive hashing	hash, then tree search	single-step
extension	none	new hash function	extend tree	alter master table
number of hash calculations	one	one per level	one	one

Figure 14: A comparison of extensible hashing schemes

The dynamic hashing of Larson uses index tables at intermediate branch nodes, which is reminiscent of the generic partition tree in figure 7**Error! Bookmark not defined.**; Fagin's extendible hashing uses a variation on the trie concept without such intermediate look-up. These two schemes may be amalgamated to form a new algorithm in which intermediate nodes are based around tries. This means that the same search algorithm may be used throughout the structure. The new scheme is completely regular, scalable and distributed.

A basic table is first built using a prefix of the key value *c*. Call this prefix *c′*, of (say) three bits in length. This table will initially point to buckets – eight for a three-bit prefix. When a bucket overflows, it is replaced by an intermediate node which

uses a prefix of the remaining key (say c'') in the same manner as the initial table. The contents of the split bucket are re-distributed between the new buckets (and the original bucket may be re-used as a bucket at the deeper level). Thus each lookup descends one level of the tree by stripping a prefix from the key being sought and forwarding the rest to the object identified by the stripped prefix. Eventually a leaf node – bucket – will be encountered, which will hold all records containing the given key. This structure is shown in figure 15.

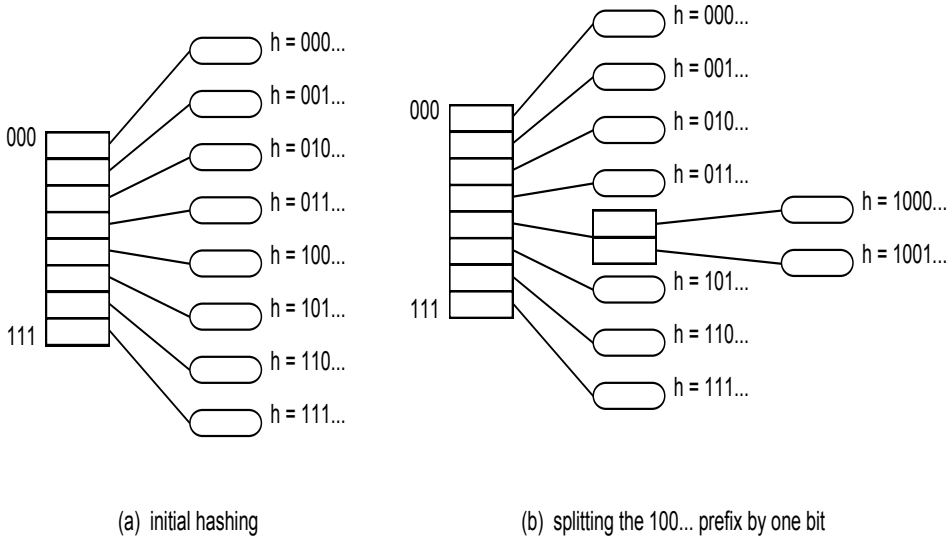


Figure 15: Distributed extensible hashing: splitting a bucket

The splitting of a bucket generates an intermediate node, rather than adjusting a master table. Indeed, there exists no master table: the information needed to maintain the structure is distributed between the branch nodes of the tree. Each split has only a local effect, so no information need be propagated to the rest of the structure. Each intermediate node uses a prefix of the key to cascade the search, and may do so using purely local information. The fan-out from a branch node may be arbitrarily large.

A similar approach may be taken if the membership of the structure should shrink. If, for example, all the child buckets of an intermediate node become (nearly) empty, they may be *joined* to form a single bucket and the unnecessary intermediate node removed: an exact reversal of the procedure used to split a bucket.

In the current context, the most important feature of the algorithm is its regularity. The same function is used at each branch – master or intermediate – as was the case in the basic trie. In addition, however, hashing may proceed from *any* intermediate node to the correct bucket, by the following argument: if an intermediate node knows its own prefix (the prefix which is common to all its descendents, then it can determine for a given key whether the bucket for values with that key occurs below it in the hashing (*i.e.* if the key is prefixed by the node's prefix) or whether it must lie above it, down some other branch (if the key has a different

prefix). In the limiting case, a key will pass through the master table, which has no prefix.

However, access to a record may require many sub-hashes, rather than the guaranteed one of the original extendible hashing. This is less of a problem that it might appear since, in the current system, all items will be in memory (albeit on different nodes) so it is more important to emphasise distributed control and scalability. There is a one-to-one correspondence between intermediate nodes and partitions.

Locality of Reference

The bucket in which an element is stored within a hashing algorithm is defined by its hash key (or a prefix thereof): it is not implicitly defined by semantic considerations, as was the case in arrayed storage, but by a hidden (and rather complex) value generator – the hash function. We must therefore consider the effect which extensible hashing has on applications wishing to exploit locality of reference.

There are two alternative approaches to this question. The first would allow an application to define, at some high level, that objects inserted into associative storage should be stored together; the second forces applications to work within the locality framework imposed by the memory architecture itself.

In the first case, an application might define that (for example) all Linda tuples with a particular type signature will be hashed to the same (or very similar) values and will hence be stored in the same or closely-neighbouring buckets. An application wishing to process all tuples of this type would then be able to assert that they are stored together, and would be able to locate its processing activity so as to minimise the access overheads (by accessing the correct bucket(s) directly). The disadvantage here is that the amount of parallelism may be reduced, as only a small number of buckets will be used in processing.

Using the same example, the second case would distribute tuples according to some hidden mechanism. Processing all tuples with a given type signature would involve potentially accessing all the buckets in the structure, but each activity need only access the tuples held locally by the bucket to which it is assigned: in other words, each activity would iterate through a single bucket of tuples, filtering-out those in which it was interested. This scheme is maximally parallel (within the partitioned object framework), but may result is activity to no purpose if there are buckets with no suitable tuples within them.

We have here another example of a trade-off, between increasing parallelism and increasing the programmer's knowledge of and control over the associative structure. The partitioned model allows applications to be constructed at either extreme, and at any point in-between.

Structure

We shall briefly make clear the mapping between distributed extensible hashing and the partitioned model.

Referring back to the generic partition diagram (figure 7**Error! Bookmark not defined.**), the correspondence is as follows: each component represents a bucket in the hashing algorithm, whilst every partition represents an intermediate trie node. A bucket holds records having a certain key prefix: the partitions holds mapping tables which can strip a prefix from a key and either return the bucket matching the prefix or forward the remainder of the key to the appropriate sub-partition. Partitions must also know the prefix which identifies them from above.

Hash keys are composed of long unsigned numbers. In principle, of course, a limit on scalability is imposed by the length chosen for hash keys; in practice a number of (for example) 64 or 128 bits will be sufficient for all but the most demanding applications. It is possible, in any case, to generate infinite keys by using a smaller number as a seed to a pseudo-random number generator.

On receiving a request for a particular key, a bucket compares it against its own local prefix: if they match, then the record sought may be acquired locally. (Of course there may not *be* such a record – the bucket may be empty, or later matching may fail. This is unlike the arrayed case, where an element will always be present. An associative store must provide a failure case.)

If the prefix does not match, it may be forwarded to the receiver's partition for resolution. The partition performs an action depending upon the key value.

If the key has a prefix which matches that of the receiving partition, then the sought-for record must lie below the partition in the tree. It may therefore strip the next part of the key, match it within its table of descendents, and forward it appropriately.

If the key has a prefix different to that of the receiving partition, or is shorter than it, then the sought-for record must lie either in another branch or above the current partition: in either case, it must be resolved up the tree by passing it to the partition's parent.

Adding an element to a component may cause it to split. The split operation generates a new partition object and a set of additional components: the new sub-partition tree is linked-in to the partition tree in place of the split component. The contents of the split component are re-injected into the structure to distribute them into the new buckets. (There is a slight danger that all the elements might be re-hashed into the *same* bucket, causing a ”split cascade." Allowing buckets to expand more than usual deals with this case.)

3.3.3. Directed Storage

Graphs – directed and undirected – seem to be the most ubiquitous structures in computer science. It is common, for example, to see a problem which may be treated simply as a problem in graph theory (most search problems fall into this category), and many applications have data structures which are based around the notion of a graph or tree.

A graph is a set N of *nodes* and a set E of *edges*. A node represents a “place” in the graph, whilst an edge represents a “route” between two nodes. Hence an edge may be represented by a pair (n_1, n_2) where $n_1, n_2 \in N$. If the order of the pair is irrelevant, *i.e.* if $(n_1, n_2) = (n_2, n_1)$, then the graph is said to be *undirected*; if order is

significant, the graph is said to be *directed*. Edges may be *traversed* from one of their nodes to the other: in a directed graph, traversal is only allowed in the direction of the edge, whilst edges in an undirected graph may be traversed in either direction. Variations on the basic theme allow nodes and edges to be *labelled* to identify them.

There is a sizeable body of knowledge on the mathematics of graphs, including algorithms for traversing all nodes and detecting cycles of edges. In particular, it is common for an edge's label to be interpreted as a weight designating the "cost" of traversing the edge, and an important class of problems involves minimising the cost of moving between a pair of nodes.

An important special case of the graph is the *tree*, which is an acyclic graph in which there is a single node, called the *root*, which is not the target of any edge.

Basic Approach

The basic approach to creating a graph is to store a node and its label (if any) alongside the set of edges which leave it: thus a node *A* would be stored with the edges *(A, B)*, *(A, D) et cetera*, but not with the edge *(B, A)* (for which *A* is the target) nor the node *(B, D)* (which does not affect *A* in any way).

A graph may be seen as being composed of a number of smaller sub-graphs, in much the same way that an array is a collection of smaller arrays. In the graph's case, the edges leading out of one sub-graph will be related directly to the edges leading into another sub-graph. This means that the sub-graph may be used as the unit of distribution, with a single sub-graph being stored in a single component.

From the point of view of navigation, a graph is the simplest of the partitioned architectures. All navigation between nodes must proceed on the basis of local information – the set of edges leading out of the current node. Therefore "resolution" – it can hardly be termed this in so simple a case – is simply a matter of looking-up the name of the target node of the edge to be traversed.

However, there are a number of complexities in representing graph structures. The first is the sub-division of a graph into sub-graphs. Since there may be no general rule as to where new nodes are added, a sub-graph may grow unpredictably: like an associative memory, but unlike an array, the bounds of a component cannot be fixed at creation-time. This means that identifying a sub-graph, and deciding when to create a new one, may be complicated. Furthermore, there is the issue of naming and deleting items from a graph.

Fortunately there are a number of special cases for which simple solutions exist. The most important is the tree, in which sub-graphs are actually sub-trees, which have well-defined properties.

Locality of Reference

A graph has a very well-defined notion of locality of reference. If an edge is seen as representing a single "step," then two nodes are close together if the number of edges which need to be traversed in order to move between the nodes is small.

The problem is complicated for the general case of graphs, however, as there may be many distinct paths between two nodes. This makes the problem of deciding

whether two nodes are metrically close together a difficult task: in the limit, it is one which can only be solved by an exhaustive search. The case is somewhat simpler for trees, however. In a tree, one may count the number of levels by which two nodes differ as a metric of the distance between them.

The locality of reference in a graph is vitally important. Any application manipulating a graph can only move between nodes by traversing edges: if such traversal results in a remote reference – as the target node of the edge is located in another component – then applications will incur a severe performance penalty. In spite of the complexity it is important that components contain sub-graphs as far as possible. There is a trade-off to be made between the penalty in finding sub-graphs against the penalty of remote accesses if applications make remote references out of a sub-graph which could be avoided.

Distributing Nodes

Distributing a graph, as mentioned above, may be seen as simply being a matter of dividing-up the nodes into a number of sub-graphs – not necessarily connected – which are then used as the basis for decomposition. This scheme is complicated by the fact that the node population of a graph may vary with time, with nodes being created and (possibly) deleted. This could lead to some convoluted distributions.

Consider a graph consisting of a single node, held in a single component. As new nodes are added as children[11] of this node, the contents of the component may grow until it is necessary for it to be split. This split then generates one or more new components, possibly re-injecting the existing nodes in order to balance the load. Unless care is taken, the nodes will end up being split in such a way as to destroy any possibility of locality of reference within the graph.

The first option would be to split nodes temporally: all new nodes are placed into a new component. For example, ten nodes are created in a component, but the eleventh causes a split to be made and is placed into a new component. A moment's reflection indicates that this will be unsatisfactory, as the distribution is completely unrelated to the topology of the graph.

Locality may be maintained, however, by ensuring that nodes which are metrically close – are centred around a common node, for example – remain in a single component. This would imply, for example that all immediate children of a node *A* would be located in the same component.

It is possible that in an arbitrary graph there are edges between arbitrary nodes, without any respect for the creation order. However, there is *no* simple solution for the arbitrary case short of exhaustive search, and creation-respecting distribution is a good solution for many of the special cases of graphs (trees and some commonly-encountered types of directed graphs in particular). If required, the exhaustive search technique could be used by re-defining the partition being used.

[11]The term *child* is, of course, rather inappropriate for a general graph, as all nodes are in some senses at the same level in such a heterarchy. The terms is apposite only for graphs like trees, with a well-defined sense of hierarchy. It is used here for convenience.

Naming Nodes

The most obvious naming scheme is to use the name of the object representing a node as the node's name. This is a direct translation of the traditional scheme for creating graphs into the partitioned environment: an edge is essentially a pointer to the target node. Such a solution has two main problems.

The first is the problem that, as objects in their own right, nodes can exist apart from directed collections. Related to this is the fact that, by giving clients handles onto objects which are part of the *internal* arrangement of the collection, it might be possible for clients to induce state changes pathologically. Allowing access to internal objects also weakens the collection's encapsulation.

In addition to these, passing out handles to nodes means that the collection is severely restricted in the ways in which it can re-arrange its internal structure. This is because the collection cannot safely destroy an object to which another, external client may hold a handle: to do this would result in a "dangling" pointer[12].

The alternative approach is to create some other form of naming for nodes which does not suffer from the above difficulties. In effect, this involves creating a naming scheme is which nodes are named by indexing values, in the same way as associative memories and arrays. For our current purposes, there is a restriction that such names lend themselves to simple distribution and resolution (since the names will be an integral part of the resolution process).

There are several possible naming schemes, but the most attractive is to name a node according to its position in the graph; or, alternatively, to name a node according to a path between it and some other node. In a general graph, there may be a number of paths between two nodes, but what *can* be guaranteed to be unique is the nodes' order of creation.

We shall make the assertion that all nodes except one (the first) are created as children of some other node which already exists in the structure. That is to say: a node may only be created by connecting it to an existing node using an edge. This has the corollary that a directed structure must always be *connected*. A node may then be named by taking the name of its immediate parent and affixing a unique identifier for it: for example, a sequence number in the creation of children for that node. A simple recursive argument shows that such names will always be unique. Essentially a tree structure built from the order of node creation is superimposed onto an arbitrary graph structure, as shown in figure 16.

[12]In most systems, anyway. There is no reason for not having objects which are garbage collected rather than explicitly destroyed, but few curent object-oriented systems take this approach, which is particularly tricky in distributed systems.

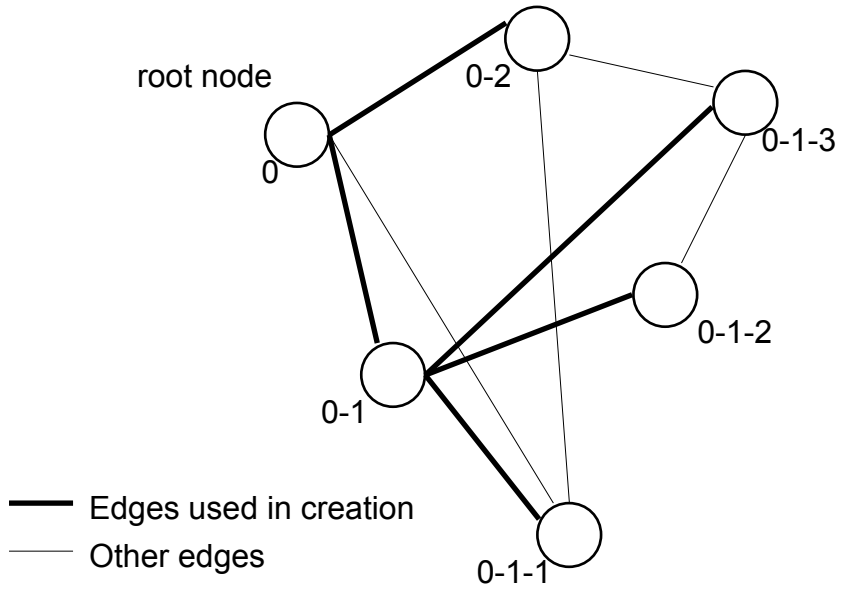


Figure 16: A strategy for naming graph nodes

There are obvious similarities between this naming scheme and the distributed extensible hashing. Node names are prefixed by the names of their parents, and occasionally the mapping of prefixes to components changes in order to re-order the collection. The differences are mainly at a higher level: node names are generated by node creation, not intrinsically from their value, and several "prefixes" may be held by a single collection.

Creation and Management

The creation of a directed structure is superficially closely related to the techniques described for associative memories above, §3.3.2.

Each component contains a particular set of nodes in the graph which have a small set of common parent nodes (the size of this set may vary). Nodes are created by providing them with an explicit parent node, and are identified by an abstract value. Internally, this value describes the creation path of the node – its parent, its parent's parent *et cetera* back to the root.

As nodes are created, the population of a component will increase. If the component's population grows too large, it may be split to form a set of new components, and its nodes may be re-arranged. The criterion controlling this is that all the direct descendents of a particular node will always reside in the same component: this imposes the restriction that there is a maximum fan-out within a graph which is governed by the maximum number of nodes which may be held within a single component.

Navigation in such structures is trivially easy. A node contains a set of edges leading from it (which may be interpreted as being directed or undirected as

required). The target of an edge is the name of its terminal node. The set of edges defines the set of possible resolution requests which may be made, and the node names contain enough information to perform resolution.

The actual resolution process is closely related to that of the associative memory. A nodes name to be resolved is compared with the set of node parents held by the current component and, if it does not match them, is forwarded to the partition. The partition then maps a prefix onto a component or sub-partition, or forwards it to its parent. Since the nodes names form a tree, the tree structure of the naming scheme may be matched by the partition tree structure, in much the same way as the distributed extensible hash space (but without the necessity of a hash function initially: put another way, the hash function for graphs is based on node creation placement and ordering rather than on a node's value).

3.3.4. Mathematical Structures

There is a temptation, when discussing data storage architectures, to provide some the "standard" mathematical objects such as sets, bags, mappings and the like. There is, however, a danger of embedding such structures into a programming language. Computing systems are not purely mathematical evaluators, and their needs sometimes conflict with those of the mathematics which they attempt to follow. Providing an object calling itself a set which does not behave *exactly* like its theoretical counterpart of far worse than not providing a set at all.

Some languages, notably Smalltalk, provide sets and bags as basic data types. There is, however, a subtle and (to our knowledge) previously unremarked problem with the Smalltalk implementation of these objects. It may be illustrated with the following example. Consider a Smalltalk set built from three `Point` objects:

```
s ← Set new.
s add: (1 @ 1) ; add: (2 @ 2) ; add: (3 @ 3).
```

where `(1 @ 1)` creates a `Point` object representing the point (1, 1). The set may be queried to see whether a given point is contained within it:

```
s includes: (1 @ 1).
--> true
```

Let us now insert a fourth point into the set whilst retaining its name outside:

```
p ← (3 @ 4).
s add: p.
```

Since we have retained the name of the object, we may still interact with it, and one of the possible operations is to induce a state change by altering one of its ordinates:

```
p y: 3.
```

changing the point to represent (3, 3) instead of (3, 4). The result of this is that the set `s` now contains *two* points with the *same* value, which violates the invariant of sets – but the set has no way of knowing this. Enumerating the contents of the set will result in two values the same: if, for example, we sum all the points in the set

```
s inject: (0 @ 0) into:
   [ :acc :elem | acc + elem ].
--> (9 @ 9)
```

the result is incorrect – it should be (6, 6). By covertly changing the state of an object we can subvert the properties of the set. The same is true for a bag. The point is that it is difficult to import value-based abstractions into a state-based computational framework.

The problem may be overcome, but not easily: three possible remedies are sending "value changed" calls to all collections containing an object whenever its state changes, using copy-in semantics for objects in collections, or using invariant semantics. The first solution incurs a large overhead, and requires that all objects in the system (both built-in and user-defined) follow this convention; the second may be undesirable if it is actually a particular object, rather than a value, which is to be stored; the third restricts the objects which may be placed into such collections to those which do not change their state.

In choosing the array, associative and directed storage architectures for the partitioned model, we side-step these problems. These architectures make no unworkable guarantees, but may be used to create any desired structure in a scalable and distributed manner. A set, for example, can be implemented as an associative memory. It is then the programmer's task to devise a suitable semantics for cases such as the one described.

3.4. Creating User-level Data Structures

Having discussed the infrastructures of scalable memory, the questions arise: how may these frameworks be used to construct "real" applications? To what extent may users customise the memory access protocols and distributions whilst still being shielded from the low-level details of implementation? This section addresses these issues.

3.4.1. Customisation and Refinement

For a programming environment, customisation is important: the ability to extend or adjust features of the supplied software so that it better matches the task in hand.

Refinement is the process by which something is "made better" in some sense. In computing, the term often denotes a method by which an abstract description of a program is converted, by successive steps, into a more concrete version of the same program. In other words, refinement is a *semantics-preserving program transformation*. An example of refinement is top-down design, where a program in

developed in terms of high-level structural components, each of which is decomposed recursively until an "atomic" stage is reached.

The architectures discussed above provide a framework for particular kinds of storage, but will only provide a rudimentary interface to the programmer. The great advantage of object-oriented programming, however, is the ability to encapsulate the functions of one class within the interface of another, using the sub-classing mechanism. This allows the programmer to create data structures offering high-level interfaces whilst using the basic architectural implementations.

Scalable memory, as observed before, essentially abstracts away from four low-level details:

a) *where* an individual datum is stored;
b) *which* data are allocated to which local memory;
c) *how* data values are accessed by clients; and
d) *how* data values are retrieved when requested (internally).

The basic approach when creating a user-level structure is to provide an interface which is useful to the programmer in performing some task, masking the features of the selected storage architecture which are not relevant to its use.

We shall here consider the reasons why customisation might be performed, and the sorts of things which might reasonably be customised in an environment based around the partitioned model. We shall leave until later the issues of exactly how this customisation occurs, when we discuss a prototype implementation of a partitioned-model environment (chapter 5).

Custom Access

The partitioned model implements storage architectures, not data structures *per se*. The programmer's interface to an architecture is derived from the architecture but is not intrinsically a part of it, so different interfaces to essentially the same architecture are perfectly possible.

Why would different interfaces be needed? The obvious answers concern functionality and grain size.

Functional Interfaces

A storage architecture provides only the most basic operations: retrieve an element, store an element, remove an element, move an element *et alia*, depending upon the exact architecture being used. An application might conceivably use such an interface, but only at the cost of a large amount of readability.

An application in general builds structures to hold values of a given type, and a data structure will usually be constructed so as to hold values of only a single type. The exact type system used is a feature of the host language: object-oriented systems usually provide inclusion-polymorphic type systems, but one might also consider type systems based around a more (or less) powerful base. The storage architectures themselves make no explicit reference to the type of values which they store, although they may place a small constraint upon its type (such as the existence of a

hash function or an equality operator). The same architecture may be used to store values of *any* type representable within the host language: an application may require that a structure has its type (*i.e.* as an array of integers) fixed before its use. Thus there is an immediate need for a functional interface: to restrict the types of object which may be manipulated by the basic access operations.

A second need is when the memory performs more complex operations than simple accesses. An example would be a database (represented as an associative memory with customised add and retrieve operations). The database may need to perform (for example) a "project" operation, placing additional constraints on the elements' types.

Grain Size

Although we have tried to avoid discussion of effects which are purely caused by distribution, eventually these effects must be considered. Since all requests for data may in principle cause communications to occur, it is desirable from a performance point of view to reduce the amount of communication incurred. This may occur in two ways: by transferring more data per communication, or by exploiting the principle of locality. The latter will be dealt with later; the former is the issue of grain size.

In discussing their application which won the Karp prize (awarded for an application for exhibiting a large parallel speed-up), Gustafson *et alia*[55] identify several aspects of their application which contribute most to its success. One of the most important is that the application transfers data in the largest "chunks" possible – tens of elements rather than single elements. In terms of programming interface, this implies that the interface would allow access to several records simultaneously, through a single call, rather than forcing an application to perform the accesses individually.

The grain size of accesses may be altered in two ways. The first is to add a function to the architecture's interface which repeatedly calls the single-element-access functions invisibly; the other is to implement a function which accesses the inner storage mechanism of the structure directly, in the manner of the single-element functions, to acquire several elements. Both alternatives have their attractions in different circumstances. The first method, by using the provided access methods, acts essentially as a client object: it *adds* to the interface of the base class without subverting that which exists already. The second method uses the same internal information used by the base class, and may be more efficient than the first method but only at the cost of weakening the abstraction of the basic memory module.

In suitable circumstances, one might also consider the use of techniques such as pre-fetch caching of data without altering the external interface. Internally all requests would be performed in large chunks, with those elements which are not immediately required being cached for later use. This opens up a whole new area of problems in terms of cache consistency, but illustrates the fact that a partitioned memory may be very flexible in its internal organisation; moreover, it illustrates that the memory interface may become very intelligent without affecting the interface resented to client objects.

Custom Distribution

Providing a custom distribution may involve changing several factors in a memory: the way in which storage is decomposed, the way in which resolution occurs, *et cetera*; alternatively it may require quite trivial modifications. This means that the distribution management routines must be decomposed to a very fine grain, to avoid the need to re-implement what is essentially common functionality.

Parallel applications often have a distribution pattern which, in some sense, they "prefer:" the application is highly efficient when its elements are arranged in a particular topology and is less efficient in other configurations. If – as seems likely – this distribution cannot be inferred from the application's source code, it is necessary for the programmer to take a hand. The two means of approaching this problem are to place elements explicitly, or to create an automatic distribution manager which is specialised towards the application's needs.

Placed Distribution

It is quite a simple task to create a distribution manager class in which it is possible to specify exactly the distribution of components. The most obvious is a manager specialised towards distributing two-dimensional arrays which places components in a regular grid on a mesh-based machine. This ensures that neighbouring locales of the array are on neighbouring processors. The programmer has specified that a particular placement of data on processors is the most efficient for the application.

This new distribution – whilst having no semantic effect on the application's behaviour – may have a profound effect on its performance. It may be used to minimise the overhead incurred in the expected sort of resolution, that which is needed to acquire a value from a neighbouring locale.

Indeed, it is possible to go a stage farther and inform components of the locations (*i.e.* component names) of the other elements of the array. Essentially this involves using the partitioning infrastructure as a decomposition and configuration tool which is disposed of after setting up the structure. Evidently this only works for structures which have a fixed size and distribution which may be determined "statically:" the quotation marks here emphasise that the distribution may be fixed during the lifetime of the structure and may not vary afterwards, but does not necessarily have to be determinable when the structure is *first* created.

All these approaches are completely scalable. Using the example of the mesh-based array distribution manager, it is possible for the manager to determine the maximum size of mesh which may be created at run-time, and to create a mesh of this size. A further stage of placed distribution is when the programmer actually specifies the processors which will contain the various components, but this then restricts the application to execute on a system with (at least) the specified number of processors.

Adding Factors

Another approach is to create a more "intelligent" distribution manager by including new factors which affect the algorithm. This is a slightly different

proposition to the above: the system is still completely free to choose whatever distribution it determines would be best, but has more information available to make this decision.

Such a system is particularly attractive in the presence of object migration, when components (and their elements) may be moved at run-time. This would allow the distribution manager to observe the actual access patterns which occur at run-time on a particular run and alter the distribution of data dynamically in order to improve the locality of reference, communication overheads *et cetera*. This is in many ways a generalisation of the traditional object-migration approaches which collect statistics on single object interactions: a distribution manager may collect statistics on a multiple-object structure and re-distribute it.

At its most comprehensive, it would be possible for a distribution manager to achieve a near-optimal distribution of the data in a structure, and to vary this distribution to maintain its optimality across different patterns of access.

Caching

Another possible refinement involves the use of caches.

It may be observed that, when an application exhibits locality of reference, most of its requests will be sent to a small number of components in the collection – the ideal case being where all requests go to a single component with which the client is co-located. In a less-than-ideal case, however, it may be advantageous if the resolution required to access remote components is avoided.

A possible method is to cache the components which resulted from the last few resolutions. If a request for the same component occurs again, then the request may be forwarded directly to the correct component without the need for resolution.

There are obviously some additional requirements to this use of caching. Firstly, the cache must be able to recognise requests for the same component, and this implies that the cache has access to the same information as the resolution algorithm as regards the component names which are cached. In an array, for example, the cache must store the region held by a cached component alongside the component's name so that the necessary test for locality may be performed. Secondly, the cache must be advised of the component which was the end result of each request. Thirdly, the cache must be updated whenever a request is resolved to a different component (in associative and directed structures, this would occur when a component is split or joined). For all these reasons, it is best to perform this form of caching within the partition class, rather than in the components.

An important point about this form of caching is that cache entries are only hints. To illustrate this, suppose that a cache entry is held to a component in an associative memory which has been split without the cache having been updated. A request which is resolved using the cache will then possibly be forwarded to the wrong component. However, the definition of partitioned collections states that *any* component may resolve *any* request, so this mistaken forwarding will at worst result in another, subsequent resolution of the request: the use of an out-of-date cache entry is not disastrous, but will simply alter the time taken to service the request.

One might also suggest that recently-accessed *values* of requests are cached, although this introduces problems with cache consistency which cannot be addressed well in any general way: they are of necessity application-specific.

3.5. The Semantics of Failure

One of the most common claims made for multicomputer systems is that they offer increased possibilities for the construction of reliable, fault-tolerant computers, since they have no single points of failure. In a scalable system, with potentially a very large number of processors, the probability that some node will fail grows along with the system itself. A point which we have not as yet addressed is the possibility of failure in a partitioned system, either from faults in implementation or faults in the underlying hardware. Although the partitioned model – in common with most other systems for parallel programming – assumes that no failures occur, there are many features of the model and of scalable systems generally which make it well-suited to extensions to deal with such failures.

The toleration of software faults is a commonly-occurring theme in dependable systems research, and is discussed extensively in (for example) [4][107]. It is usually addressed by techniques such as multiple-version programming, coupled with extensive testing before execution.

Hardware faults can take two forms: link failures and node failures. A link failure results in the destruction of a communication path between two processors, while several concurrent failures may "partition" the network into two or more parts which have no communication paths to each other. A node failure will result in the loss of any data stored and processes running on the node, and may also exhibit all the features of a link failure on all links to the failed node.

In general an application has no control over the routing of messages: hence link failures are essentially the domain of the operating system kernel. The solution is to detect the fact that a message has not been delivered (using time-outs, sliding window protocols *et cetera*) and to re-transmit the message using a different route to bypass the failed link. This may be implemented transparently by the kernel's routing module.

Node failures present a more thorny problem. There are many issues to be considered: whether failure should be handled transparently or should be visible to client objects, whether data should be automatically or manually committed to stable storage, whether replication is worth the additional consistency overhead, and so forth. Within the partitioned model, the problem is essentially concerned with maintaining both the *contents* of the memory and the *structure* of the storage architecture.

Tolerating Faults

Consider the case of a partitioned collection executing on a set of nodes. If one of the nodes crashes – we shall assume that nodes are fail-silent – the result will be

the loss of one or more component and partition objects. There may also be activities running in conjunction with the components.

If a component is lost then all the data in that component will disappear and will need to be re-created somehow. We shall defer this issue briefly. Similarly, node crashes which affect activities will also require that the activity (including its context) is re-created and re-synchronised with the rest of the application. Neither of these faults may be "tolerated."

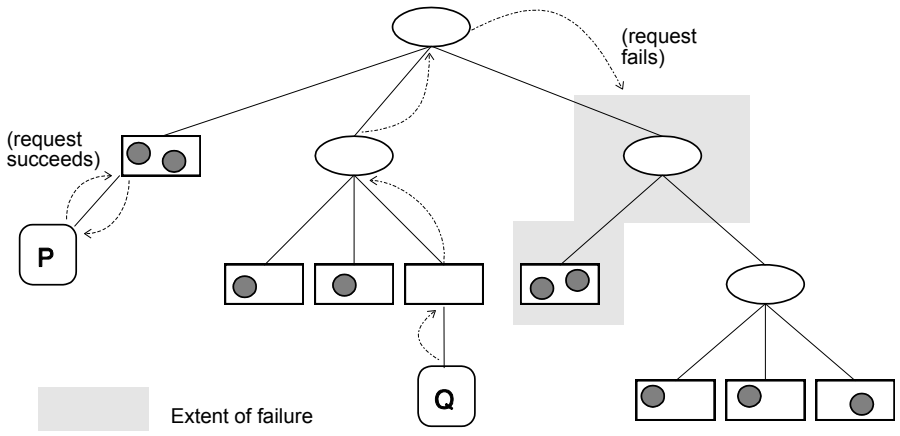


Figure 17: The effects of a node failure on resolution

If a partition node is lost, the result will be the isolation of part of the partition tree from the rest – the usual term for this is "partitioning," a usage which we shall avoid here! This is shown in figure 17: the part of the tree below the crash site is isolated. In the figure, the request from process **P** succeeds as it does not intersect with the crashed node; the request from **Q** fails. Local requests for components below the crashed partition – or remote requests for any sub-trees below it – would also be able to function providing that their resolution path did not intersect with the crashed node.

However, it may be highly desirable to tolerate failure by allowing the disjointed trees to interact – allowing, for example, process **P** to access elements below the crashed partition. Here a feature of the partitioned model comes to our aid: the fact that any request may be resolved from any partition. Conventionally a component makes a request for resolution to its own parent, which in turn interacts with its own parent and direct descendents. Another possibility, in the case of a partial failure, would be to choose a partition node at random and forward a request through it.

An algorithm for this form of fault-tolerance is as follows. A partition, on receiving a request, attempts to resolve it in the normal way. If, in the course of resolution, a partition tries to interact with a crashed node, it detects this and takes remedial action. It chooses another node from the tree at random and forwards the resolution request to that node. There is a chance that this node will be able to resolve the request without hitting the crash site, and will then be able to service the request; if it does hit the crash site, it make take the same steps.

Such a Monte Carlo algorithm always has the possibility of never terminating – if, for example, the request being resolved is targeted at a crash site itself, or through a unfortunate sequence of random selections. One might take the view that such behaviour is acceptable; alternatively, a request may have a built-in threshold on the number of stochastic resolutions in which it may be involved, after which the request fails.

The algorithm, it will be noted, is implemented purely within the partition classes, and so is simply a refinement within the partitioned model for a particular distribution strategy. The only effect on component classes is that there must be some error-handling mechanism which is triggered if a request cannot be satisfied – an exception or an "empty" return value.

It should also be noted, however, that the algorithm is incomplete: it cannot *by any means* access components which are *direct* descendents of a crashed partition, as the necessary routing information is missing. One might modify the algorithm so that it randomly interrogates components to see whether they can satisfy the outstanding request, but this seems a little *too* stochastic. We shall content ourselves with the observation that a partitioned memory can tolerate a certain amount of node failure to the extent of degrading gracefully, but cannot completely hide the effects of the destruction of its internal structures.

Recovering from Faults

Fault recovery requires three linked steps:

- the re-creation of any lost objects;
- their re-integration into the remnants of the structure; and
- the re-creation or re-acquisition of information (including contextual information) which was lost.

Although a partitioned collection can tolerate (at least partially) the loss of some of its distribution managers, it cannot re-generate a structure automatically; nor is the tolerance of faults fully satisfactory.

A partition object holds sufficient information to allow it at least partially to re-generate any objects below it. In an associative memory, for example it can identify the prefix of a sub-partition which has been lost, and then re-create this partition using the same mechanism as that by which the partition was originally created. The difference is that the re-generated partition must be able to link in the partitions below it into its own routing tables, and must re-create the components (and their data) which were held by it.

Although it is simple to create and assign storage for components, it is impossible to re-create their data: for this, it is necessary that components have periodically persisted their data onto stable storage.

The basic idea behind the use of stable stores is that data is placed onto a disc or other non-volatile storage medium and retrieved whenever it is necessary to recover from a failure. Arjuna[43] is a good example: all data manipulations are implemented as transactions which must "commit" before any permanent change is

made to an object's data. This also means that, if an Arjuna object is lost, a consistent image of it may be recovered from disc.

The partitioned architectures discussed in this chapter have not used transactions as their basic *modus operandi*: there is, however, no reason why a transaction-based interface modelled on Arjuna might not be implemented. This would then yield a scalable memory model which was highly resilient to failure: it would continue to function in a degraded manner if a node failed, and could re-create itself from an image persisted onto disc in order to avoid loss of data.

Replication

An alternative to the reconstruction of data from disc is the storage of data at several points concurrently, in the hope that at least one replica will survive a failure.

One could certainly implement some form of data replication within the partitioned model by replicating the components (and possibly the partitions) composing a partitioned memory. A failure of a node containing one of the replicas could then be tolerated by activating one of the others.

A possible amendment to the standard generic collection architecture is shown in figure 18. All the elements of the structure are replicated by "shadow" copies. There is always a single primary copy to which commands are sent, but the partition also forwards requests which alter the targeted object to all shadows. For example, a request to add a node to a tree should be made to both the main and shadow copies, whilst a request to traverse a link need not be forwarded. The result is that all the shadows remain up to date.

Concurrent processing is not affected by this organisation: if an activity is attached to the collection, replicas are generated solely at the primary components, not on the replicas.

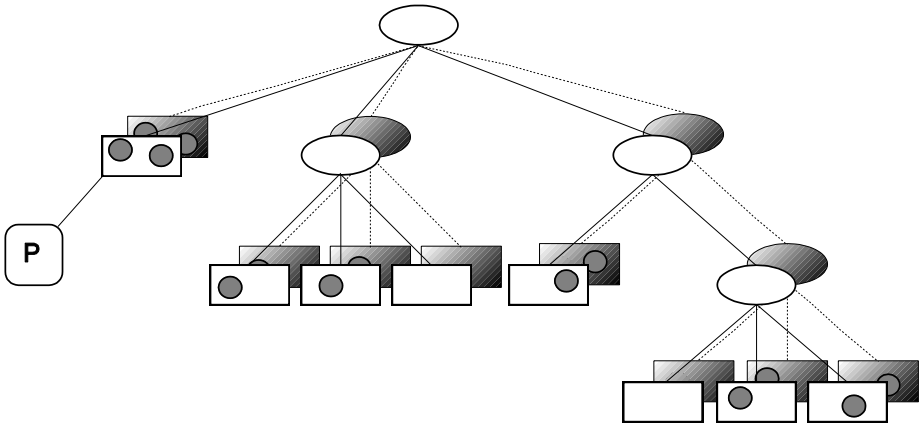


Figure 18: Generic collection with replicas

Suppose that the node containing the primary copy fails. The partition will detect the failure when it attempts to resolve an element in the failed component, and will then select a shadow to become the new primary copy. For this to make sense it

is essential that shadows reside on widely separated nodes to avoid a node failure destroying the shadows too. Once a shadow has been selected, the interrupted request may continue.

A similar argument holds for crashes involving partitions: the partition's parent or one of its immediate children will detect the failure and activate a shadow.

There is a considerable increase in the complexity of the partitioned structure to accommodate replication – it should be noted, however, that almost all of this complexity is encapsulated within the partition classes. Some small changes are also needed in the handling of requests to local data: these requests must also be forwarded to the component's partition for forwarding to the shadows.

These are comparatively trivial changes, however: the result is a partitioned collection containing a set of "hot" stand-by memory modules whose consistency is maintained automatically and which are activated when necessary to replace failed objects.

3.6. Résumé

This chapter has presented a set of techniques for implementing scalable strongly-typed memory modules of the type required for the scalable abstract machine. The requirements of such an implementation were first discussed, along with some possible implementation strategies. The most practical solution was determined to be the co-ordination of several objects to form a multiple-object entity which behaves as a single logical resource.

An overview of the technique, called *partitioning*, was presented. For the three most common memory architectures – arrayed, associative and directed – appropriate special techniques were derived based on the partitioned view of memory. This involved a discussion of the ways in which such memory architectures can be decomposed, and the ways in which the internal details of the decomposition can be masked. As part of this, a new hashing algorithm was developed which is completely distributed and scalable, in order to implement large associative memories.

The storage architectures provide only the most rudimentary storage facilities, akin to the basic read/write operations of hardware memory. In a programming environment, it is essential that higher-level interfaces are provided. The issues involved in customising the data interfaces to partitioned memories were considered.

In terms of performance, a tension was recognised between the needs of generality (for a programming environment) and the needs of efficiency (for particular applications) in terms of the distribution of components of a collection. The partitioned model allows distributions to be customised apart from the data manipulation classes, using either manual placement or an extended automatic approach to distribution.

Some consideration was given to the effects which failures of nodes and links might have on a partitioned collection. In general it was argued that a partitioned memory can tolerate failures and continue to operate in a degraded manner. An algorithm was presented which, when failure occurs, attempts to route-around failed parts of the structure. The loss of data implicit in node failure was recognised, but it

was suggested that this might be tolerated by importing some of the techniques used in distributed fault-tolerant systems. The shadowing of components and partitions was argued to be particularly well-suited to the model: such replication causes only small changes overall, although it implies the use of a suitable access mechanism such as one based around transactions.

Chapter 4.

Concurrency in Scalable Systems

Obviousnes is always the enemy of correctness.

Bertrand Russell

In the previous chapter we developed a collection of techniques for implementing scalable memory modules, as required by the abstract model of chapter 2. We shall now consider the ways in which concurrency affects the construction of applications in this fashion, and the ways in which concurrency may be expressed.

Concurrency confronts the programmer with two issues: concurrency *control* and concurrency *regulation*. The former deals with ensuring that processes do not malignly affect one another's operation by simultaneous (or interleaved) accesses to shared data, which may result in inconsistencies. The latter addresses the manner in which the number and location of processes in an application are determined. These issues are closely related, and are both complicated by scalability.

We shall begin by discussing the nature of concurrency in object-oriented systems, and then go on to discuss concurrency control. We shall consider the effects which various forms of concurrency control may have on systems constructed using the partitioned model, and hence decide upon a suitable concurrency control model.

We shall then discuss concurrency regulation, and consider the ways in which scalability complicates it. It will be seen, however, that the partitioned model allows one particular form of concurrency – multiple workers accessing a shared data set – to be regulated very easily. Furthermore, the model may be used to create process structures by creating scalable collections which are composed of processes.

4.1. Concurrency in Object-oriented Systems

We shall first consider the nature of a "process" within an object-oriented system.

Computation in object-oriented systems occurs through sequences of method calls. Method calls behave like traditional procedure calls: the caller blocks until the called method completes, whereupon it resumes computation. In a distributed object system, method calls resemble remote procedure calls[94] and the analogy still holds, although the method call may be executed by different threads if the caller and callee are on different processors.

A single logical locus of control, therefore, may be seen to animate several objects in the course of a computation: when a method is called, the site of the locus shifts from the calling method (which blocks) to the called method (which executes). When the called method terminates, the locus shifts back to the caller to unblock it. By all conventional definitions, this locus is a process: a single logical activity performing computation, albeit moving between processors in the course of its activities[13].

If we assume that there exists a single locus of control when an application is created, there is no way, in this scheme, to introduce new processes – a method call being simply a shifting of the focus of the same thread. In order to generate concurrency, a mechanism for creating new loci of control is required.

Some languages (such as Orca[14]) take the view that processes may be created only by creating a new object. When created, some objects execute a method whilst still unblocking their creator: the new method is thus a locus of control independent of the creating thread. Whilst workable, this strategy seems a little at odds with the ideas of object-oriented programming: rather than restrict the introduction of concurrency to specific methods, called in an object's constructor, it would seem more appropriate to allow concurrency to be generated by *any* method.

Method-level concurrency may be obtained *via* two routes: asynchronous calls or asynchronous returns. In the former, certain methods are designated as being called asynchronously: the caller does not block when the method is called, and both methods proceed concurrently. In the latter, a method is called synchronously but is allowed to return a value (thus unblocking its caller) without terminating its own execution. The difference between these two approaches is shown in figure 19.

Problems arise when asynchronous methods are allowed to return values to their caller in the manner of a conventional function call, since the returned value will be undefined for the period before the called method returns. This may be tackled in two ways: by using a mechanism such as futures[56][79] to control access to the value prior to its resolution, or by disallowing asynchronous methods from returning a value. The former is more flexible, the latter simpler. In any case, the functionality of futures may be implemented using objects without the need for extra syntax.

[13]We are speaking here of *logical* processes, of course. At the lowest level, calling a method on another processor will almost certainly utilise a different *physical* process to the caller.

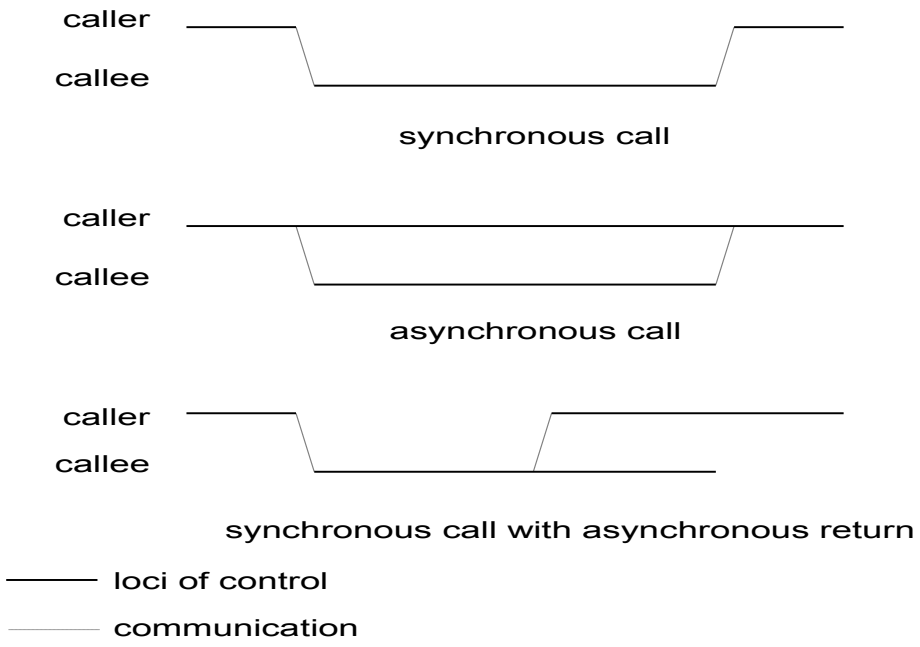


Figure 19: Different styles of asynchronous method call

A process is created whenever another process makes a call to an asynchronous or asynchronously-returning method. The new process then executes independently of its creator.

4.2. Concurrency Control

Concurrency control – also known as ensuring *sequentiality* – is one of the classic problems of computer science, dating back to some of the first high-level languages. The problem is to eliminate the danger that two or more processes, while accessing a single shared data structure, will interfere with each others' behaviours; at the same time, the overheads which this protection introduces must be minimised.

Concurrency control began on the first shared-memory machines that implemented either lightweight processes (or *threads*) or co-routines. In both cases, the logically separate activities composing an application share a common address space: lightweight processes are scheduled pre-emptively, so that a process may be interrupted at unpredictable intervals, whilst co-routines are scheduled co-operatively and must voluntarily yield control to another co-routine.

Co-routines avoid most problems of concurrency control, as they can ensure – at least in the absence of interrupts or other unexpectedly pre-emptive events – that any shared data structures are left in a consistent state whenever they yield control. The price of this safety is that the programmer must explicitly place yield statements into algorithms to ensure that other activities are not locked out, and must ensure that these changes of control occur only at “safe” points.

For threads, changes in control flow are caused by the underlying scheduler. They may occur when a thread becomes blocked on some event, exceeds its allocated time-slice, or when an interrupt occurs: in other words, the scheduling of threads follows the familiar scheduling model adopted in most operating systems for "heavyweight" processes running in separate address spaces. The use of pre-emption frees the programmer from placing explicit yields into code, at the price of having to ensure that shared data structures are protected from corruption if a context switch occurs in the middle of an update operation.

The two most popular – and most studied – approaches to concurrency control are Dijkstra's semaphores[41] and Hoare's monitors[60]. Variations on these themes include path expressions, critical regions, protected records *et alia*[106]. These methods are all *pessimistic* concurrency control protocols, as they attempt to prevent interference from occurring: an alternative is the *optimistic* or *roll-back* strategy which endeavours to repair any interference after it has happened by restoring a consistent state. (These schemes are most commonly encountered in simulation and database systems.)

4.2.1. Concurrency Control in Object-oriented Systems

Object-oriented systems differ somewhat from traditional shared-memory systems in that they follow different rules of encapsulation. Concurrency control's main effect is on the manner (if any) in which several methods may execute simultaneously within one object.

Threading in Object Models

Objects provide an obvious unit for concurrency control: one must ensure that the internal state of an object remains consistent. This means that there may be sequentiality constraints between the possible methods.

Single-threaded Object Models

The simplest concurrency controller imposes the restriction that at most one method may be invoked on a single object at any time. This ensures that at most a single thread of control is accessing the object's state at any time: provided that methods always leave the object's state consistent across their operation, there can be no interference. All concurrency control is implicit, so the programmer need not provide any additional information. This strategy is adopted by the Orca language, amongst others: it essentially imports monitor-like semantics into the object-oriented domain.

Although perfectly acceptable as a solution to prevent interference, problems arise when single-threaded objects attempt to interact with each other. Suppose that an object A calls a method on object B. Objects A and B are now locked to method calls from other objects. If, in the course of the call, object B calls a method on another object C, there are three objects which are locked as far as the rest of the system is concerned – none of the objects may be unlocked whilst a method call is in

progress, as there is no guarantee that their internal states are consistent. What is more, it may not be possible for object C to make a recursive call to one of its own methods: even if this case is recognised and allowed, it will not be possible for object C to make a *mutually* recursive call to objects A or B.

In a parallel system, the progressive locking of many objects during method interchanges will inevitably cause bottlenecks. Although single-threaded objects ensure intra-object consistency, they can be rather awkward to use when inter-object interactions occur – which, of course, is the norm in a well-decomposed system.

Multi-threaded Object Models

By allowing several method calls to be in progress within a single object at any time, we effectively re-introduce the "classic" problems of concurrency control encountered in the case of shared-memory systems: in this case, the data being shared is the local state of an object. The situation is somewhat improved, however, in that the number of operations which may be performed on an object is strictly defined by its interface.

Given that the constraints upon the methods are defined correctly, multi-threading solves the problem of recursive calls mentioned above. Moreover, in many cases the progressive locking of whole chains of objects will not occur: the fact that an object is engaged in some operation does not preclude another operation from executing, providing that the operations are compatible in terms of the object's sequentiality constraints.

If multi-threaded objects are used, a means must be provided by which object designers may specify intra-object concurrency control constraints.

Specifying Intra-object Concurrency Constraints

The simplest solution for the language designer is to force programmers to implement their own concurrency control strategies, using semaphores or some similar primitive. This solution is adopted in Smalltalk, where it imposes a particularly heavy burden as the standard classes are not safe for use in a concurrent environment. In systems where concurrency is the norm rather than the exception, such approaches are unacceptable.

A slightly better approach allows a distinction to be drawn between *read-only* and *read-write* methods, where a read-only method does not alter the object's state. Several read-only methods may progress concurrently in complete safety, but read-write methods must execute alone. The programmer need only supply a single piece of information – the category to which each method belongs – and the system may implement the necessary controls automatically. This is the scheme adopted in Emerald[63]: depending upon the language's structure, however, it may not be possible for the compiler to check that a method designated read-only is indeed read-only. The Emerald compiler assumes that the programmer correctly designates all methods.

An even more flexible approach is used by the DRAGOON language[7]. DRAGOON provides syntactic structures for creating descriptions of permissible method interactions according to a deontic logic. It is possible, for example, to

control the number of instances of a method which may be executing concurrently; to force methods to execute only in a particular sequence; and to specify whether methods may execute concurrently. From these descriptions, the compiler can generate appropriate concurrency control protocols transparently.

An important problem with DRAGOON's concurrency controllers is that they cannot be inherited: once a class is assigned a concurrency controller, no sub-classes may be derived from it.

Arjuna's concurrency controllers[96] are simply objects in their own right, with each object possessing a concurrency controller as part of its local state. Each method, when it begins executing, registers with the concurrency controller *via* a method call which only returns when the concurrency controller will allow the method to proceed. At the end of execution, the method informs the controller that it has terminated. The basic concurrency controllers in Arjuna are specialised towards transaction-based distributed processing, but there is no reason why other forms of control – optimistic or pessimistic – cannot be implemented.

4.2.2. Concurrency Control and Scalability

We must now consider the effects which the various schemes for concurrency control would have on a scalable system, in which the amount of concurrency is very large and unpredictable. In particular, we must consider what concurrency control strategies are most appropriate for use within partitioned collections.

Components

Components perform all the data-access operations for collections. Although these operations may be very complex, due to sub-classing, there are five main primitive tasks which a component must perform and from which other operations may be built:

a) create or delete its local storage
b) read from local storage
c) write to local storage
d) resolve a request for a (possibly remote) element onto a component, which may be itself or some other component
e) (for some architectures) re-arrange its local elements between other components, or amalgamate more elements into its own storage

Task (a) occurs only when a component is created or deleted. It may be assumed that at this point no user-generated actions may reasonably be serviced, so the component should be locked to all actions except those directly related to the creation or deletion operation.

The other four tasks may occur at any point during the component's lifetime. Task (b) is a read-only operation, which may occur in parallel with other such operations. Task (c), on the other hand, must have exclusive access to the local

storage – we are assuming here that locking for read-write operations occurs on a component-wide basis.

Task (d) will occur as a necessary prelude to tasks (b) and (c), identifying which component is to receive the local access task. Many instances is task (d) may proceed in parallel.

Task (e) – which is not applicable to all storage architectures (for example fixed-size arrays) – is similar in its scope to task (a): no other operation may sensibly proceed while the component's storage is being re-assigned.

A component's primitive tasks therefore fall naturally into three categories for concurrency control purposes:

- tasks which involve the local storage of the component – (b) and (c);
- tasks which are concerned only with forwarding – (d); and
- tasks which affect the mapping of elements to the component and its fundamental structure – (a) and (e)

The first category may be further sub-divided into tasks which access local storage in read-only mode, and those which require read-write access.

Partitions

Partitions perform the creation of components and the resolution of element requests. Like components, there are a small number of primitive tasks which they may perform:

a) divide a set of elements into a partition tree
b) (for some architectures) re-map descendent elements, creating or deleting components and sub-partitions
c) resolve requests for elements

Task (a) occurs whenever a partition is created – either at the collection's creation or as a result of its growth. Until the division of elements has occurred, it is impossible to perform resolution: therefore no resolution requests may be accepted until the process is complete.

Task (b) – which only occurs in some architectures – is similar to task (a) in that it involves the mutation of the structures necessary to perform resolution.

Task (c) uses the partition's internal tables to perform resolution. It is a read-only operation, and may proceed with other resolution requests; it must be blocked when one of the other tasks is in progress.

There are only two categories of task within partitions, then: those which affect the integrity of the tables used for resolution, and those which simply uses these tables.

Concurrency Controllers

We shall now use the foregoing analysis to derive a suitable concurrency control regime for use in partitioned collections.

The division of components' and partitions' primitive tasks into categories mitigates against the use of the single-threaded objects; similarly, it would be unacceptable to use a simple read-only/read-write distinction for partitions as there are three categories of method. Moreover, we should like if possible to avoid embedding concurrency control information into the syntax of the language, as this means that sub-classes cannot provide other, more sophisticated protocols if required.

Deontic Logic Concurrency Control Objects

The auxiliary concurrency controllers of Arjuna are very attractive: they allow concurrency control to use the full facilities of the underlying host language to perform its task, rather than using a restricted (and possibly restrictive) sub-set. The mechanism used, however – transactions coupled with support for highly reliable programming – is rather unsuited to the needs of highly parallel processing. We might suggest, therefore, that the idea of concurrency control objects be used without using Arjuna's control policy.

We shall instead adopt the deontic logic of DRAGOON to specify constraints. The reasons for this choice are quite simple: firstly, the constraints being expressed within partitioned collections may be captured very succinctly using this representation; secondly, the use of a deontic logic removes implementation details from the specification. Since constraints are expressed as logical statements, rather than in terms of locks, they are more easily analysable.

DRAGOON's Deontic Logic

In order to specify the sequentiality constraints required by partitioned collections, we must first review the deontic logic developed for DRAGOON. We shall then use this logic to provide the specification.

The logic consists of a single predicate function *per*. A specification is composed of a number of clauses of the form

$$per(op) \Leftrightarrow e(t)$$

where *op* is an operation and *e(t)* is a time-variant Boolean-valued expression. This statement may be interpreted to mean that *op* has permission to execute if and only if *e* is true. The logic may be easily extended to deal with multiple sets of operations having a common constraint: if *OPS* is a set of operations then

$$per(OPS) \equiv \forall op \in OPS \bullet per(op)$$

which divides the available methods into a set of equivalence classes.

In order to define *e* three monotonically-increasing functions are maintained for each object. These functions record the number of events which have occurred since the controller was started:

req(OPS) – the number of requests for execution by an operation in *OPS*;
act(OPS) – the number of operations from *OPS* which have been given permission to start; and
fin(OPS) – the number of operations from *OPS* which have terminated.

The nature of these functions requires that $req(OPS) \geq act(OPS) \geq fin(OPS)$.

There are some functions which are so common that they are best provided as standard:

- *active(OPS)* – the number of operations from *OPS* currently executing, defined as *act(OPS) - fin(OPS)*
- *requested(OPS)* – the number of currently outstanding requests for operations in *OPS*, defined as *req(OPS) - act(OPS)*

Using these functions, it is possible to specify a number of important concurrency control constraints. Some examples may make this clearer. Given two sets of methods *A* and *B*:

a) at most one operation from each of A and B may be executing at any time:

$$per(A) \Leftrightarrow active(A) = 0$$
$$per(B) \Leftrightarrow active(B) = 0$$

b) any number of operations from A may execute concurrently, but at most two operations from B may be in progress at any time:

$$per(A) \Leftrightarrow true$$
$$pre(B) \Leftrightarrow active(B) < 2$$

One may define another condition which commonly occurs: where an operation must execute exclusively. This is represented in the logic by the symbol *exclusive*[14]: for any O_1 to O_n of sets of operations,

$$per(O_i) \Leftrightarrow \forall j | 1 \leq j \leq n \bullet active(O_j) = 0$$

[14]In DRAGOON this property is represented by ><, but the use of the word *exclusive* seems clearer.

i.e. for the exclusive operation to begin, no other operation may be in progress; no other operation may start whilst an exclusive operation is active.

We may now use this logic to specify the constraints encountered in partitioned collections.

A Logical Specification of Component and Partition Constraints

Consider first the categories of operation encountered in component objects. We shall represent these operations by the following sets:

- sets LS_{ro} and LS_{rw} of operations manipulating local storage in read-only and read-write modes respectively;
- set F of operations performing resolution and forwarding; and
- set ST which are concerned with the fundamental structure of the component object, its creation and deletion.

For a particular component, we may give the constraints on these categories as follows:

$$per(LS_{ro}) \Leftrightarrow active(LS_{rw}) = active(ST) = 0$$
$$per(LS_{rw}) \Leftrightarrow active(LS_{rw}) = active(LS_{ro}) = active(ST) = 0$$
$$per(F) \Leftrightarrow active(ST) = 0$$
$$per(ST) \Leftrightarrow exclusive$$

For a partition object, there need only be two categories of method:

- set R of operations performing resolution; and
- set RA of operations performing (re-)arrangement of elements.

The constraints of these categories are as follows:

$$per(R) \Leftrightarrow active(RA) = 0$$
$$per(RA) \Leftrightarrow exclusive$$

It should be noted that these constraints are free from the possibility of deadlock within a single object. Since permissions apply only to operation's activating, not to their termination, an operation cannot become blocked waiting to finish. Moreover the constraints on start-up have no cycles, so an operation cannot become blocked in a cycle awaiting another operation to start. Therefore, providing that every operation in every set is guaranteed to terminate, deadlock cannot occur.

Since objects and partitions interact, it is also necessary to eliminate the possibility of deadlocks between cycles of objects. This is simple: there is no interaction whatsoever, in terms of concurrency control, between methods in components and in partitions, so deadlock cannot occur between objects. The same is true of operations which pass between partition objects during resolution.

Overview of the Implementation of Concurrency Control

Within the partitioned model, concurrency control is implemented using Arjuna-style auxiliary objects. Any object may create a concurrency controller and use it to ensure synchronisation between its methods.

The concurrency control class implements an interpreter for the deontic logic given above. This allows synchronisation constraints to be expressed directly in the logic, with little or no translation.

Every method in an object having a concurrency controller may be classified into a particular concurrency control class, such as LS_{ro} in components. When called, the method interacts with the concurrency control object to determine whether it may execute. This is a blocking interaction: the method is only unlocked when the controller, after solving the deontic equations, determines the method may be safely started.

Each method also registers its completion with the concurrency controller, which may unblock other methods which are waiting permission to start.

The Implications of Sub-classing

An important facet of the partitioned object model, when used as the basis for a programming environment, is its ability to create novel memory architectures by sub-classing existing collections. Before leaving the topic of concurrency control, we must therefore consider the effects which sub-classing has upon concurrency control.

In creating a sub-class, the programmer may add new state, add new operations, and re-define the meanings of existing operations. This introduces a problem with concurrency control, as any new operations may affect the concurrency control constraints of the existing operations. In DRAGOON, this problem is tackled by outlawing the sub-classing of classes which have a concurrency controller attached (the so-called *behavioured* classes). The effect of this is to completely separate functionality from concurrency control: a class' functionality is first written – possibly by sub-classing an existing class – and a concurrency controller is then written for it.

By placing concurrency control into an instance variable – as in Arjuna, and the current system – this problem is side-stepped. A sub-class will have a concurrency controller created for it by its parent. It may then either use this controller "as-is" or may delete it and substitute another which better fits its needs. The only constraint is that all the sets of operations provided by the original controller are provided in the new controller: presumably the new controller will also maintain the semantics of the original object, but may extend them as required. Care must be taken if multiple inheritance is used, as a new controller must them mimic the behaviours of all parent classes – a task which may be impossible if the parents have conflicting requirements.

Another important facet of the partitioned model is the decoupling of user-defined functionality from the basic functions which manage the components' internal storage structures. It is possible in many circumstances that a sub-classed collection will provide new operations built from the primitive functions: but if the sub-class provides no new state, it will require no additional concurrency control

since it utilises the functions which are provided in the basic architectures, and these are fully protected.

4.3. Concurrency Regulation

Concurrency regulation is a problem addressed must less frequently than concurrency control. The reason for this is simply that it is a more recent problem: it is only with the rise of highly parallel systems that concurrency regulation has become a major problem.

4.3.1. Approaches to Concurrency Regulation

The most common approaches to concurrency regulation are tightly integrated with the common programming practices for parallel systems. A program is typically written to solve a particular problem as quickly as possible, using a particular target machine. The program will be given complete control of the machine, possibly with a rudimentary operating system harness. The configuration of the machine – the number of processors, their topology and local memory sizes – are known in advance.

Concurrency regulation in these cases involves creating the optimal number and distribution of processes on the target machine: usually one process per processor, with processes distributed so as to minimise communication delays. Since the application has sole use of the machine, it may place processes according to its own best interests.

The allocation of processes to processors is known as *configuration*. It usually occurs after the application has been compiled but prior to link-time.

The `PLACED PAR` construct of Occam is probably the most basic configuration system. It allows the elements of a `PAR` statement to be placed on a particular processor. Occam channels may then be mapped onto particular Transputer links. In principle, processes may be placed on whatever processor is most suitable: in practice (with current Occam implementations on current Transputers) configuration is complicated by the fact that only a single channel may be mapped onto any one link, so programmers must manually multiplex the use of links. This makes configuration a very difficult task.

A further problem with this form of configuration is that it is not really separate from the code of the application. A `PLACED PAR` may only place a `PAR` which has already been written and is known to the application: it may not be used to control the replication of processes (for example) independently of the text of the application, which must explicitly create all processes. Therefore configuration and design are tightly coupled, one affecting the other[15].

A better solution is to adopt a "building block" approach in which a set of "black box" processes are connected to form an application. The processes themselves

[15]There is a "replicated PLACED PAR" construct in Occam, but there are restrictions on it to force the amount of replication always to be constant and defined at compile-time. In general it is not possible to create processes "on the fly" in Occam.

present an abstract interface – a set of input and a set of output channels, for example – and the channels are connected by the configuration language. One great advantage is that it allows a library of useful processes to be created and included into any suitable applications. Constructing a new application involves constructing new "black boxes" for processes which are not in the library, and then linking them together to form a network. The use of configuration languages still leads to largely static configurations of processes, although the processes' locations may be dynamically determined.

The Helios shell uses this form of configuration. As mentioned in chapter 1, Helios is a Unix look-alike running on Transputer systems. Each command – **ls**, **grep**, and the rest of the Unix tools – is treated as a process having (usually) one input and one output channel, corresponding to the **stdin** and **stdout** file streams. A shell command of the form

```
cat file.txt | grep "Helios" | less
```

is configured so that (if possible) each element of the pipe executes on a different, though neighbouring, processor. This allows the pipeline to execute in true concurrency. There is also an external configuration language, CDL, allows more complex (though static) networks of processes to be created.

In a more sophisticated form, this style of configuration is also adopted by the Darwin configuration language (§1.3.4).

4.3.2. Regulating Concurrency in a Scalable Environment

Configuration suffers from a crippling handicap for the current purposes: an application is always configured prior to run-time, and so is in a static configuration when it is actually run on the machine. In a scalable system, this presents two problems.

Firstly, the number, topology and size of processors may be unknown prior to run-time. A scalable application must be written in such a way that it may take advantage of whatever resources are available, without intervention, re-compilation or re-configuration. This means that decisions on how processes are mapped to processors must be deferred until the program begins execution. In systems where similar processes are replicated to obtain parallelism – process farms and multiple-worker systems being the most common – it is impossible *a priori* to determine the optimal number of processes to create.

Secondly, the assumption that an application has sole use of the machine is breached when multi-user systems are introduced. On start-up an applications is competing for resources with all other applications in the system. This makes the selection of an "optimal" process distribution impossible ahead of time.

A tightly-coupled configuration system forces all decisions on distribution and replication to be taken when the code of the application is written; configuration languages allow them to be deferred until just prior to run-time. For a scalable system, we require that the decisions are taken *during* the program's execution, for greatest flexibility.

4.3.3. Concurrency Regulation in the Partitioned Model

In §2.4 it was observed that concurrency paradigms may be divided into two categories: data-based and stream-based. It was further noted that scalable memory allows data-based algorithms to be constructed and regulated automatically, as the size of a memory may be used to determine the amount of concurrency used to process it. We shall now expand on these ideas, and consider the ways in which the regulation of concurrency interacts with the creation and management of scalable memories.

Multiple-worker Concurrency

The use of scalable memory suggests the use of concurrency structures which are based around access to a large shared memory. Such access might occur in two ways:

- by a number of functionally specialised processes accessing a memory; or
- by a number of replicas of the same process accessing memory, each performing the same function on different elements.

In some senses the second case is subsumed by the first: a single logical process might be composed of a number of replicas, so several functionally distinct process groups might access a single memory concurrently.

The use of replicated processes in this way is often called the *multiple worker* paradigm. Each process is a "worker" performing part of a larger task. Each worker is assigned a part of the data to be processed, with workers taking disjoint data sets which together cover the complete data set. The workers are run concurrently, with the activity finishing when all its workers have finished processing their part of the data set.

In order for this paradigm to function correctly, there must be some means of determining how many worker processes to use, and of dividing up the data between them. There are hence two complementary aims: assisting the programmer in constructing suitable processes (which may process correctly data sets whose exact bounds are unknown) and allowing the system to replicate and locate them correctly.

Activities and Attachment

The general form of a scalable memory (or partitioned collection) was shown in figure 7. The collection is by its very nature divided into a number of smaller parts – the components – and such sub-division is exactly what is required for multiple worker concurrency. If an activity is assigned to process each component, then together the activities will process the entire collection. Moreover, they will access data through the distribution-transparent interface of the component: this means that

they may access any data item within the collection regardless of its location if required to perform their task.

Specifying Activities

Concurrency in object-oriented systems occurs at the granularity of the method call (§4.1). This level is, however, wholly unsuitable for constructing parallel applications, being to parallel programs what the `goto` statement is to sequential programs: powerful, but completely unstructured. Just as "structured programming" evolved to meet the demands of large-scale sequential systems, it is essential to provide support for the introduction of parallelism in a controlled way.

Although it would in principle be possible to use any object as a worker in a multiple worker system, a better solution is to provide some support, in the form of a suitable protocol, for the construction of worker processes. Such worker objects may be sub-classed to provide the necessary specific functionality while still being guaranteed to provide the functions needed by the system. We shall term this category of objects *activities*.

An activity has four important attributes, providing

- a method of replication;
- a way of "attaching" it to a component;
- a way of obtaining the elements which it is to process; and
- a means of supplying task-specific functions in a manner well-integrated with the preceding two points.

The replication method is used to create as many replicas of the process as required. Attachment involves informing each replicated activity exactly which component of a partitioned collection it is to process (with which it will be closely co-located). Once attached, the activity must be able to obtain the elements assigned to it – the locally-held elements of the component – either by knowing their names or by iteration. Finally, there must be a well-defined method for adding the task-specific functions required.

Attaching Activities

From the programmer's view, attaching activities to a memory is a simple task: the operation may be encapsulated into a method call. An application can call the method, supplying the activity to be run as a parameter. This activity is then replicated near each component of the collection, with each replica being attached to its assigned component. They may then execute concurrently: the method which initiates this activity may wish to await the termination of all the activities, or may continue without waiting.

Internally, attachment involves traversing the partition tree. At each leaf (component) of the tree, a copy of the supplied activity should be created. This process is shown in figure 20.

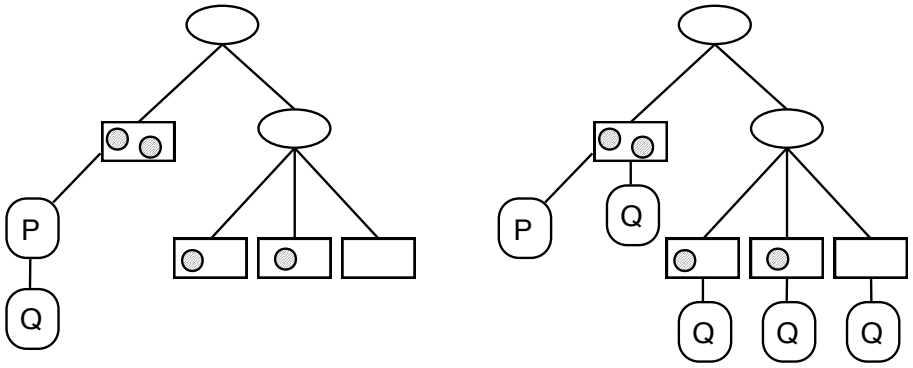


Figure 20: Attaching activities to a collection

The attaching algorithm is simply a traverse of the partition tree. One might write this algorithm as a parallel process: in practice, except for extremely large collections, the sequential version of the algorithm is sufficient. The only constraint on the algorithm is that replicas of activities should be created as close as possible to the component to which they are to be attached: this minimises communication delays, and may be performed transparently of the programmer according to the distribution of the collection.

Processing-Memory Interactions

On the surface, the interaction between activities and memory is simple: activities are attached according to the distribution of the partitioned memory, and run on the processors over which the collection is distributed. There is a more subtle issues however, in deciding the optimum granularity for dividing the collection,

The division of a collection into components, by whatever manner, serves two purposes: it distributes the data of the collection, allowing elements to reside on different processors; and *via* this distribution it provides a mechanism by which concurrency may be regulated. The distribution of concurrent activity, and the granularity at which it occurs, follows the distribution of data elements.

This scheme has a lot to recommend it. When considering large sets of data being processed in parallel, the size of the data set is often a good measure of the complexity of the problem. If such a data set is represented as a single partitioned collection, then the size of the collection – in terms of the number of components – is controlled by the size of the original data set: the larger the data set, the more components the collection will have, and the more processing nodes it will use. Similarly, the more components a collection has, the more activities will be created when an activity is assigned to the collection, so a larger data set will generate more

concurrent activity. Hence the size of the problem is the single factor controlling distribution and parallelism in a scalable application.

However, determining the size of individual components may be a difficult task. By way of illustration, consider two problems of identical size, each composed of a number of integers, and each consisting of a function applied to each integer in the collection. The function is encapsulated into an activity. The first problem's function is a simple factorial calculation; the second is a more complex cellular automaton which must access neighbouring values in order to compute its result.

In the first case, the activity will access only local data elements, so the more components there are in the collection the more activities will run concurrently, and the faster it will execute. In the second case, however, activities must access other, potentially remote, elements, and this introduces remote requests when elements at the "edge" of a component are processed. Although increasing the number of components will increase the amount of parallelism, the reduced component size will mean that proportionally more requests will be remote due to edge effects. Concurrency control is performed *en bloc* over a component, so a large component introduces more synchronisation. There is therefore a complex balance to be struck between two factors.

In both cases, we have ignored the set-up times for both the collection and the activities. Larger collections will take longer to create than smaller ones; larger numbers of activities will similarly take more time to create than smaller numbers. Here is another trade-off[16].

Ideally these trade-offs would be resolved automatically: in practice, this involves automatically determining the computational weight of a given piece of code, which is equivalent to the halting problem; moreover, the piece of code to be executed may not be known at compile-time.

There is a partial solution to hand, however. The model of scalable memory does not mandate any particular distribution grain size, whilst the concurrency regulation scheme can handle any grain size. If some means exists to specify the grain size at run-time, without re-compiling or in any other way altering the compiled application, the performance of the application – as determined by the granularity of distribution – may be optimised on an exploratory basis.

The grain size may be treated as a property of a scalable memory, retrieved from the property database mentioned in chapter 3: grain size is simply another property, like the degree of the partition tree or the enabling of caching.

Other Concurrent Forms

We mentioned in chapter 2 that, whilst the multiple worker form of concurrency seemed best suited to scalable processing, there were several problem domains for which the model seems inappropriate. We shall now consider two of these forms – pipelines and process farms – and how they relate to scalable memory.

---

[16]It should be noted in passing that most authors ignore set-up time when presenting results of experiments in parallel computing, which are always assumed to be small compared to the computation time and is all incurred before any "real" processing occurs. In scalable system this is not the case, as the distribution of processes and data may change dynamically.

Pipelines

Pipelines, as shown in figure 3, are formed when a number of functional units are connected by channels, down which they pass data. Each stage of the pipeline accepts values from its predecessor, transforms them in some way, and passes them to its successor. Stages of the pipeline may be replicated to improve throughput. Since each stage of a pipeline is functionally different, there is not necessarily a correlation between the number of stages in a pipeline and the amount of data to be processed.

However, memory is used in two places: as the source of data into the pipeline and as a sink for the values produced. The first stage of the pipeline is responsible for removing data from the source and feeding them into the pipeline; the last stage performs the opposite function. Of course, stages in the pipeline may have local storage, and may interact with other memory modules like any other object.

In many cases, it may be possible to replicate the pipeline. If the first stage is an activity, it may be attached to the source and replicated; it may then create the rest of the stages of the pipeline. The final stage should be passed a handle to the sink collection, in which the final data is to be stored.

Handling replication within a stage is a problem which cannot directly be tackled by the partitioned model, although it is possible to express this construction quite simply.

Farms

A process farm is a variation on the multiple worker idea with an important difference. In a multiple worker system, the work which each worker is to perform is assigned initially; for a farm, each member requests an new piece of work whenever it is free, until no more elements of work exist.

It is simple to implement a process farm using the partitioned model: one may in fact use the same attachment and regulation mechanism as with the multiple worker paradigm. Rather than each process computing with the elements of its component, however, a slight variation is required: the workers must iterate through the collection as a whole.

Each worker could begin processing the local elements. If these elements are exhausted, however, the worker should attempt to obtain another, remote element for processing. This implies that the work which in a multiple-worker system would be assigned to one process is in the farm performed by another process which would otherwise be idle.

Implementing a process farm requires that there is some means of determining those elements of a collection which have been processed. This could be provided by simply tagging each element, and marking its tag when some process deals with it. A request protocol is then required which will locate some unprocessed element: in the beginning, these elements should come from the component receiving the request, and will only access remote components when the local pool of work is exhausted. When stated like this, it is evident that process farms may be implemented within the partitioned model by sub-classing the existing storage structures.

Processes Within Collections

There is another possibility for creating process structures: placing the objects which compose the structure into a scalable memory, and using the memory's operations to co-ordinate the processes' actions.

This is a radical departure from the concurrency structures considered so far. Rather than memory being a repository for data to be used by processes, it becomes a repository for the processes themselves: processes become special forms of data to be stored in memory. It is hardly a surprising suggestion – the Von Neumann model keeps code, data and process tables in memory (separated to a greater or lesser extent) – and is closely akin to the notion of an "active tuple" in Linda systems. However, the partitioned model offers considerable benefits of this form of process structuring is adopted.

Firstly, all the benefits of the model – distribution transparency, type safety, high-level interfaces, customisable distributions *et alia* – may be used to control process structures. This means, for example, that an array of processes – a cellular automaton – may be built in which the processes communicate using the array protocol: by position instead of by process name. Another example is processes placed into an associative store, from which they may be retrieved by a complex name: this might be used as a "yellow pages" name server. A third possibility is the emulation of CSP's channels by using a directed storage architecture with processes as the nodes.

In principle, this form of structuring is very attractive. It restores the full equality of activities as data elements whilst allowing them to be manipulated in large abstract collections in the same framework as other objects. The extent to which this would affect programming practice is still a matter for investigation. We shall return to consider it further in chapter 7.

4.4. Résumé

In this chapter we considered two facets of concurrency within scalable systems: concurrency control and concurrency regulation. We examined the ways in which they interact with each other and with scalable memory, and have developed strategies for dealing with them.

Some examples of concurrency control in object-oriented systems were summarised and compared. From them, a system was synthesised which allowed flexible concurrency control protocols to be constructed in a logical manner. The sorts of concurrency encountered in components and partitions was discussed, and from them a small number of primitive sets of operations was derived. These sets constitute the only operations which access the internal structures of the partitioned collections. Constraints required to ensure that they execute correctly in a parallel environment were then derived.

The problem of regulating concurrency in a scalable system was considered. A common paradigm for concurrency regulation – the multiple worker paradigm – was examined in detail, and the ways in which the number of workers could be regulated automatically using scalable memory were presented. This illuminated some trade-

offs between distribution and parallelism in this form of system. Other process structures were considered more briefly, showing that the partitioned model can support a range of parallel architectures within its framework.

Chapter 5.

Phœnix: a Prototype Environment

"Well, can I walk beside you?
I have come here to lose the smog,
And I feel just like a cog in something turnin'.
Well maybe it's the time or year,
Or maybe it's the time of man,
And I don't know who I am, but life's for learnin'."

Joni Mitchell, *Woodstock*

Programming is a human activity. In advocating any new or improved programming method or tool, one must evaluate how the new item relates to programming as practised by programmers, and not simply evaluate the system's purely technical merits.

An evaluation of the partitioned object model may be conducted from its theoretical description, but in order to assess its true usefulness some practical experience is also necessary. For this reason a prototype programming environment, based on the partitioned model, has been constructed. The prototype – called *Phœnix* after the legendary keeper of Wisdom – is not intended as a production-quality piece of software, nor as a practical programming platform. Its goal is to provide a means of exploring two questions:

- to what extent is it possible to *practically* hide the problems of distribution and concurrency within a scalable environment? and
- in what manner can this abstraction be used to encourage the design and re-use of designs and implementations?

These are problems which cannot adequately be tackled without an implementation, however primitive.

5.1. The Structure of Phœnix

As with any large piece of software, Phœnix is not monolithic: rather, it is composed of a set of sub-libraries, or "kits," each of which control one aspect of the toolkit's operation. The layered structure of the Phœnix kits is shown in figure 21.

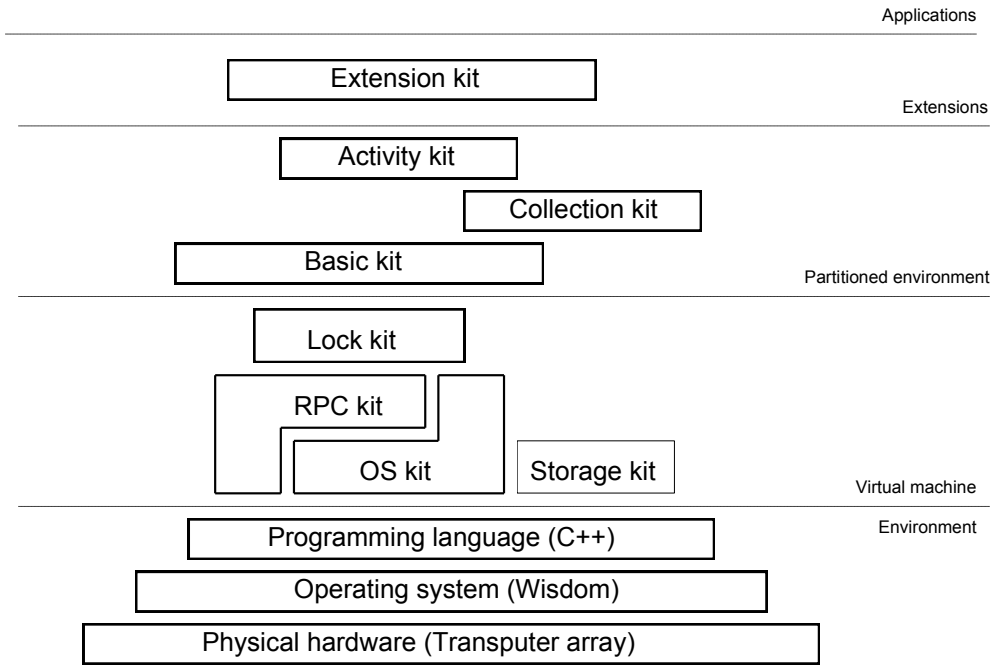


Figure 21: The Structure of Phœnix

Phœnix is composed of four layers: environment, virtual machine, partitioned environment and extensions. Each layer provides a new abstract machine for the layers above, in the manner described in chapter 2.

The environment layer includes the hardware and operating system platform on which Phœnix is executing, together with the host programming language used to create Phœnix applications.

The virtual machine layer provides the abstraction of a virtual object space. The OS kit presents a high-level interface to the process control and message transport functions of the operating system, while the RPC kit is used to implement remote procedure call[18][94] outside the host compiler. The Storage kit manages memory in a typeless manner; the Lock kit implements concurrency controllers of various varieties.

The partitioned environment layer provides the abstraction of scalable memory modules having various storage architectures. The Basic kit contains classes of general use, and those classes which are used internally by partitioned collections; the collection kit contains the basic components and distribution managers for the supplied storage architectures; the Activity kit contains the support necessary to provide scalable processing capabilities using multiple worker tasks.

The extension layer contains a small number of partitioned collections which are considered to be particularly useful: an array of real numbers, a dictionary and a binary tree. These are built on top of the partitioned environment classes using sub-classing. This layer allows applications (or additional layers) to work with strongly-typed scalable memories.

Each layer of Phœnix thus provides an essentially new memory abstraction to the layers above: from distributed memory to object-oriented memory to scalable memory to typed and extensible scalable memory. There are also changes in abstraction with regard to concurrency, beginning with manual concurrency control and regulation and finishing with an object-oriented per-method concurrency control system coupled with automatic concurrency regulation based around typed scalable memory.

5.2. The Host Language and Environment

We shall begin by describing the host language and environment used by the Phœnix prototype.

5.2.1. Design Issues

In designing a suitable prototyping platform, two main decisions must be made: on what hardware is the system to be implemented, and in what language.

The partitioned model is targeted at scalable, highly parallel, tightly-coupled multicomputer systems. Most of the currently-available technology in this area is based around the Inmos Transputer which, although deficient in many respects, makes an adequate testbed for experiments with scalable systems. A small network of Transputers was made available for experiments. On top of this system was running the Wisdom scalable operating system nucleus (§1.2.2 and Appendix B).

One effect of using Wisdom was that it provides a load balancing module, which may be modified to work on a per-task basis. The module follows the "ink blot" style, and thus respects the assumptions of the partitioned model with regard to object locations (§3.2.5).

The choice of a host language was restricted to those object-oriented languages which were readily available to run on a Wisdom system, for speed of implementation. Although it might have been preferable to implement a new language from scratch, it was felt that a prototype implementation in an existing language was more suitable for the project.

The language selected was C++, running a translator which converted C++ into C. This has the advantage of portability, as the C++ translator is available for use on any target system which supports a C compiler.

5.2.2. The Phœnix Pre-processor

Pure C++, however, does not support certain features which are essential for Phœnix: it lacks support for the implementation of a virtual object space and for the introduction of parallelism.

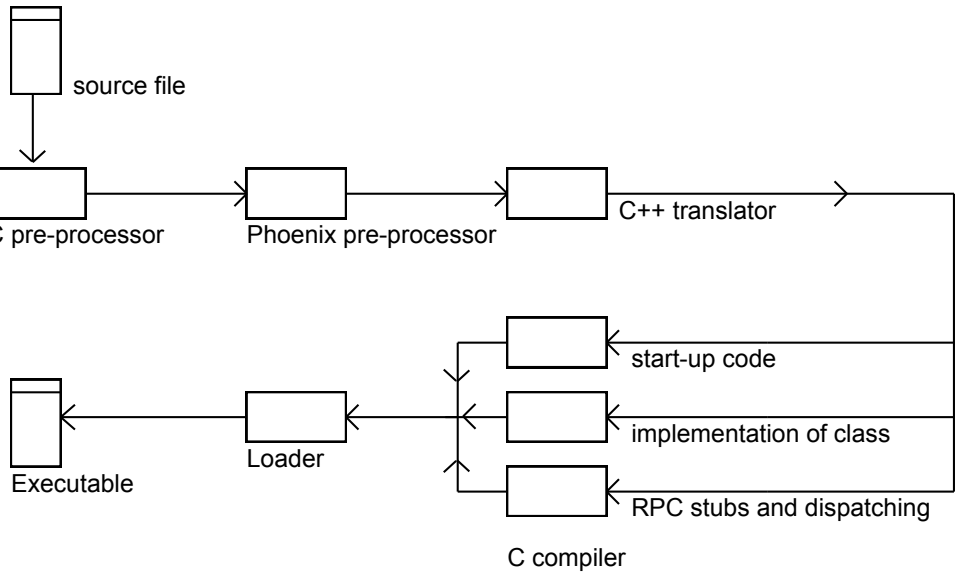


Figure 22: The compilation process

For this reason, another pre-processor was implemented to translate a minimally enhanced dialect of C++ into the pure language suitable for the C++ translator. This lead to the five-stage-pipeline compiler architecture shown in figure 22.

The pre-processor follows the style of the standard C pre-processor, translating "directives" dealing with the creation of objects in a distributed environment, their interaction and concurrency, into C++.

Introducing Distribution

To build a distributed object-oriented system is to abstract away from the physical location of objects and instead to allow any pair of objects to interact regardless of their relative locations. This is achieved by creating a *virtual object space* in which the name of an object is sufficient to identify it uniquely within the system. Using this name, together with an appropriate support structure, objects may interact *via* method calls even when they reside on different processors.

Implementing a virtual object space within C++ is a difficult task. The problem centres around a design decision, taken deep within the C++ language, that an object

name is a pointer to an area of memory. So deeply is this assumption buried that it is impossible to change objects names to be some other, more complex structure without fatally damaging the language's semantics. It is, for example, perfectly legal – though not encouraged – in C++ to perform C-style pointer arithmetic on object names[17].

Two possible solutions suggest themselves. The first is to keep object names as pointers into some intermediate object table, such as is found in most Smalltalk systems. In this table could be kept the information necessary to identify the object being referenced. The other possibility is to introduce some new object naming system outside the syntax of C++, and convert it into "pure" C++ using a pre-processor.

The first solution, while attractive, leads to a rather problematic implementation due to the way in which classes are represented. The second solution, however, is easy to implement. The pre-processor converts a slightly embellished dialect of C++ into the "pure" language, with the embellishments supplying enough information to generate the necessary remote method invocation code. The disadvantage is that a program is no longer strictly speaking a C++ program, and must be transformed by another compiler pass.

This solution was adopted for Phœnix. Object names are represented by the abstract "handle" type, which may be stored in variables, compared and passed as parameters without restriction.

Method calls are made using statements interpreted by the pre-processor. A call takes the form

```
#Call(class, res = handle -> method ( p1, p2, ... ) )
```

which is expanded into "pure" C++ by the pre-processor.

Introducing Concurrency

The model of concurrency favoured for the partitioned model was presented in chapter 4: methods in a class may be defined as being called asynchronously, whereupon the caller does not block and does not expect a value to be returned.

C++ provides no mechanism for introducing new class syntax: it is not possible to embellish the properties of methods. Therefore there is no way, within the language, to specify that a method is to be executed asynchronously. (This is one reason which many concurrent C++ implementations adopt a model of concurrency based around objects having special properties.)

The choice of a pre-processor, as described above, now comes to our aid. By adding suitable pre-processor constructs, we may maintain sufficient information about a class' definition to specify that some of its methods be executed asynchronously. This information may be used by the method call statement to create the correct calling code transparently The programmer may call any method

[17]If C++ is implemented on a machine with a segmented architecture, pointer arithmetic fails anyway when a segment boundary is crossed. This makes the choice of object names as pointers, *de rigueur*, even more questionable.

using the `#Call` directive, whereupon the pre-processor will automatically determine whether the method is asynchronous or not and generate the correct code. The same declarations may also be used to generate RPC stubs.

To obtain the information about a class' methods, the simplest solution is to provide a "parallel" definition for each class specifying the type signature and calling convention of each method[18].

Restrictions

It should be apparent that, in terms of language design, the Phœnix C++ compiler leaves much to be desired. No apologies are made for this: the host language is purely a vehicle for experiment, and such needs only be as finished as necessary for the desired experimentation. However, it is worth pointing out exactly which features of the host language would require work in a "real" system.

The first point is the use of a pre-processor to expand method calls. The statements used are completely outside the scope of C++, and do not integrate well with it. Furthermore they are outside C++'s type system, and so cannot be guaranteed by the compiler to be type-correct: in particular, the "handle" type used to name objects does not preserve their type, so the programmer must ensure that all operations are applied to objects of suitable type. Since "type-correct" is almost a synonym for "sensible[69]," the programmer must take on through discipline much of what, in a real language, would be enforced by the type system.

A related problem is the use of parallel class descriptions which the programmer must ensure are consistent. In practice, inconsistencies are caught by the compiler – though not in the place which might be expected. This consistency checking is an additional burden which, again, would be carried out by the compiler in a real language.

The use of the pre-processor adds another pass to the compilation process. In expanding definitions into "raw" C++, it also increases the amount of compilation and (more importantly) linking required to create an application. This leads to significantly increased compile-edit-debug cycles.

5.3. The Virtual Machine Layer

The purpose of the virtual machine layer is to act as an interface between the host language and the higher-level components of Phœnix. It provides four services:

- process creation;
- remote method calls;
- concurrency control; and
- local storage management

[18]It would be more attractive to parse the existing C++ class definitions to obtain this information. However, due to yet another "feature" of C++, parsing class definitions potentially requires a program to be able to parse the *entire* C++ language rather than only a restricted sub-set. For simplicity of implementation, *and for no other reason*, parallel definitions are to be preferred.

These services are all intimately related to the operating system being used at the environment layer: their encapsulation means that the operating system may be changed with minimal impact on the rest of the Phœnix system.

5.3.1. Design Issues

Many of the features of the virtual machine layer are mandated by the object model chosen for Phœnix. Other features are constrained by the capabilities of the operating system and host language used.

The chief design decision concerns the way in which objects themselves are represented. This decision was taken in the host language: each class is represented by a stand-alone executable file which manages all objects of that class on a single node. Class servers are created and located as required.

The host itself, however, does not implement facilities such as links to the operating system or object interactions itself: rather, it generates calls to the objects which are defined in the virtual machine layer to perform these tasks. The layer may thus be regarded as the Phœnix "run-time library."

The basic structure of a Phœnix application is a set of stand-alone class servers. Each server manages objects of a single class, providing the interface between objects and the other objects in the virtual object space. The actual interactions are performed by the virtual machine layer.

5.3.2. Implementation Overview

We shall give only a brief overview of the implementation of the virtual machine layer, before moving on to the implementation of the more novel parts of the Phœnix toolkit.

Managing the Virtual Object Space

The primary function of the layer is to manage a virtual object space containing all the objects composing the applications in the system. The space must allow objects to be named and to interact regardless of their relative locations.

The Guardian

The basic object of a class server is an instance of the `Guardian` class. Every class server has exactly one Guardian, which acts as a gateway to the virtual object space.

Guardians perform several functions: they

- implement the start-up and close-down code needed by class servers;
- manage several pools of frequently-used resources; and
- provide a remote procedure call service (see later);
- co-operate to ensure that class servers are created as needed;

- shield the rest of Phœnix from the details of message transfer, process creation and other low-level operating system tasks.

When a class server is first created, the Guardian executes the start-up code necessary for the correct working of Phœnix. This includes registering the new server in the system's name space (along with its location), and the creation of the resource pools.

Two pools are managed by Guardians: a pool of RPC packets and a pool of processes. The former are used whenever a method is called on another object, or a reply to a method call is sent from an object managed by the Guardian; the latter are used to service incoming method calls. The sizes of both pools may be controlled at run-time.

The remote procedure call service is used internally by Phœnix when making method calls on remote objects. It is dealt with fully in the next section.

Phœnix takes steps to ensure that only a single class server for a particular class is running on any one node – although in general there will be several class servers for a class in the system, each running on a different node. The Guardian provides a set of functions which allow the RPC service to determine whether, when creating an object on a new node, it is necessary also to create a class server. Each class server registers its name and location in the system name space, along with a capability down which it may be accessed.

In general the programmer never interacts with the Guardian, as the Phœnix pre-processor and support classes provide higher-level abstractions. For example, the process-creation functions are perceived as asynchronous method calls.

When started, the Guardian begins to run an event loop monitoring the capability which it registered with the namer. All requests for service – remote method calls, the creation and deletion of objects – occur on this capability. The event loop is very simple: when a request is received on the capability it acquires a server process from the process pool and assigns it to process the request; it then awaits the next request. Although the capability forms a single point of communication (and hence is a potential bottleneck), the simple nature of the event loop minimises the chances of serious delays at this point.

A server process, when assigned a request, performs all the actions required by that request and then returns itself to the pool. Typically a request will animate a method call in an object managed by the Guardian.

Remote Method Call

Phœnix objects interact using method calls, which are dispatched across the network by the underlying virtual object space management system. Methods are implemented using remote procedure call[18][94] (RPC).

Objects make calls to other objects using the pre-processor `#Call` directive. These are expanded to:

- obtain an RPC request packet;
- add the necessary routing information;
- marshal all the parameters to the call;

- for synchronously-called methods, create a capability down which the return value will be sent;
- interact with the Guardian to send the request;
- for synchronously-called methods, wait until the result of the method is received;
- tidy up, returning the packet to the Guardian's pool.

A single method call is an exchange of datagrams. The Guardian is responsible to sending the request packet to the correct site, but this packet contains a capability to which the method's return value will be sent. This means that the caller's Guardian is not involved in the reply: the receiver transmits the reply directly to the calling process.

A method call is assigned a server process when it is accepted by the Guardian which is managing the target object. This process identifies the object which should receive the method, unpacks the parameters and makes the method call. The process will execute until the called method terminates, and will be responsible for returning any return values to the caller. For asynchronous methods, no such values are returned: the difference between a synchronous and an asynchronous method call is all at the caller's end of the RPC system, not at the receiver's.

A variation on the simple RPC request involves the creation and deletion of objects. The former is simply a call to one of the class constructors, and may be treated like a synchronous method call whose return value is the handle to the newly-created object; object destruction is a synchronous call with no parameters and no return value.

Although all method calls are logically remote, a simple optimisation occurs if the targeted object is on the same node as the caller. The Guardian performs a "short-circuit" to avoid invoking the kernel to pass messages, simply calling the necessary routines in the target object's Guardian directly. (This particular short-cut only works because, on the Transputer, all tasks share a common address space. In a processor with virtual address spaces, this direct-call strategy would be impossible.)

Object Naming

An issue not yet touched upon is the manner in which objects are named.

Virtual object space names are perforce more complicated that those for simple shared-memory systems. An object's name must uniquely identify it, and must also allow method calls to be routed to it from any part of the system.

The solution adopted for Phœnix is that, in the absence of object migration, the Guardian managing an object is fixed at its creation: therefore a name which identifies the Guardian and also the object within that Guardian's management domain will serve to identify all objects uniquely. A Phœnix object name – also called a *handle* – is composed of the capability which the Guardian registers with the system namer and the address of the object within the Guardian's address space.

This scheme has the advantage that an object name contains all the information needed to route messages to the object directly, since all RPC traffic occurs through the Guardian's capability; furthermore, it contains all the information which the Guardian needs internally.

The major disadvantages of the scheme are that it makes object migration difficult (though not impossible, if a scheme similar to Emerald's forwarding addresses is used), and it makes object names rather large since capabilities are usually quite large objects (in Wisdom, a capability is around forty bytes).

Concurrency Control

Concurrency control is the province of the Lock kit, which provides a set of classes implementing the concurrency control model described in §4.1. A lock is an object which embodies a concurrency control policy. The functions which the lock exports are then used by objects to ensure that method calls are "safe" according to that policy.

Phœnix defines locks in the abstract as instances of the Lock class. This class exports a simple interface: methods call the lock whenever they wish to execute, and this call only returns when the lock determines that the execution is safe. Methods also call the lock when they complete execution, so as to allow other methods to run.

A particular sub-class of the general `Lock` is the `DLock`, which implements an evaluator for the deontic logic presented in §4.2.2. Sub-classes of `DLock` may then use the logic to define their concurrency control policies. The important special cases of single writer/multiple reader and the lock used in the partitioned collections themselves are pre-defined, and may be re-used by any class in Phœnix.

Low-level Storage Management

Although the partitioned model provides a model of memory as seen by applications, another system is needed to manage memory at the most primitive level.

Phœnix provides two classes for dealing with storage in this manner. Both manage storage as collections of fixed-sized, untyped values: one implements a vector of elements accessed using an index, whilst the other implements a variable-length list of elements. Both are sub-types of the `Storage` class.

The memory management classes all manage memory in untyped units called `StorageElements`. A `StorageElement` may be placed into a storage class and later retrieved. The size of each `StorageElement` is fixed at its creation, although different `StorageElements` may have different sizes. They export a completely untyped interface.

Phœnix manages local memory in collections as follows: the elements of a collection are stored in `StorageElements`, which are themselves placed into an instance of a `Storage` class. The storage routines themselves do not manipulate the values which they store, and may thus be made independent of the values' type. The collection classes are responsible for ensuring that values are stored and retrieved in a type-safe manner.

5.4. The Partitioned Environment

The heart of Phœnix are the classes which directly support the partitioned model. Phœnix provides a collection of partitioned data structures which are designed to as to be completely distributed, scalable, highly concurrent and simple to extend through sub-classing.

5.4.1. Design Issues

The Phœnix Collection and Partition class hierarchies directly implement the functions described in §3. There are, however, some subtle differences introduced by the choice of host language. The two main differences concern the way in which resolution is performed, and the creation and deletion of components of collections.

Resolution

Resolution (§3.2.4) involves forwarding requests for data *in toto* between components. A component which receives a request which it cannot service locally forwards it *via* its partition to another component which can deal with the request. In essence, the continuation of the request is passed to another site.

This architecture is very attractive for a number of reasons: it is simple, logical, and means that a site receiving a request may simply forward it and continue processing, thus avoiding a potential bottleneck. It proved rather difficult to implement for Phœnix, however, for two reasons: the nature of C++ procedures and the use of RPC.

C++ internally treats function names as pointers. Once a function has been called, therefore, all other information disappears. It is not possible in C++ – as it is in Smalltalk – to obtain the text of the request being serviced. This makes forwarding of requests difficult.

A second difficulty is the use of RPC, as the client blocks awaiting a reply from the called object. Passing a request involves changing the object which will reply to a request. (Actually this is not as difficult as it sounds, as Wisdom allows capabilities – return addresses – to be passed to another process. Whilst possible, such an architecture would require a "pass responsibility" primitive, increasing the complexity of the programming model.)

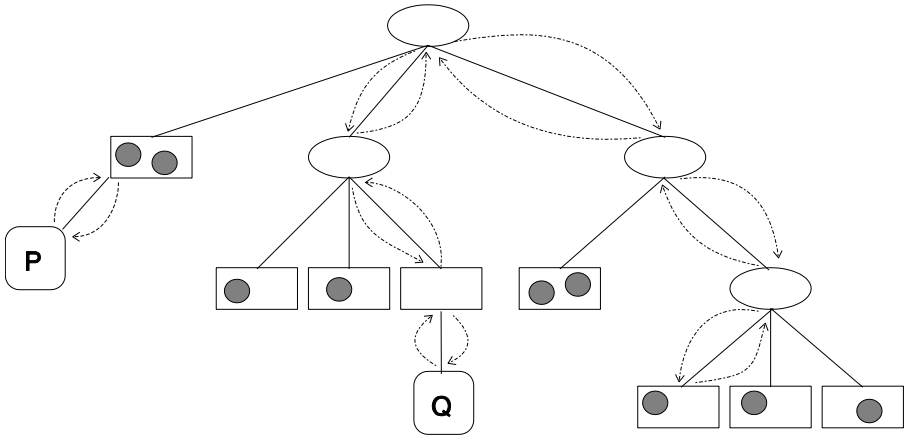


Figure 23: Control flow for resolution in Phœnix

The solution adopted for Phœnix is to implement resolution using simple RPC. The difference in control flow is illustrated in figure 23. It is interesting to compare this diagram with that in figure 8: the "circular" flow of control is replaced by a "chained" flow. This has several disadvantages. The first is that there is an amount of unnecessary message-passing occurring: the partitions which were involved in resolution have no interest in returning messages, and act simply as forwarders. This introduces overheads. Secondly, such circularities may introduce unnecessary synchronisation. They are accepted here purely for the sake of a simple implementation: we shall discuss possible improvements in chapter 7.

Re-arrangement, and the Deletion of Components

The second major problem concerns the aspects of the partitioned model which involve the deletion of components, specifically the *join* operation in associative and directed storage.

Suppose that an application creates a dictionary (associative memory) and attaches an activity to it, so that a copy of the activity is attached to every component. In the course of processing, a replica removes an element from the dictionary, which causes the dictionary to join several components into a single component. This operation is perfectly well-defined within the partitioned model, and is the reverse of the *split* operation: the elements held by one or more components are amalgamated into a single component, and the extraneous components are then deleted.

It is this deletion which causes the problem: there will be activities attached to these components, and their deletion will result in "dangling" pointers. Even if deletion operations use some form of reference counting, the components will be detached from the collection.

Another variation of this problem occurs in the opposite case – when elements are added which cause splits. The new components will then have no activities attached to them, and it is not safe to attach activities as this may result in elements being processed twice.

The general statement of this problem is that *it may not be safe to perform re-arrangement of data structures during processing*. This is an annoying, but not crippling restriction, and Phœnix does not enforce it. It only affects those structures which may suffer re-arrangement (*i.e.* anything except arrays); indeed, it may be safe to re-arrange such structures even during processing, depending upon exactly what application is running.

A similar problem affects iteration: iteration across a structure is not completely deterministic in the presence of possible re-arrangements. Iterating through a structure whilst simultaneously adding elements to it (or deleting elements) may result in the iteration omitting some elements of the structure, or returning duplicates of some elements.

5.4.2. Basic Classes

The Phœnix Basic kit implements classes which are used directly or indirectly by several parts of the system.

Common Functionality of all Phœnix Objects

C++, unlike Smalltalk, does not force its class definitions to form a tree: they may form a forest of trees[19]. It is useful, however, for all objects to have a certain minimum functionality, and this is best accomplished by having a single base class for all objects.

Phœnix defines a class `Object` for this purpose. It provides the following guaranteed functions for each objects:

- class naming
- instance naming
- equality comparison
- copying
- hashing
- concurrency control
- property access
- error handling, logging and debugging messages

The class name allows an object to access its type at run-time. Each object may have an instance name to distinguish it (at a textual level) from other objects of its class. The equality comparison, copying and hashing protocols are "virtual" (re-defined on a per-class basis, but always with a common type signature): hashing is implemented to support associative memory architectures.

[19]Actually, using multiple inheritance, a forest of directed acyclic graphs.

The concurrency control methods allow a method to register its starting and finishing with the object's concurrency controller (if any) in a manner independent of the actual controller class being used.

Property access will be discussed in more detail later, but is concerned with accessing a database of user-supplied "hints" to control aspects of an application's function.

Error handling allows common error conditions, such as run-time exceptions, to be reported through a standard interface. There is support for logging actions to the user (or to a file) and for accessing a debugging level to control the amount of debugging information generated (without re-compilation).

When created, an `Object` runs through the sequence of constructors defined by its inheritance hierarchy. The result of construction is to create an `Object` whose class and instance names are defined and which has a concurrency controller installed.

Properties

One important aspect of Phœnix yet to be touched upon is the way in which the programmer supplies "hints" to an application at run-time.

In §3.4 it was mentioned that the partitioned model allows decisions about a structure's exact configuration to be deferred until run-time. A distribution strategy – embodied in a particular `Partition` sub-class – may make use of run-time conditions and programmer-supplied hints to determine the exact distribution pattern used.

The judicious use of hints allows programmers to influence markedly the distribution of structures. To take one example, a hint might suggest an order-of-magnitude number of elements to be stored in an array component. The distribution policy may then use this suggestion to decide how many components to create for arrays at their creation.

The hints which may be supplied are obviously very dependent upon the structure being hinted at. By taking care in selecting what features may be controlled by hints, it is possible to generate a very flexible control mechanism for distribution at very little cost. Moreover, it might allow the use of automated tools for optimising distributions.

Phœnix provides a hints mechanism *via* its *property sheet*. A property sheet is an object of class `PropertySheet`, of which exactly one instance exists per application. When created, this object reads hints from a file to create a hints database which may be accessed by any object.

The form of properties is modelled on that of X Windows[103]. Properties are described hierarchically by naming the class hierarchy to an object. For example, the hierarchy to the `Array` class is

```
Object.Collection.ArrayedCollection.Array
```

meaning that `Array` is a sub-class of `ArrayedCollection` which is turn a sub-class of `Collection` and so on. A property specification takes the form of a class path followed by a property name and value, for example

```
Object.Region.verbosity:                    2
```

which sets the verbosity (amount of logging information generated) in all `Region` objects to be 2.

Properties may be set generically for parts of the class hierarchy by using the `*` wildcard. So, for example, to set the verbosity of all collections to 2, one would supply the hint

```
Object.Collection*verbosity:                2
```

Generic properties are overridden by properties given farther down the tree: to disable debugging logging on all collections except `Array`s, the following hints may be used:

```
Object.Collection*verbosity:                0
Object.Collection*Array.verbosity:          2
```

Properties may also be set for particular objets by giving an object an instance name and using it in a hint: adding the property

```
Object.Collection*Array.test.verbosity:     0
```

would disable the logging in an `Array` called `test`. The `Object` class automatically creates the property path name for all objects, and there is support within the base `Object` class to obtain properties.

5.4.3. Collections

The components of partitioned collections are, in Phœnix, all derived from the `Collection` class. Each storage architecture – arrayed, associative and directed – has an associated `Collection` sub-class hierarchy.

The hierarchies are separated into two classes: an abstract class defining the protocols and functions common to all implementations of an architecture and a concrete class implementing a particular local-storage model. Many of the functions in the abstract class are empty, being used to define the type signatures of functions will which be defined later.

The classes do *not* implement any functions intended for application-level access to data. All data access is performed at an untyped level: it is the responsibility of sub-classes to implement appropriate access functions.

Iteration

As an alternative to using access functions, it is possible to iterate across collections. Iteration allows client objects to access all the elements in a component "anonymously," one at a time: it also has the advantage of being a protocol shared by *all* architectures, so a client may iterate across any storage architecture.

All the `Collection` sub-classes multiply-inherit the `Iterator` class[20]. This defines a set of functions which return the first element in a component, and then successive elements until the component has been fully iterated. To allow concurrent access, iteration uses a "key" held by the client object: the iteration protocol itself is stateless, and holds all its context in the key value. Each concrete `Collection` sub-class must provide functions to interpret the key value and return the correct element: this is provided automatically by the built-in concrete classes. There is no pre-defined order for iteration – indeed, there is no single order which is meaningful for all architectures – so sub-classes are free to define their own orderings.

Iteration is not necessarily safe in the presence of concurrent addition or deletion to structures, and this may cause problems in associative and directed collections. The reason is that these operations may cause the collection to be re-arranged, and this may confuse the iteration routines. It would be possible to disallow these "dangerous" operations whilst iteration is in progress using the Lock kit, if so required.

5.4.4. Partitions

Associated with each storage architecture hierarchy is a storage management hierarchy, derived from the `Partition` class. The partition hierarchy exactly mirrors the component hierarchy.

The partitions defined for each collection implement the general-case distributions described in chapter 3. They can thus distribute an instance of an AMM in some manner, although this is unlikely to be the most efficient distribution for particular applications. The class hierarchy is defined so as to allow individual aspects of the distribution to be changed independently of other aspects, to allow easy refinement.

"Area" Classes

Distribution of structures is performed using instances of auxiliary classes. The elements held locally by a component are represented by an instance of such a class, and other instances are used internally by partitions when constructing a collection and resolving requests.

The arrayed architecture uses a Region class, which defines a small region of an *n*-dimensional discrete space. The associative collections use a Slice object defining a portion of a hash space. Directed collections are sufficiently simple that they need no area class.

The advantage of this approach is that, by altering the behaviour of the area class, the behaviours of the collections may be altered. For instance, defining a new Region sub-class allows different shapes of region to be used in array decomposition, without changing either the component or the partition class.

[20]This is the only example of the use of multiple inheritence in Phœnix.

5.4.5. Activities

Multiple-worker tasks are supported in Phœnix using sub-classes of the class `Activity`. Each sub-class defines a different "process," and contains the supporting protocol necessary to interact with the collections to provide scalable parallel processing.

Within the `Collection` and `Partition` classes are functions which allow activities to be attached to collections in a single operation. These attachment functions are provided at the top of the class hierarchy, and require no support from sub-classes.

A single function, `Body()`, implements the activity's main function. This function is called automatically when the activity is attached to a collection. Functions are provided so that the activity can gain access to the component to which it is attached.

It is possible to determine the state of an activity – whether it has started, finished, raised an error *et cetera* – and to interrupt or kill it[21].

Groups

It is sometimes necessary to manipulate a group of processes *en bloc*. This is particularly true in a scalable system when, although it may be know that a number of concurrent processes are running, it may not be known exactly how many processes have been created.

The `Group` class is a sub-class of `Activity` which collects together a number of activities under a single name. All the functions concerned with attachment work with groups of activities, although many of them hide the `Group` object from the outside world.

`Group` supports exactly the same control interface as `Activity`, and dispatches the commands to all its members. It is possible, therefore, to create a group of activities and start their execution with a single command to their `Group`. The states of members are reflected in the `Group` so that, for example it is possible to wait until any one member of the `Group` completes its execution. This might be used for multi-version or speculative parallel computation, where the other members are killed as soon as one completes successfully.

Applications

Every Phœnix application contains exactly one instance of a sub-class of a special form of activity, an `Application`. This class contains special start-up code which initialises the Phœnix environment.

[21]Actually, although the kill function is provided for completeness in the interface to `Activity`, it is unimplemented: it is not possible to kill a process in a Transputer system.

5.5. Extensions

A major advantage in the use of partitioned collections as a memory model is that the model which an application uses is, at its most basic level, extensible. This allows memory to be made "intelligent" to better reflect the operations used in the application. This encapsulation of intelligence into memory modules both increases the level at which programming takes place and reduces the amount of communication needed in an application.

5.5.1. Issues in Extending Phœnix Classes

To minimise the amount of code which must be re-written, however, classes must be designed with sub-classing and extension in mind. This ensures that all the functions which may sensibly be re-defined are made independent of each other, so that single facets of a class may be changed.

Another important factor is the use of *abstract* classes: classes which cannot be instantiated directly, but which define the interface protocols for a selection of sub-classes. The abstract class is an "umbrella" defining a collection of possible implementations of the same abstraction, which may be used interchangeably.

Phœnix has been designed very much with sub-classing in mind. There are several abstract classes used in the programmer's interface – for example `Collection` defining the basis for all partitioned collections, and `ArrayedCollection` defining those functions which are specific to all arrayed collections. The bottom-level concrete classes, such as `Association`, may also be considered rather abstract, as they do not define a user-level access protocol.

In addition, Phœnix separates all aspects of the partitioning process into separate methods. This allows a single facet of the process – for example the generation of sub-regions of an array – to be re-defined by sub-classing whilst allowing the existing functions for creation and distributing these regions to remain unchanged.

5.5.2. Example Extensions

The Phœnix Extension kit contains some useful class pre-defined. These include some commonly-encountered memory modules and some sample custom distributions. It is these concrete classes which will be used as the basis for the evaluation of Phœnix in the next chapter.

Custom Components

Three collections have been implemented as part of the Extension kit: an array of real numbers, a dictionary mapping strings onto objects, and a binary tree.

For illustration, the interface to the `FloatArray` class – slightly abbreviated for clarity – is as follows:

```
class FloatArray : public Array {
public:
   FloatArray( Region *lr );
   FloatArray( int x, int y );
   ~FloatArray();

   /* copying */
   virtual void BasicCopy( Object *o );
   virtual Object *BasicReplicate( NodeAddr a );

   /* access */
   virtual void AtPut( Point p, float f );
   virtual float At( Point p );
};
```

A few points may be made about this class. The first is that it contains very little "real" code. The constructors simply pass information back to the parent `Array` class' constructor, whilst the access operations provide type-checking for the `BasicAtPut()` and `BasicAt()` methods implemented as part of `Array`.

Similar points may be made about the `Dictionary` and `BinaryTree` classes: in all cases most the work is being done by the memory architecture classes, leaving the user-defined classes to concentrate of application-level issues.

Custom Distributions

The partition sub-classes described earlier all implement particular, very general distribution strategies. Phœnix provides an additional distribution for one particular type of structure – the array – based around the most commonly-occurring mapping of arrays onto processors: as a rectangular mesh, with one component being placed on each processor.

```
class ArrayMeshPartition : public ArrayPartition {
private:
    ...
public:
    ArrayMeshPartition(  Collection *rc,
                         Partition *p =nil );
    ~ArrayMeshPartition();

    /* copying */
    virtual void BasicCopy( Object *o );
    virtual Object *BasicReplicate( NodeAddr a );

    /* partitioning */
    virtual void PerformPartitioning( void );
    virtual void BasicPartition(  int srs,
                                  Region *sr[],
                                  boolean ism );
};
```

The new partitioning policy – overriding that in `ArrayPartition` – is implemented by the `PerformPartitioning()` and `BasicPartition()` members. The latter takes an array of sub-regions generated by the former and creates a partition tree based upon them. The function is called recursively during the tree's creation. Hence the policy is divided into two parts: the division of the array into sub-arrays is performed by `PerformPartitioning()`, which then calls `BasicPartition()` to build the tree.

The same approach may be taken with other custom distributions. The various parts of the distribution strategy are separated into different functions, which may thus be altered individually as required. This minimises the amount of re-coding which is needed to implement new distribution strategies.

5.6. Résumé

This chapter presented an overview of a programming system based around the partitioned object model . The tool kit, called *Phœnix*, is composed of a set of classes written in a dialect of C++ extended to support the distribution of objects around a network and the asynchronous execution of methods.

The issues important in the design of the various layers of Phœnix were described to illustrate the possible alternatives in the design. The implementation of the various classes were then presented.

Some consideration was given to the extensibility of the tool kit, allowing programmers to create sub-classes of the important classes to define application-specific functions. It was shown that, by giving careful attention to the facets of the system which might meaningfully be re-defined, it is possible to maximise the flexibility with which users can customise the Phœnix classes whilst maximising the amount of design and code re-use within the system.

Chapter 6.

Evaluation

The pursuit of happiness is just a bore.

Mary Coughlan, *Mother's little helper*

The preceding chapters have presented an argument in favour of the partitioned approach, and its ability to abstract away from the most difficult attributes of scalable systems, and have described an implementation based on this model However, such arguments seem somehow incomplete: there is a need to discuss *post facto* the features of the system.

The partitioned model is *not* intended to extract the best absolute performance from a system: its aim is to simplify the programming task, possibly at the expense of efficiency. At the same time, parallel programming's *raison d'être* is to tackle computationally challenging problems, so too great a sacrifice in performance is unacceptable. For a prototype system such as Phœnix, however, it will suffice to identify areas of inadequacy and discuss ways in which they can be perfected. This is the focus of the following evaluation.

The evaluation proceeds along four paths. Firstly the partitioned model itself is evaluated in terms of the abstractions which it presents, and is contrasted against other possible implementations of scalable memory. Secondly the Phœnix prototype is discussed as an implementation of the model, and its shortcomings *as a programming system* highlighted. Thirdly, statistics are presented on some of the experiments run practically on the prototype, showing that, whilst its performance is wholly inadequate in practical terms, the overheads incurred are due to identifiable (and correctable) flaws in the prototype implementation. Finally some case study problems are developed using Phœnix.

6.1. The Partitioned Object Model

The partitioned object model attempts to provide a view of memory which is scalable, in terms of resource consumption and concurrency.

6.1.1. Meeting the Aims of Scalable Memory

The partitioned model is an attempt to implement the goals of scalable abstract memory laid-down in chapter 2, and we may compare it with this abstraction in order to determine how well it succeeds in this aim.

The aims of scalable memory were defined in the Introduction: to provide a system which

- manages and co-ordinates large quantities of structured data in a distributed-memory environment;
- regulates and controls massive amounts of concurrent activity;
- hides architectural details from programmers through the use of an abstract programming model;
- provides a supportive programming framework with scope for re-use, to avoid unnecessary re-invention; and
- ensures scalability by ensuring that applications can take advantage dynamically of whatever resources are available at run-time.

We may evaluate how the partitioned model meets these aims.

Distributing Data

A partitioned collection of data is essentially a memory module, according to the arguments presented in chapter 2. Since the data in the collection is *physically* distributed between its component objects, it may be implemented on a multicomputer without unnecessary centralisation; but since the data is *logically* centralised – all elements being accessible through any component, regardless of which component actually holds the item being sought – it effectively hides the physical distribution being used.

This has two advantages. The first is that a major source of complexity in distributed programming – the management of data locality – is removed from applications. However, since collections are actually distributed entities, there is still scope for the knowledgeable programmer to control the distribution of data as required. The important point is that this control is largely an optimisation of an application, and is not essential to its correct function.

Regulating Concurrency

In the partitioned model, the amount of concurrency used and its location, follows the distribution of data. Altering the distribution of a collection will affect the pattern of concurrency used to process it.

The justification for this view is that the size of a collection is often a good metric for deciding how to process it in parallel, whilst the distribution of data allows the programmer to exploit the possibilities for true concurrency provided by multicomputers.

Abstracting Away from the Architecture

The architecture has several effects on applications. It will determine the amount of data which may be held locally by a processor; suggest a certain "grain size" for concurrent computation; and define the cost of communications between processes on different processors.

In the partitioned model, all these factors may be balanced indirectly by altering the sizes of components and their distributions. This allows a partitioned application to be mapped efficiently onto any given architecture, but the mapping occurs *post facto* and need not affect the code of the application. The statement of problems within the model is to a large extent architecture-independent.

Programming Support

The kernel of scalable memory modules provided by the partitioned model may be re-used in many different applications, since they implement "general case" storage requirements. They may also be extended incrementally to develop new, application-specific structures. There is considerable scope for the re-use of designs and code within such a framework, simplifying the construction of distributed applications based around large amounts of shared data.

Ensuring Scalability

By providing a scalable memory model, the partitioned model ensures that a central aspect of program creation – its data organisation – is completely scalable. The use and re-use of partitioned data structures need not affect the basic algorithms used internally, so "intelligent" memory sub-systems may be created from the basic structures provided.

Moreover, the use of memory as a concurrency regulation infrastructure ensures that, for properly-written applications, the amount of concurrency used in an application is completely variable according to its distribution pattern.

These features do not, of course, guarantee that an application can scale. It is still the programmer's responsibility to ensure that applications have as little centralisation as possible, and that locality of reference is exploited to the full when creating worker tasks.

6.1.2. A Comparison of Possible Alternative Implementations

In §2.4 we suggested that a scalable memory would be best implemented using a community of objects, and it was this suggestion which gave rise to the partitioned model. There are, however, a number of other alternative implementations which

might also be considered, and we shall compare partitioning against the four most promising alternatives: Linda, distributed shared virtual memory, the use of objects within a DSVM framework, and Concurrent Aggregates.

Linda

The similarities between scalable memory and the Linda tuple space abstraction (§1.4.2) are obvious: both allow large collections of entities to be stored and manipulated *en masse* by processes distributed across a network.

Linda's tuple space is a large shared associative memory. Depending upon the implementation there may be multiple tuple spaces in existence, but most current systems implement only a single space shared between all processes in an application (and occasionally between all applications in a system). Single tuple spaces introduce the problem that processes must ensure that the tuples which they inject into tuple space do not conflict with those of any other processor.

A more serious complication – or advantage, depending on one's viewpoint – is that Linda completely hides distribution from the programmer. Whilst it is true that scalable memory aspires to the same ideal, the latter also seeks to allow the programmer to intervene to control distribution if desired in order to increase applications' performances. Early Linda implementations were very inefficient precisely because they could not automatically deal with the problems of efficiently distributing tuples.

It is perfectly possible to build structures like arrays in Linda, by using appropriate patterns of tuples. However, this reduction of arrays to tuples destroys all those spatial characteristics of arrays which are useful in distributing elements efficiently. For example, consider the case of a two-by-two array called *array1* implemented using the following tuples:

```
("array1", 0, 0, -1)
("array1", 0, 1, 0)
("array1", 1, 0, 0)
("array1", 1, 1, -1)
```

A priori, there is no way of identifying the relationship which these tuples have to each other – other than the fact that they will be matched by a common pattern, for example:

```
("array1",?x, ?y, ?v)
```

which information is insufficient to perform any intelligent distribution. The Linda assertion that “all tuples are equal” is a double-edged sword.

By comparison, the partitioned model retains the information about a data collection's essential structure, and may thus exploit it in creating a distribution pattern. The programmer may explicitly become involved, if he so desires, with the distribution of collections, using the full power of the host language rather than simple annotations.

Concurrent processing in Linda comes from the use of "active" tuples inserted into tuple space. By the same token as above, Linda prevents programmers from placing processes onto nodes (even as an optimisation step) and prevents processes from making use of the principle of locality (since there *is* no idea of locality in Linda). This places a large burden onto the Linda system implementor to manage the distribution of tuples intelligently, and there is no evidence that this is possible without programmer involvement – and Linda itself does not provide mechanisms for this involvement.

Distributed Shared Virtual Memory

DSVM is another logically shared memory for use in distributed systems, but one which is centred around access to memory at the word level. In seeks to simulate a simple "flat" address space by using the local physical memories of nodes as page caches in a virtual memory system.

Some problems in the scalability of DSVM systems have already been mentioned (§1.3.1). The problems of page usage and allocation, cache sizes and thrashing mitigate against the use of DSVM as a scalable memory implementation.

Representing Large Objects using Distributed Shared Virtual Memory

There is, however, a second possible use for DSVM. If a conventional object-oriented language is executing in a DSVM environment, then its object abstractions may be used as a model of memory without any intervention on the part of the programmer.

This is a very attractive possibility. The same collection techniques as used in (for example) Smalltalk or a C++ data structure class library could be introduced directly into the distributed computing arena. A collection of arbitrary size could be represented easily, since not all the pages of an object need fit onto a single processor: they may be distributed between several nodes, with the guarantee that any access to a "remote" data item will cause that item to be acquired transparently through a page fault.

This approach hides the distribution of data onto nodes – indeed, it makes it impossible to discover what data is on which node – and so could not be used as a concurrency infrastructure. The best one could achieve is to decide the number and location of processes and then allow them to divided the data between themselves using page faults. This means that processing becomes process- rather than data-oriented.

Indeed, this illustrates a major problem with *all* DSVM applications. There is a distinct separation between data and code, despite the fact that code must reside in memory. It is essential for an application to place its processes with great care in order to perform load balancing and minimise communication overheads: but the features of DSVM make this impossible by preventing the programmer from controlling distribution. In a very real sense, DSVM is equivalent to Linda in this respect, but at a lower level of abstraction.

Concurrent Aggregates

The system in the literature which bears most resemblance to the partitioned model is Chien and Dally's Concurrent Aggregates[36] (CA) (§1.3.2).

The main use of CA is as a concurrency-management system, since aggregates are inherently parallel to the degree of the number of objects within the aggregate – objects themselves may remain strictly sequential, but aggregates are concurrent. However, as a collection of objects manipulated using a single name, the parallel with the partitioned model is obvious.

The major difference is one of emphasis. CA is intended to introduce concurrency into objects, something which we have assumed to be present within the partitioned model and have controlled using auxiliary objects. CA is simply a framework, onto which may be added any desired functionality.

CA does not address the problems of deciding how many objects to create as part of an aggregate, nor of determining how to distributed the component objects or selecting a target site for interaction. All these are features of the partitioned model.

It seems, however, that CA would make an excellent possible host language for a partitioned system, since it provides many of the necessary features (especially concurrency control and message delegation). Although the described system was not strongly typed, there is no reason why this might not be added. It would be interesting to explore the effects on CA programs of introducing partitioned memory.

6.1.3. Some Problems with the Chosen Implementation

The partitioned model does, however, present some problematic aspects. Most serious are its use of software in routing requests, which is a direct consequence of the model's flexibility.

All routing of requests for data are sent round a partitioned collection using an algorithm embodied in the partition classes. The use of such software control makes the partitioned model very simple to optimise, as the algorithms used may be changed without necessitating the re-writing of the entire structure.

However, other possible implementations – notably Linda and DSVM – may make use of hardware acceleration to speed-up the routing of requests. The partitioned model, on the other hand, is not so susceptible to the use of hardware accelerators: this implies that, in purely speed terms, it is demonstrably inferior to the alternatives.

This objection may be answered in two ways. Firstly, there is a trade-off to be made between flexibility (which comes from software) and speed (which arises from hardware). A system with a hardware accelerator is very much tied into that accelerator's algorithm, and cannot adopt a different strategy if circumstances warrant.

Secondly, there is some scope for the use of hardware acceleration in partitioned systems. Hardware message routing would be a great advantage, and may be provided within the operating system kernel. Furthermore, many of the partitioned model's algorithms involve look-up against a table of possible values. There is a great deal of knowledge about the creation of hardware-based associative

memories[74], which could be used by partitioned structures with little problem (storing routing tables in a special memory with hardware support for the necessary searching).

6.1.4. The Programming Model and Method

The programming style encouraged by the partitioned model is one of shared data processed by several largely independent worker tasks. It is thus a *shared memory* model as opposed to a *message passing* model.

Since the partitioned model favours one of the two major parallel programming paradigms, it is natural to ask: for what classes of problem is the partitioned model suitable? For what classes is it unsuitable? Are these two classes sufficiently recognisable to ensure that unsuitable applications are avoided?

A shared data model, when implemented on a distributed memory machine, suffers from overheads whenever tasks request data which is not held locally to themselves. The farther away – in network terms – the data resides, the longer it will take to access. Although Valiant's work indicates that such a shared memory is implementable with only a constant factor overhead[113], it gives no clues as to the magnitude of this overhead. Hence in order to reduce the potential for unacceptable overheads it is essential that:

a. locality of reference is available *and is exploited*; and
b. data is accessed sufficiently frequently to justify its organisation.

The first condition implies that an application should not make "random" access to a data structure – by "random" we mean accesses which target elements without any pattern; or, put another way, there exists no distribution pattern such that the accesses may exhibit locality of reference. For an array, this might imply that elements accessed should be metrically close; for a graph, that only a few edges are traversed. Applications which *do* make random access will incur significant overheads.

The second condition states that applications must access the shared data a number of times. If an application accesses a particular data item only once, there is no advantage to be gained by structuring the data: it would be better to pass it explicitly using streams.

These conditions together identify a class of algorithms which manipulates a large shared data pool for a considerable length of time. The first excludes applications whose data accesses are unpredictable, the second those applications which use data "in passing." (Interestingly, these are precisely those conditions identified by Li and Hudak in analysing their DSVM system[77]. This would suggest that partitioning and DSVM are largely equivalent, with the former offering higher-level abstraction and the latter providing a more kernel-oriented approach.)

6.2. The Phœnix Prototype

In evaluating Phœnix it is necessary to compare it with both the theoretical qualities of the abstract model and with other parallel programming systems. We wish to determine whether Phœnix is a good implementation of the partitioned model, and whether it compares favourably with other similar programming systems.

6.2.1. Sufficiency of the Base System

The basic Phœnix system provides three things: a distributed and parallel dialect of C++, a set of high-level distributed memory structures, and an infrastructure for regulating concurrency. We shall first examine this system as it appears to the programmer, without refinement or specialisation, before going on to consider these essential issues. Doing this enables us to assess the ease with which Phœnix may be used for prototyping, before considering refinement.

It is possible to write object-oriented programs without using any tool kit support, simply using the facilities of the C++ dialect. There is little support for concurrency or distribution control, but applications *could* be created this way. The significance of this is that it shows that aspects of a problem which are *not* covered by Phœnix – or by the partitioned object model – may still be written. Phœnix attempts to make this process easier, but does not outlaw other approaches being used, and the admixture of several different paradigms may in some cases be beneficial[123].

The Phœnix memory classes provide as standard roughly those structures described in the standard work on the subject[70]. The abstract classes are chiefly concerned with refinement operations; the concrete classes provided in the extension kit (figure 21) supply basic functionality for several commonly-occurring structures. Applications could use these basic functions to implement an algorithm, with all the algorithm's sophistication being built into clients rather than memories. Although potentially less efficient, this approach is completely workable for a first cut at a problem.

6.2.2. Extensibility

Extensibility – the ability to re-use existing code and designs to create new classes for new applications – is a *prima facie* advantage of object-oriented programming. It reduces the amount of work, both in design and implementation, which an application requires

Extension can occur in two directions. When existing classes are specialised to provide new functions on an existing framework, it is termed *extension by differentiation*; when classes are being created to provide new functions, it is termed *lateral* extension. In practice creating a new class is often a mixture of these two forms of extension.

Extension by Differentiation

The mechanisms for differential extension are perforce limited to those in the host language. In C++, the main mechanism is the *virtual* member function which replaces the definition of the parent's function with a new function which is called in preference to the parent definition. The usefulness of differential extension is therefore largely defined by which functions are declared to be virtual, and by the decomposition of tasks into well-defined functions which may be replaced selectively.

In other languages, different mechanism exist. In Smalltalk, for example, *all* functions are by definition virtual: the dynamic binding of names to functions is the only mechanism provided, whereas in C++ static binding is used by default. The C++ version is in many respects more powerful and safer, as it allows functions to be defined which *must* be used in *all* sub-classes.

Phœnix was designed with extension in mind, and the interfaces provided to sub-classes for extension are intended to be the most flexible possible consonant with the need to maintain the integrity of the structure. Phœnix defines as virtual functions all those aspects of a collection or partition which might be changed as a matter of *policy*, whilst leaving statically-bound (*i.e.* non-virtual) those methods which maintain the *structure* of the collection.

This is a vitally important distinction, as it ensures that extensions may be made to storage architectures without compromising their integrity. Consider, for example, the partitioning of an array: the method which divides a region into sub-regions ready for allocation or further partitioning is defined as a virtual method, and may be re-defined in sub-classes to implement different partitioning policies; the method which takes these sub-regions and creates the partition tree from them is defined statically, since it is a matter of structural integrity, not policy.

Some of the classes in Phœnix are not directly related to partitioning, but are used by collections internally. Examples are the `Region` and `Slice` classes, defining the elements held by components of arrayed and associative collections respectively. These classes may be sub-classed like any other, and the sub-classes used indirectly to affect partitioning. A `Region` sub-class (for example) may be supplied to an `Array` to define its global storage: Phœnix is written in such a way that, when partitioning, the `Array` will use instances of the sub-class wherever an instance of `Region` would be used by default (all internal objects are created using copying rather than explicit creation). This makes it an easy matter to create an `Array` whose components hold hexagonal rather than square areas of the array's elements, simply by defining a `Region` sub-class with the given shape and passing it to the `Array` constructor[22].

Similar mechanisms may be used to define and utilise new `Partition` sub-classes to define novel distribution strategies. A `Partition` sub-class is created to implement the required policy and is then supplied to the root component of the

[22]Unusually-shaped array decompositions are found in applications such as computational wind tunnels, where a polyhedral locale may be used to improve the connectivity betwen neighbouring locales[118].

structure to be partitioned. The structure will then use the sub-class rather than the original `Partition` sub-class henceforth.

Both these forms of extension are completely type-safe within C++, as the type signatures of the member functions ensure that only suitable sub-classes may be passed into the collections – it is not possible to pass an `AssociativePartition` sub-class to an array, for example.

Lateral Extension

Phœnix naturally encourages the creation of new classes. If these new classes are defined as sub-classes of the Phœnix `Object` class, they will have the same functions and privileges as the basic objects: they will be available throughout the network to any object which know their name; may be copied and placed into partitioned structures; and may be used safely in a parallel environment.

A possible weakness is that there is no compulsion on programmers to derive new classes from `Object` (as there would be in Smalltalk), so it is possible to introduce classes into Phœnix applications which "misbehave" in some way. This is a result of C++'s class model, which does not force the class hierarchy to be a tree.

The creation of new partitioned collections is, of course, a major undertaking, requiring analogous functions to those contained in the Phœnix collection and partition objects to be implemented. Such extensions should hopefully be needed only rarely, if ever, since any data structure may be created by sub-classing one of the existing storage architectures,

6.2.3. Refinement

We shall now return to the issue of refinement within Phœnix, which is concerned with two things: allowing memory to manipulate data in an intelligent fashion and creating novel distributions of data. The former removes intelligence from parallel activities and places it into the memory; the latter allows the distribution of data to be customised. Both forms of refinement may proceed using differential extension of the basic Phœnix structures, so applications may be *progressively* refined.

Concurrency may in some applications be viewed as a refinement – an example would be an algorithm which is first implemented in a sequential manner and is then parallelised – but it is more likely that parallelism will be inherent in applications from their conception. Refinement in this latter case takes the form of balancing the distribution of a structure using its properties, in order to achieve an optimal trade-off between concurrent execution and communications overheads.

Intelligent Memory

Making memory "intelligent" essentially creates a memory module which is targeted directly at a particular application domain. This movement from the general to the particular may be accompanied by increased efficiency and readability in the

resulting applications and, if performed carefully, may allow significant amounts of re-use within the domain.

Since Phœnix separates its `Collection` class hierarchy into classes providing protocol and classes providing a storage model conforming to that protocol (for example the `AssociativeCollection` and `Association` classes respectively) it is a simple matter to supply new storage models for the same architecture. An example might be an array whose storage was created lazily using list-based storage rather than eagerly using indexed storage. It is also possible to change the storage model deeper in the inheritance hierarchy, since all storage management and access functions are virtual.

By default, Phœnix' memory modules simply allow access to individual elements. In many applications, however, data may be dealt with in larger chunks – entire array rows or collections of logical assertions, for example. By allowing the memory to perform the chunking internally, several advantages accrue to the programmer.

Firstly, an application's activities may deal with large conceptual units rather than with the raw units of memory storage. This allows algorithms to be expressed at the appropriate level. Chunking reduces the communication necessary between activities and memory, as more data is transferred per step: this allows the costs of resolution to be amortised across several data items.

Secondly, a memory may make use of knowledge about its distribution to optimise operations for speed. This weakens the independence of data manipulation and data distribution, but is useful as a refinement step. The simplest example of such optimisation would be the caching of recently-accessed data elements which were known to be read-only (or not, if cache consistency is implemented in an appropriate form). Another would be the pre-fetching of data in order to service later requests faster.

Thirdly, careful accumulation of intelligence into a memory allows applications within the same domain to share the intelligent memory. An example would be a bitmap (a variant of the array) which incorporated image-processing operations: many image processing applications could usefully share the common operations. The result of this process is the construction of domain-specific toolkits of classes, having the advantage over other class libraries that they would be distributed, scalable, and parallel, and could make extensive use of parallel processing.

Specialised Distributions

It is widely recognised that the distribution of an application has a major effect on its performance. There are several factors to be balanced:

- objects which interact heavily should be placed close together; but
- heavy interactions generate communications hot-paths and -spots;
- application peculiarities determine the appropriate grain size for distribution; but

- grain size is difficult to determine *a priori*;
- completely novel distribution strategies may be useful to optimise particular algorithms; but
- optimisation is a (usually) performance issue, not a fundamental question of design; and
- the appropriate distribution may not be immediately obvious for a new, complex or irregular application.

The issue of distribution is a complex one. The common solution is to provide a secondary configuration language to allow the distribution of program elements to be specified after the fact: examples of these are the Occam toolset, the Helios configuration language, and the Conic and Darwin systems. Phœnix takes the view that distribution, from the general viewpoint, is the concern of the machine, not the programmer. This implies that the system takes responsibility for placing objects, and allows it to re-configure dynamically.

The distribution of a memory is controlled by the partition class being used. This may be made arbitrarily intelligent by extending from the basic classes, which themselves provide a simple distribution suitable for prototyping. Hence the strategy used to distribute data may be refined: moreover, it is largely independent of the data manipulations being performed by the collection.

Furthermore, properties may be used to provide hints to the distribution controller at run-time. The classes supplied as standard allow various parameters to be set, so that their run-time behaviour may be altered – in important but semantics-preserving ways – until an acceptable pattern is found.

Covert Parallelism

One attractive possibility is the use of hidden, or "covert" parallel evaluation for complex methods in a scalable memory.

A method appears to the user as a sequential operation which runs to completion and terminates, with the caller being blocked throughout. Internally, however, the method is free to use concurrent techniques to improve its performance. As a refinement step, a simple method may be converted to use parallel evaluation without changing the external interface.

In doing this, a method may make use of all the concurrency regulation features of Phœnix. It may define an `Activity` sub-class which is then attached to the collection being processed, and wait until all the activities thus created have completed evaluation.

The need to use a new `Activity` sub-class for this operation is a problem with Phœnix, as it is extremely inconvenient and results in yet another class definition. A better approach would be to allow activities to be constructed from first-class functions – a point which will be addressed later, §6.2.4.

6.2.4. Defects

The problem with the use of a tool kit in programming, rather than a complete new language, is that the tool kit can do little to ameliorate problems introduced by the host language: any defects in the host propagate through to the tool kit. Many of the defects which one may identify in Phœnix are a direct consequence of the use of C++ as a host language.

However, Phœnix also suffers from other defects as a programming system. Both these classes of defect will be dealt with here: a third class, those concerned with performance, will be deferred until the next section.

A Hybrid Object Model

One of the major criticisms of C++ as an object-oriented language is that, by inheriting the functionality of C, it allows programmers to break its object model. This causes difficulties for the class designer.

In a "true" object-oriented language, all instances of all entities are objects. They may be all implemented in the same way (as in Smalltalk) or some may be optimised to provide a better representation, but all are objects conceptually. In C++, built-in types like integers are not objects, and follow completely different rules to those of application-defined classes. C++ makes great use of pointers. Not only are pointers used as object names (§5.2.2), they are also used to represent strings.

Both these factors complicate the construction of a class library. In order to be useful, the Phœnix collections must be able to store not only all types of object but also a variety of different entities which require different handling. Phœnix would be considerably simplified by a host language in which "all objects are equal."

Initialisation and Termination Functions

Another defect in C++ concerns the way in which initialisation and termination of objects are performed using virtual functions..

Consider three classes `A`, `B` and `C`, where `C` is a sub-class of `B` which is in turn a sub-class of `A`. `B` defines a virtual function `f` which is called from its constructor, and `C` re-defines this function.

Construction of an object of class `C` occurs by executing the constructors of `A`, `B` and `C` in order. One might expect that the constructor of `B` would, when calling the virtual function `f`, actually call the re-defined version in `C` (in accordance with usual practice for virtual functions): in fact, the original version in `B` will be called. The reason for this is that the constructor of `C` has not yet executed, and so the re-defined version of `f` may rely on initialisations which have not yet occurred.

The rule is therefore that a virtual function called in a constructor calls the version of that function "at its own level" (or lower) in the class hierarchy. A similar effect is observed (in reverse) with virtual functions called in destructors. This policy prevents errors caused by non-initialised variables and the like.

An effect of this choice, however, is that it is impossible to define a protocol for "top-level" initialisation of derived classes which is called automatically from the

constructor. It is necessary for the user to explicitly call a virtual function after construction has occurred. This is an added complexity, and somewhat at odds with C++'s goals of automatic object initialisation and destruction; it also makes it more difficult to create complex class libraries.

A solution would be to define two new special functions within the language, called (for example) `Initialise()` and `Terminate()`. Such functions should be implicitly virtual, and should be called automatically after construction completes (or before destruction commences) to allow top-level initialisation (or destruction) operations to be defined virtually.

Type-safety

When creating any general-purpose programming system, it is desirable to make it as general as possible. Of all the facets of a system which affect its generality, its type model is probably the most profound.

A statically- and strongly-typed language offers the possibility of creating programs in which errors caused by applying operations to inappropriate values can be eliminated. Although based on C – often used as the classic example of a language with no type system – C++ has a considerably tighter and more flexible type system. In particular, it allows a particular brand of polymorphism sometimes termed *inclusion* polymorphism (which Cardelli and Wegner[32] define as the property that "an operation may be applied to objects of different types related by inclusion") and overloading of both member functions and operators[23]. In practice this means that an object of type *B* which is a sub-class of class *A* may be used in all circumstances in which an object of class *A* might be used, since they share the same interface; the reverse substitution does not hold, as class *B* may define elements in its interface which are not available in class *A*. As with most object-oriented languages, C++ integrates both polymorphism and overloading through the inheritance hierarchy.

Phœnix uses this form of polymorphism to great effect. A function may, for example, be defined to accept an instance of class `Collection` and will then work with *any* `Collection` sub-class. Similarly, a collection may be refined without altering the code which depends upon it.

However, a major defect in C++'s type system, from Phœnix' point of view, is that it does not support two other useful forms of polymorphism: *functional* or *parametric* polymorphism[40].

Functional polymorphism allows functions to accept arguments of any type, which they do no manipulate in any operational fashion. An example of this would be ML[117], where a function such as that which performs the *map* operation (which constructs a list by applying a function to all members of another list) may be defined as

[23]There is occasionally some confusion over the difference between overloading and polymorphism. Overloading (sometimes called *ad hoc* polymorphism) allows the *same* identifier to refer to *different* functions, with the appropriate function being chosen at run-time according to the type of its arguments; polymorphism allows the *same* function to apply to items of *different* types, with no run-time selection necessary.

```
val map = fn [] f -> []
        | fn h::l f -> (f h) :: map l f ;;
val map = fn : 'a list -> ('a -> 'b) -> 'b list
```

and can work with any values of `'a` and *`'b`* (which represent type names) as it does not manipulate values of this type explicitly – it is sufficient to know that the list is of *some* type. This is similar to the common C (and C++) trick of type-casting values which are not used as `void *`, but with the important difference that casting in this way loses all information about the original type of the value.

Parametric polymorphism is found in languages such as Russell[22], where a type may be passed as a parameter to a function. A Russell implementation of the *map* operation would be

```
map ==
   func [  l : val List a ;
           f : func [ val a ] val b ;
           a : type {} ;
           b : type {} ] val List b
   {
      if l = Null ==>
         Null
      # else
         f[hd[l]] ^+ map[tl[l], f, a, b]
      fi
   }
```

whose major difference from the previous definition is that the types `a` and `b` are passed explicitly as a parameter and is available for use within the function's body, although in this case there are no operations defined for them. In general, it would be possible to use any of the operations known to be available on that type, and the function could be applied to any type providing these operations – and this type conformance could be checked statically[24].

For Phœnix, it would be useful to use both these polymorphic arrangements. Collections could then be made type-safe for any type of value stored, without the need for sub-classing.

The templates feature defined in the C++ standard (and the Ada generics mechanism from which it is derived) are *not* equivalent to the above. A generic package in Ada must be instantiated for a particular type at its creation, before use. Thus it is not possible to define a generic type `List` parameterised by the type of elements in the list, and to then write a function which will perform *map* over `Lists` (although one could construct a generic function to do so, which would then itself have to be instantiated).

[24]A similar effect may be achieved in languages which allow sub-typing without allowing type-valued variables. In effect, the sub-type information makes available a set of operations which may be used.

Storage

Local memory in Phœnix is represented by the `Storage` classes `ListStorage` and `IndexStorage`, which are parameterised by the size of their elements and are accessed in a typeless manner. The use of a parametrically-polymorphic type system could remove this typeless-ness by allowing the type of values which a collection is to hold to be passed to the storage object.

The use of variably-sized storage is, however, only necessary because of the variable size of possible elements, and this is in turn related to the tight coupling of objects to memory found in C++. It is not reasonable to represent integers as objects, for example – especially not in Phœnix, where object names are of the order of tens of bytes long – and so it must be possible to store shorter values within a collection. The use of shorter object names, and the implementation of Phœnix in a language in which all object names are of the same, small size (such as Smalltalk) would alleviate this.

The Proliferation of Classes

A cursory glance at the Phœnix class hierarchy shows that it contains a large number of classes, many of which are largely empty of new functionality but which are needed to implement type-safety or some other slight interface variation. This problem is again largely solved by the addition of other polymorphic forms to the host language.

The most problematic feature comes in the creation of `Activity` sub-classes. Every worker task is represented by an instance of such a sub-class. The creation of a new activity is therefore a very heavy-weight operation and, what is more, must be performed at compile-time.

It would be far more attractive to be able to create activities from functions so, for example, a new activity could be created simply by instantiating the `Activity` class with the function which it was to execute, rather than defining a new sub-class and embedding the function within it.

In order for this to be practical, functions must be *first-class* entities, The availability of first-class functions has many advantages, and this additional use as a means of defining parallel activities comes "for free." It means that a function may be created dynamically, according to run-time conditions, and be then turned into an `Activity`.

6.3. Performance

Our intention in evaluating the performance of Phœnix is to illustrate those features of the partitioned model which most influence an application's efficiency, *not* to evaluate the current prototype as a practical programming environment. We shall first analyse the theoretical sources of overheads in Phœnix, derived both from the partitioned model and from its overall implementation, before presenting some experimental performance figures.

6.3.1. Theoretical Performance

In performing this analysis one important source of overheads and delays must perforce be ignored – those arising from network contention and routing. The reason for this is simply that the very features which allow the partitioned model to scale and precisely those features which mitigate against being able to model communications patterns to the accuracy necessary to include routing delays. We shall therefore make the assumption that calling any function takes a unit amount of time, no matter how the objects are distributed and what else is happening in the network.

Creating and Destroying of Objects

Object-oriented programming tends to make extensive use of objects with very short lifetimes, so the mechanisms for creating and deleting objects can make an important contribution to application speeds.

Creating an object involves three steps:

1. Decide upon the node where the object is to be created (using explicit placement or load balancing);
2. Interrogate the namer to see whether a class server is available on that node;
   2a. If no server is available, interact with the host operating system to create one;
3. Make a remote procedure call to the appropriate constructor in the selected class server.

The major cost, of course, is the creation of the class server, which (if necessary) will involve calls to the machine's file system. Once loaded, the class server's executable image must be transported across the network to the selected site and then started up.

Deleting an object is somewhat simpler, involving only two steps:

1. Make a remote procedure call to delete the object;
2. (Optionally) Close-down the server if the it no longer holds any objects.

As with the calling of constructors, calling destructors may cause a certain amount of additional activity. The decision as to whether a server should close-down when it has no objects remaining is a matter of policy: leaving the server running means that, if objects are created on the node at some time in the future, the server must be re-started; leaving it running may use memory unnecessarily.

Accessing Data

Data access may be divided into two possible cases: accessing data which is held locally, and resolving data which is held remotely. In the latter case, local access is performed after resolution has occurred, so the local-access operation is common to both cases.

Local Access

Accessing data held locally involves the following steps:

1. Testing whether the element held is held locally;
2. Acquiring (or possibly failing to acquire) the data, or setting the data, or some other operation on the local element.

The locality test is reasonably simple for all the storage architectures. For an array, it involves testing whether a point lies within the component's local region; for an associative memory, testing a hash key prefix; and for a directed structure testing whether the node's parent is one of those whose children are held locally. All of these cases, in the default architectures, involve communication between the component concerned and another object, so at least one remote call is needed for local accesses.

If the element being accessed is indeed held locally, then it must be located from the local storage. This operation obviously involves no remote communication, but will involve a search of local storage.

Resolution and Remote Access

If the locality test fails, then resolution must be invoked. This involves making a request to the partition associated with the receiving component, the resolution process itself, and a local data access operation at the servicing component.

At each stage of the resolution process – *i.e.* for each partition visited as part of the process – some cost will be incurred for further routing. This cost will always include the cost of the remote call made to the partition: the rest of the cost will depend upon the architecture involved.

For arrays, if we assume that the regional decomposition strategy is used, each partition will hold references to (on average) half the total number of sub-regions in the array. In each component, assuming a linear search, each resolution step will need to test on average half these sub-regions before finding a match: therefore, denoting the total number of sub-regions by $n_{regions}$, the total cost per step will be given by $1 + \frac{n_{regions}}{4}$ method calls. For associative and directed structures, resolution may occur without any remote communications and will thus take a single call (that to the partition) per stage.

		Structure		
		Arrayed	Associative	Directed
Local data		$1+\tau$	$1+\tau$	$1+\tau$
Remote data	Request	1	1	1
	Per stage	$1+\frac{n_{regions}}{4}$	1	1
	Local access	$1+\tau$	$1+\tau$	$1+\tau$
	Total	$1+n_{stages}\left(1+\frac{n_{regions}}{4}\right)+\tau$	$1+n_{stages}+\tau$	$1+n_{stages}+\tau$

Key: τ = local access time
n_{stages} = number of resolution steps
$n_{regions}$ = average number of local sub-regions (arrays only)
(All figures are in units of one remote method call.)

Figure 24: Analysis of costs involved in accessing data

The total costs of resolving remote data elements is summarised by the table in figure 24. This shows a general pattern in the cost of accessing data: the initial request, a number of resolution steps, and the access to local storage involved in accessing the data when it is eventually resolved. In other words,

$$overhead(e) = 1 + n_{stages} f(e) + \tau$$

where *overhead(e)* is the overhead involved in an operation accessing an element *e*, *f(e)* gives the cost involved in the resolution of *e*, if any, and τ is the local access time. For locally-held data, *f(e)* is zero; for associative and directed structures, it is a linear function of the number of resolution steps required (*i.e.* of the metric distance between the receiving component and the component holding the element); for arrays it is a product of the number of stages and the number of sub-regions (also linear, since $n_{regions}$ is constant for a given array).

The result for arrays indicates the importance of better distributions in the array case: for large arrays, where $n_{regions}$ is large, a significant overhead is incurred at each stage of resolution; moreover, it is reasonable to assume that n_{stages} varies with $n_{regions}$, so large arrays will also involve more resolution steps.

Although the costs of resolution are linear in terms of the number of stages or resolution performed, the number of stages is itself often a logarithmic function.

Consider the case with an associative memory: each level of resolution increases of decreases the number of possible values accessible by a multiplicative factor. The resolution complexity of *any* partitioned collection is logarithmic; the actual cost per stage varies according to the type of collection and the distribution used.

6.3.2. Experimental Performance

Rather than fill the section with data, we shall concentrate on the most important low-level performance aspects when performing the experiments. These are the factors which affect all other aspects of the prototype's performance, and so are most representative of its drawbacks.

The experiments were performed using Phœnix on the existing Wisdom kernel. This is composed of a four-by-four mesh of T800 Transputers connected *via* a serial line (100Kb/s) to a Sun file server acting as a Wisdom host, which is in turn connected by a 10Mb/s EtherNet to the departmental filing system. All Wisdom executable programs are loaded using the Sun NFS protocol, with Wisdom itself acting as an NFS client.

For the experiments, Phœnix was instrumented with a class `Logger` to record logging information. This information took the form of a single asynchronous request, and was stored along with a time-stamp and a source-node record. Logging was activated and de-activated through a property.

The experiments were designed to identify the factors which introduce overheads when creating applications with Phœnix, and to quantify these overheads.

In general, it may be noted that it proved very difficult to obtain timings of any real significance for Phœnix. The system's flexibility in allowing different aspects of its behaviour to be changed easily means that representative timings are hard to come by: there is always the possibility that a better distribution pattern exists. The instrumentation used is rather intrusive, and can distort the transport times for messages significantly. Problems were also experienced in the C run-time library with regard to the accuracy and length of the system timers, making it impossible to obtain long-term timings.

Basic Properties

The start-up of a Phœnix application involves the following tasks:

- creation of the master (`Application`) object;
- creation and loading of the `PropertySheet` and `Logger` objects;
- the running of the `Application` object's main code section.

The first experiment determines the overheads in this process, which may then be discounted in future timings. The average time taken to start an application was recorded by the `Logger` as 305ms. This figure, of course, does not include the time taken to load most of the object servers – for `Logger`, `PropertySheet` and the

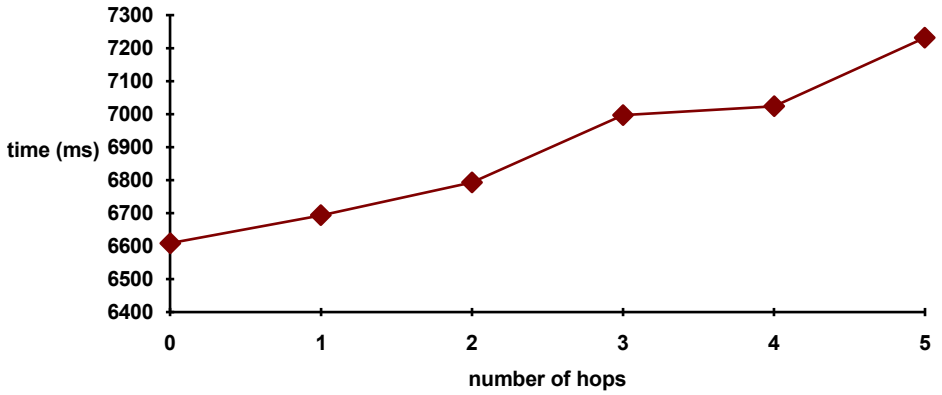
`Application` – so these occur "before time" and may be discounted in this and all future experiments.

The time taken to run an application which performs no action – simply starts-up and then terminates – is 492ms.

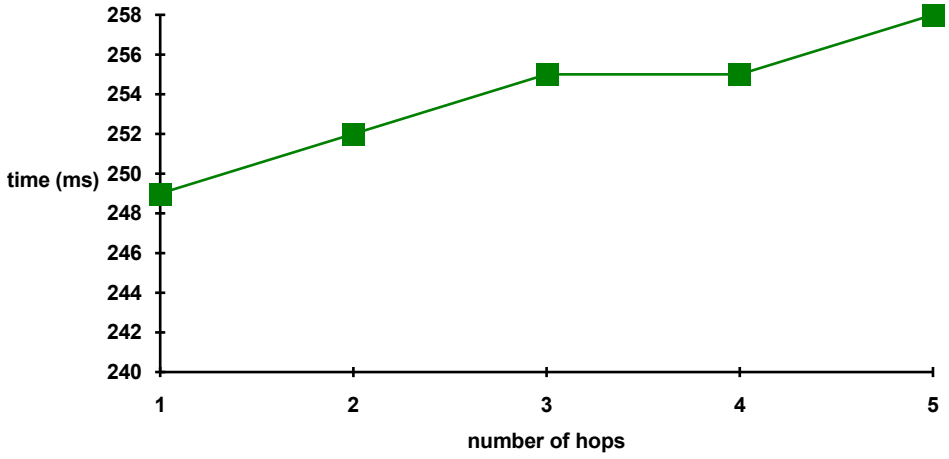
Object creation

The creation of objects may occur in two modes: in the first, an object is created onto a node which is not running a suitable object server; in the second, an object server is running at the target node.

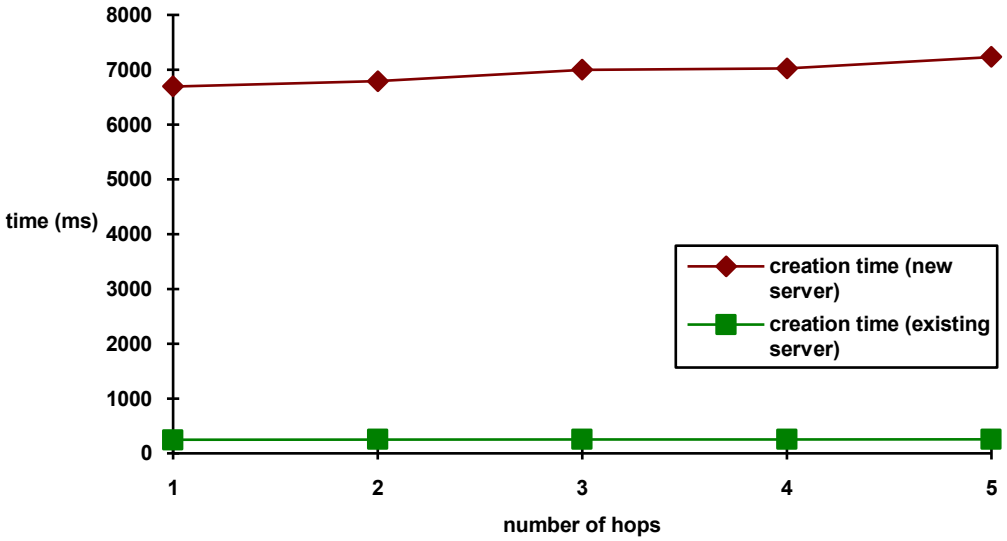
In both cases, the number of "hops" (communications links traversed) between the creating object and the new object was varied. In the first case, the results were as follows:



In the second case, the results were:



We can illustrate the difference between these two sets of timings in the following graph which plots both timings on the same axes:

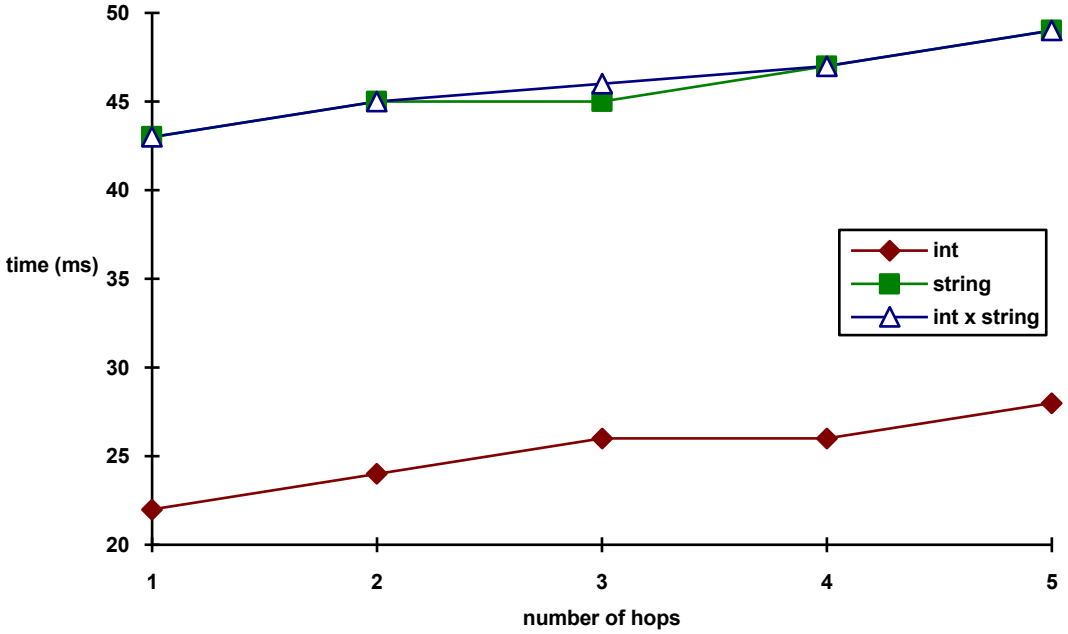


From this it can be seen that there is a generally linear relationship between distance and object creation time, and that the time taken to start a new class server is of the order of 6 seconds. This is a quite appalling figure! It is somewhat mitigated when the current experimental set-up is examined. Files must be retrieved from the host file system across a serial line, which achieves a transfer rate of less than 100Kb/s. In addition to this, there is a certain (variable and effectively unquantifiable) overhead associated with accessing the departmental file servers. (File system design in scalable is discussed extensively by Austin[9][11], and one conclusion is that filing systems require extensive kernel support if they are to be efficient, especially in the area of large-packet message transport. This feature is not currently available in Wisdom, with the result that file system access times suffer.)

Still, the figure is quite unacceptable. It indicates that significant gains in performance may be expected if the system has all its necessary code servers loaded ahead of time, rather than loading them "on demand." This is exactly the approach taken by other systems, where all an application's code is assumed to reside on all nodes. For a multi-user scalable system this assumption is not realistic: in the interests of realism, we shall use lazily-loaded code in all experiments, but the benefits of eager loading should be borne in mind.

Method Calls

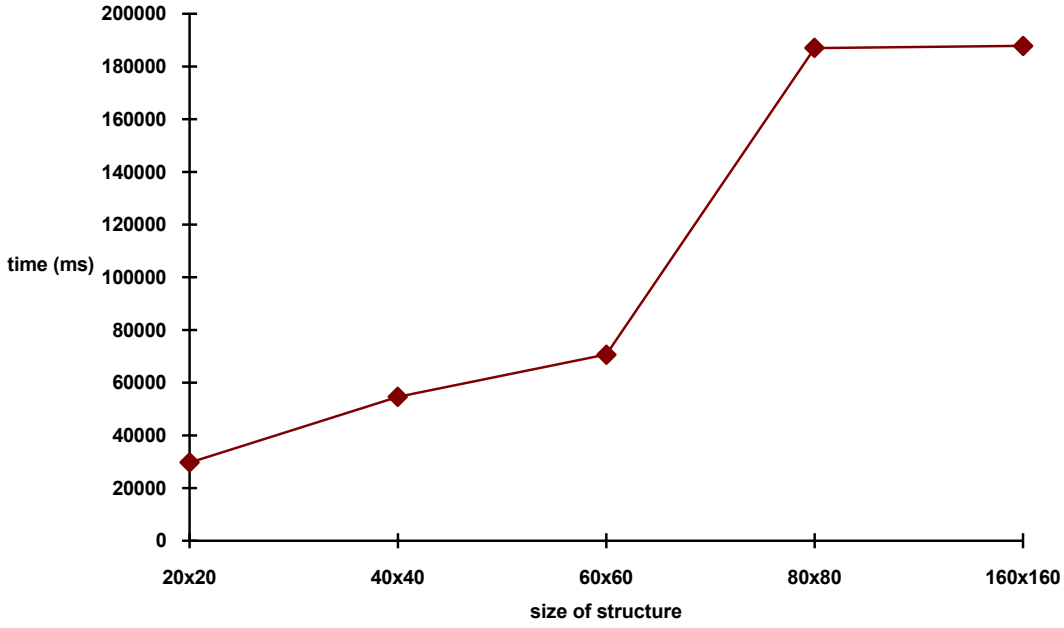
An experiment was performed in which method calls were made between objects at varying distances. Three different methods were tried, taking arguments of an integer, a string, and an integer and a string. The results for varying distance (averaged over 100 repetitions) were as follows:



These figures indicate that the (un-)marshalling time for integers is negligible, but that for strings is of the order of 20ms (the marshalling time for reals is comparable with that for integers, as is the time taken to marshal an object handle). There is a constant overhead of around 20ms from acquiring packets, creating server processes *et alia*. The time taken to send a method call averages at about 1ms per hop (in each direction).

Partitioned Collections

Arrays, unlike the other storage architectures, allocate all their elements at their creation. The creation time for various sizes of array were as follows:



Note the unusual shape of this graph: there is a "step" where a significant number of components must be created, but then a tailing-off as components are placed onto nodes which already have servers running.

Creating associative and directed collections takes approximately the same time as taken to create an array with a single component: experimentally, this comes to around 20 seconds (adjusted) A more meaningful time is the time taken to split a component of such an architecture when one overflows.

Splitting a Component

An associative memory splits a component whenever the number of elements in that component grows to be too large. The time taken to perform a split depends upon several factors, notably the number of new buckets which are created as a result of a split operation and the number of elements which were in the bucket being split (both controlled by properties).

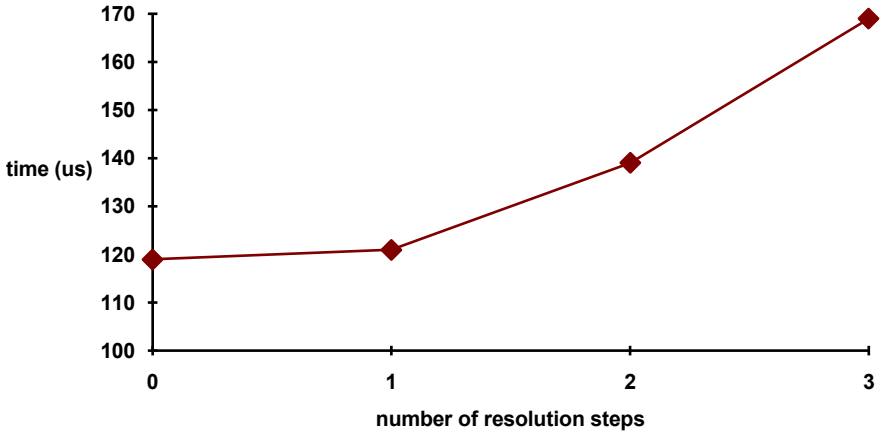
Take as an example a structure with ten-element components which, when split, generates four new components (*i.e.* adds two bits to the hash key). The splitting operation takes 27674ms (creating nine new objects and four new class servers). Added to this is the time taken to re-arrange the contents of the split component between the new buckets (averaging around 2200ms – ten resolution operations).

The times for directed structures are almost identical, and are not shown. This is hardly surprising, given the similarity between the algorithms used in splitting both structures. The re-arrangement operation averages at 1800ms.

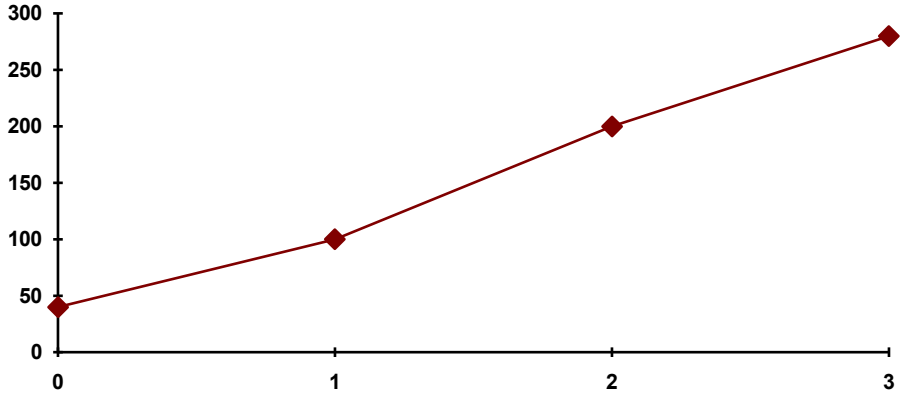
Resolution

The resolution of elements within an array depends upon the distance (in terms of resolution steps) between the receiving component and the servicing component:

there is also a factor of distance in terms of hops, although this is less significant. The following results were obtained by for accessing an array of fixed size, varying the targeted element to vary the distance between receiver and servicer:



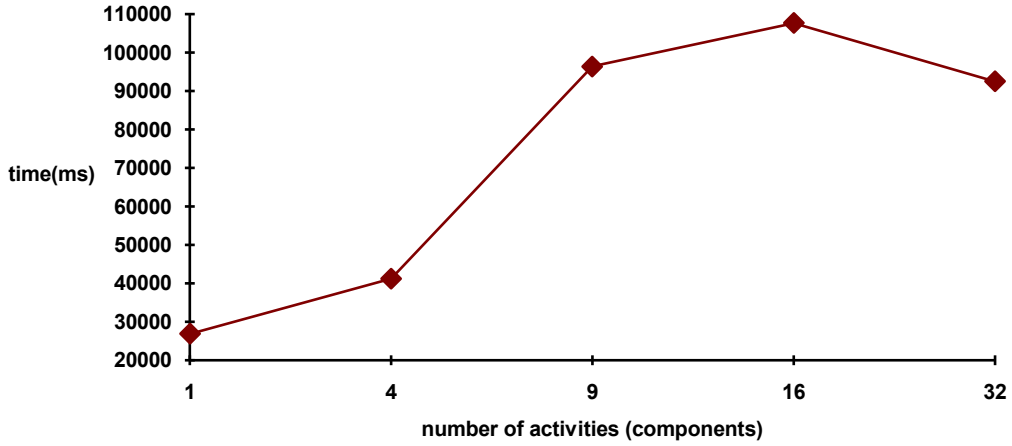
An associative memory incurs similar overheads, but with better linearity due to the simpler resolution protocol:



A directed collection resolves along edges in a constant time, approximately 20ms. Retrieving data from each node follows the same timing profile as retrieving from an array.

Activities

The time taken to create the replicas of an activity is the main factor affecting an application's start-up time. The time taken to create replicas of an activity (performing no action) are as follows:



Notice again the "step" in the graph where re-use of servers occurs.

6.3.3. Discussion

In discussing the figures arrived at above, we shall start with the admission that they are disappointing in the extreme. There are two main reasons for this: the prototype nature of the low-level support system and the granularity of decomposition.

The low-level supporting system of Phœnix, especially the RPC kit, are very simplistic in their approach to communication, and could be significantly optimised given time. In particular, the way in which parameters are marshalled could be much improved.

At present a parameter block is built by moving the value of each parameter, individually, into the block. This is somewhat unnecessary, as there already exists a suitable block holding all parameters: the stack frame of the current RPC stub. In principle the contents of the stack frame could be transferred directly to a parameter block; in practice, the way in which C++ handles certain structures (especially strings) as pointers to local memory, and the internal architecture of the Transputer, make this rather too difficult. Given a different, more regular host language on a different processor architecture, it would be possible to reduce significantly the delays associated with RPC.

The second problem is more serious. As mentioned above, Phœnix is written so as to be highly modular and hence highly extensible and customisable. This is accomplished by a detailed object-oriented decomposition in which each object represents a separate logical entity.

Unfortunately this results in a profusion of objects. Consider an array. An array consists of several components, each having an associated `Region` object holding the bounds of its local data. This arrangement allows new `Regions` to be defined – defining, for example, hexagonal areas of space – without altering the definition of the component class. This makes for easier extension; it also means that each test for locality in a component results in a method call to another object. Even though the `Region` is likely to be on the same processor as the component, the parameter

marshalling overhead is still incurred. A similar problem exists in associative memories, in which slices of hash space are represented by objects.

The overheads thus incurred are prohibitive, and make Phœnix impractical as a programming environment. Ironically, this is a *direct* result of *proper* decomposition! The use of the "correct" discipline results in a severe degradation in efficiency.

In some ways, however, this is point in Phœnix' favour. It is possible to develop software according to the traditional, well-accepted disciplines and have it run in a scalable manner – albeit very slowly. Phœnix then allows an application to be refined, and one possible refinement path is to remove some of the auxiliary objects – effectively freezing the application into a particular pattern and reducing its flexibility – in order to reduce communications overhead and hence improve performance. This introduction of performance after-the-fact may make for faster software development times overall, as the early stages may be programmed experimentally.

6.4. Three Examples

We shall conclude this evaluation by creating some Phœnix applications. Firstly we shall review the practical comparison of Phœnix with the Booch Components. We shall then follow the creation of two applications from start to finish. This derivation illustrates the stages which Phœnix encourages in creating scalable parallel applications.

In the examples, the code presented is functionally that required by Phœnix. We have, however, converted the syntax into "pure" C++ rather than use the pre-processor directives necessary in the current prototype, and have elided some low-level details. The structure of the code, however, is that of Phœnix.

6.4.1. Example One: the Booch Components

The Booch Components[25] are a library of data structures used extensively in the Ada community, and which have since been implemented in Ada, C++ and Ada-9X. As an established class library consisting almost entirely of data structures, it offers an ideal vehicle for testing the flexibility and extensibility of the Phœnix memory modules: the result is a parallel, distributed and scalable implementation of an existing software tool.

An Overview of the Booch Components

The most accessible presentation of Booch's work is [24], which covers the components' implementation in C++. The original version of the Components, in Ada, is presented in [23].

The Component library consists of four orthogonal elements:

[25]We shall use the term *Component* (with a capital C) to refer to the Booch components, to avoid confusions with Phœnix' *components* (with a small c).

- abstract data types;
- internal memory representations;
- concurrency controllers; and
- utilities which may be applied to structures.

The abstract data type classes define minimal abstract data type interfaces to a component. For example, queues, sets, strings, rings, trees and maps are all provided as abstract data types. For each data type one of several memory representations may be used, each having a different time/space complexity, which may include garbage collection. Concurrency controllers control access to data structures in a multi-tasking environment. The utilities implement functions such as sorting and searching, and may be used in conjunction with structures of several different types.

The first three element categories exist (in the C++ implementation) as independent class hierarchies. To create a usable data structure, an application creates a class which multiply inherits one of the data type, storage management and concurrency control classes. The class itself need provide no extra functions (although it can if required).

Comparing the Components with Phœnix

Some immediate differences between the Components and Phœnix are evident.

The primary difference is that the Components are *not* classes: rather, they are templates from which classes may be constructed by combining an ADT, a storage manager and a concurrency controller. The classes are parameterised according to the type of their contents, using the templates mechanism defined in the C++ standard.

Phœnix collections are classes in their own right, containing their storage management and concurrency control protocols within them. The classes are not parameterised by type: instead the basic architecture classes are parameterised by the *size* of elements which they are to store. Type-safe access is provided by sub-classing the component class and tightening the type checking at its interface.

At first glance, it would seem that Phœnix' collections are considerably less flexible than the Components. Their storage management and concurrency controllers are in-built rather than being composed from "outside," and their type parameterisation is less safe. However, the internal architecture of Phœnix makes it quite straightforward to implement a new local memory architecture by re-defining the collection's basic access members: this is a result of the careful use of virtual functions. The same is true of concurrency controllers.

Type safety remains a problem in Phœnix. The templates mechanism of C++ is not implemented by very many compilers, and those which *do* implement it usually use some form of macro expansion. In a shared-memory environment this may be acceptable, but its use in Phœnix would be complicated since every new class must have its own class server. Ideally a more flexible type system than that of C++ would be used to alleviate these problems.

Local storage management architecture is slightly less flexible in Phœnix than in the Components, as Phœnix aims to maximise the use of global rather than local memory. Phœnix adopts a slightly more cavalier attitude to local memory, but

allows global memory to be co-ordinated; the Components assume that all memory is shared, and attempt to optimise its use by providing storage managers which make different trade-offs in time and space.

The concurrency controllers provided for the Components are quite low-level, especially when compared to Phœnix' use of deontic logic. They are tailored for use in an environment where concurrency is the exception rather than the norm: in Ada, one might reasonably expect that only a few tasks will share a set of data structures. There is no attempt to minimise potential interference between concurrent accesses.

The Component's associated utility objects – for searching, sorting *et cetera* – are similar in flavour to a set of commonly-needed activities which might be implemented easily in Phœnix.

The most radical difference, of course, is that Phœnix collections are distributed and have support for concurrency regulation. There is no feature in the Components corresponding to Phœnix' use of data structures as infrastructures for concurrency regulation, and the Components do not address issues of distribution. Since each Component is a single object, they cannot be distributed as they stand even with a distributed host language (*e.g.* [62]). This makes then unsuited for use in a distributed, highly parallel environment.

Sample Implementations

Several Components were re-implemented in Phœnix to examine whether there were any important gaps in the Phœnix class hierarchy as compared to that of the Components.

A selection of the Components were easily implemented in Phœnix. These re-implementations shared all the important features of the Components in terms of external interface (functionally, though probably not syntactically, as the C++ Components' interfaces are not available in the public domain). For example, the Ring component can easily be implemented using a directed storage architecture. (The Components do not provide an array type, as this is provided intrinsically by the host language.) The Phœnix associative memories may be used to implement structures behaving like the sets, bags and maps (dictionaries) found in the Components.

The Phœnix implementations were completely scalable, which would not be the case with an implementation of the Components. They provided a better interface for concurrent access. Neither of these two results is surprising, as Phœnix was specifically designed with these aims in mind: it does illustrate, however, that there are important differences between computing in scalable and non-scalable environments.

One difference which was discovered was that large-scale queues and lists may be highly inefficient in the partitioned model, essentially because of their single points of access: adding to the head of a list must always be directed to the same component of the directed structure which composes the list, for example. Fortunately such large lists – or, at least, this sort of manipulation of large lists – are quite rare, and it is acceptable to create partitioned lists (or queues, or dequeues, or other similar structures) as directed collections which are seldom partitioned. This may be accomplished very simply in Phœnix by setting an appropriate property: the

result is that all elements of the list will tend to reside on the same processor, which has the same problem with hot-spots but avoids any partitioned overhead. Hence the properties mechanism allows problems like these to be addressed without changing any of the application's code.

6.4.2. Example Two: a Cellular Automaton

Cellular automata are a commonly-encountered processing model for highly regular, highly parallel systems. They are especially prevalent in simulation studies of physical systems such as gases and fluids. A good overview of the field is presented by Wilson[118].

Coarse Structure

A cellular automaton is essentially an array of points, each representing a discrete amount of "real" space. A single point in the automaton has the "average" value of the area which it represents: the more points the automaton has, the closer it will approximate to a continuous space and the better its behaviour will model the "real world." The value of each point is typically a simple integer or a vector.

Conceptually, a cellular automaton simulation centres around a large array of points, where each point is a structure which holds the properties being modelled as they evolve through time.

Computation in the automaton occurs in the following manner. A point is updated at time *t* by obtaining the values of its immediate neighbours (and itself) at time *t-1* and averaging these values. Hence a point's behaviour is only directly influenced by its immediate neighbours (although indirectly it is influenced by the entire model), and so the computation exhibits an almost ideal locality of reference.

Parallelism in such systems comes by updating the values of several points simultaneously. In principle, every point may update its own value: in practice, in automata consisting of millions of points, this amount of parallelism is unmanageable and processes are assigned to update the values of a locale of points.

Points and Access

A single point within a cellular model may be quite complex. A simple example is a model of electrical field characteristics, where each point holds the field intensity; more complex is a fluid flow system in which each point holds the local flow vector.

It is the updating function – controlling how a point's value is updated by its own and its neighbours' values – which determines the functionality of the model. For our current purposes, the details of this function are largely irrelevant: a set of values must be obtained from the array and have a simple function applied to them.

An important feature of a cellular system is that there may be some amount of asynchrony in evaluation. It is not necessary to force all points to be updated for time *t* before proceeding to time *t+1*: it is only necessary to wait for the values *of those points actually being interrogated* to arrive at the current time. This does, however, imply that a point can carry with it a certain amount of its past history, and that values from times in the past may be obtained.

A Primitive Implementation

The most primitive implementation possible builds the automaton from the basic array class, extended to hold elements of type `Cell`:

```
class Cell : public Object {
private:
   ...
public:
   Cell( int history =DEFAULTHISTORY );

   float GetValueAt( int t );
   void SetValueAt( int t, float v );
};
```

For simplicity, we shall assume that the `GetValueAt()` operation will block until a value for the requested time is set, and that the last value set defines the current time. This may be expressed using a customised `Lock`:

```
boolean CellLock::CanStart( int st, int t ) {
   switch(st) {
      ...
      case SetValueOp:
         ...
      case GetValueOp:
         return (Finished(SetValueOp)>=t);
   }
}
```

The automaton may be built using the ordinary `ObjectArray` provided as part of the Phœnix extension kit. The cells' values are initially set to zero, with two high-value “peaks” being placed manually towards the centre of the model:

```
ObjectArray *a;   Cell *cell;
Point p(2, 0, 0), q(2, 10000, 10000);
Region *r =new Region(p, q);

cell=new Cell;   cell->SetValueAt(0, 0.0);
a=new ObjectArray(r);   a->Initialise(cell);

cell->SetValueAt(0, 5000.0);
p.Is(100, 5000);   a->AtCopy(p, cell);
p.Is(9900, 5000);   a->AtCopy(p, cell);
```

A `Cell` is first created with an initial value of zero. An `ObjectArray` is then created with the desired dimensions (specified by its two diagonal corner `Points`), and is initialised using this `Cell` (which will be copied into all values).

`Initialise()` causes the array to be partitioned. The two peaks are then inserted by placing copies of a high-valued `Cell` into the array.

Properties

The scalability of a collection is controlled by several factors, the most important of which is the size of each component. Two considerations affecting the selection of a component size are the size of each node memory and the amount of work which will be performed on a component by activities.

The size may be specified as a property by altering the property sheet. At its creation, an arrayed structure acquires this property and uses it as a hint to determine the size of components, the number of components created and (indirectly) the number of nodes over which the collection distributes.

In this example, selecting a component size of 1000 elements on each side (1000000 elements per component) would result in 100 components, each of which might potentially be placed on a different processor; it would also result in 100 parallel activities being created to process the array.

A Customised Access Protocol

The interface for accessing cells – using the `At()` and `AtPut()` operations in conjunction with `Points` – is rather awkward for the current application. The mechanism is designed so as to cope with arrays having any number of dimensions, but in the current application we know that there will always be exactly two dimensions to the array. We may thus specialise the `ObjectArray` class to provide a new access protocol which will be more usable:

```
class CellArray : public ObjectArray {
public:
   CellArray( int x, int y, int d );

   /* access protocol */
   float GetValueAt( int x, int y, int t );
   void SetValueAt(  int x, int y, int t,
                     float v );
};
```

These new functions may easily be implemented in terms of the existing interface, but provide a better, more convenient interface for external objects:

```
float CellArray::GetValueAt(  int x, int y,
                              int t ) {
   Point p(2, x, y);
   Cell *cell;
   Region *r =GetGlobalRegion();

   if(!r->Contains(p))
      return 0.0;
   cell=At(p);
   return cell->GetValueAt(t);
}
```

Note that the new function makes no references to the structure's distribution, or to resolution and partitioning: it is acting, in many respects, as a client to the original `ObjectArray` methods. This is also true in concurrency control terms: the new function do not require additional concurrency control, as it accesses the component's state through an already-protected interface. It also extends the function of the basic access routine by returning zero if the value of a point outside the array is requested.

Processing

Performing the processing in the automaton may make use of the concurrency regulation infrastructure. It is necessary to define a new `Activity` sub-class to act as a worker:

```
class CellActivity : public Activity {
private:
   int simTime;

protected:
   float Evaluator( int x, int y, int t );
   virtual void Body( void );

public:
   CellActivity( int st );
};
```

The `Evaluator()` function performs the actual evaluation function, calculating the value of a point *(x, y)* at a time *t+1*:

```
float CellActivity::Evaluator(   int x, int y,
                                 int t ) {
   CellArray *a =(CellArray *) GetCollection();
   int dx, dy;
   float ave =0.0;

   for(dx=-1; dx<=1; dx++)
      for(dy=-1; dy<=1; dy++)
         ave+=a->At(x+dx, y+dy, t);
   return ave/9.0;
}
```

The `Body()` of the `Activity` simple calls this evaluation function once for each point in the component to which it is attached. The cycle is repeated once for each time step until the requested simulation time has passed:

```
void CellActivity::Body( void ) {
   CellArray *a =(CellArray *) GetCollection();
   Region *r =a->GetGlobalRegion();
   Iterator iter =r->NewIterator();
   Point p(2);
   int px, py;
   float value;

   for(t=1; t<simTime; t++) {
      p=r->First(iter);
      while(!p.Undefined()) {
         px=p.Ordinate(0);   py=p.Ordinate(1);
         value=Evaluator(px, py, t-1);
         a->SetValueAt(px, py, t, value);
         p=r->Next(iter);
      }
   }
}
```

This activity calculates the mean value of a three-by-three locale of points at a time *t-1* and uses this as the value for the centre point at time *t*. It uses the iteration methods of the `Region` class to iterate through all points held locally by the component to which it is attached.

Executing this `Activity` involves attaching it to the components of the `CellArray` and waiting for all copies to terminate:

```
a->AttachStartAndAwait(new CellActivity(SIMTIME));
```

By using the `AttachAndStart()` method instead, the activities could be started without blocking. In either case, a copy of the given `Activity` is attached

to each component of the collection, without explicit involvement of the programmer.

Increasing the Granularity of Access

However, a problem with this architecture is immediately apparent: since it accesses points singly, it requires ten operations – nine reads and one write – to update each point. Moreover, there will be a considerable amount of redundant acquisition of information as points are retrieved several times.

A straightforward solution to the first problem is to implement a new access protocol to the array which acquires a set of points at a single call. This reduces the number of calls required from activity to array by an order of magnitude. Although internally the array may still need to make several resolution requests to acquire all the elements not held locally, it is likely that many of the elements *will* be held locally (ideally all of them). In the original architecture, the client activity benefited from locality by the absence of resolution requests being generated; in the new system, it also benefits by the fact that no method calls are needed to acquire these elements.

A possible implementation of this new access routine is as follows:

```
void CellArray::At(  int x, int y,
                     int t, float v[] ) {
   int i =0;

   for(int dx =-1; dx<=1; dx++)
      for(int dy=-1; dy<=1; dy++)
         v[i++]=At(x+dx, y+dy, t);
}
```

The client activity's evaluation function must be re-written so that it makes use of this "chunking" of access:

```
float CellActivity::Evaluator(   int x, int y,
                                 int t ) {
   CellArray *ca =(CellArray *) GetCollection();
   float ave =0.0;
   float v[9];

   ca->At(x, y, t, v);
   for(int i=0; i<9; i++) ave+=v[i];

   return ave/9.0;
}
```

Although this difference seems trivial, it is crucial. In the first definition of `Evaluator()`, nine `At()` calls were made – all to another object. In the second, there is a single `At()` call in `Evaluator()` and nine in the new definition of `At()`: but these calls will be made to the same object, and so incur no

communications overhead unless they are for points held elsewhere. At the same time, the `CellArray` memory has become more intelligent and specialised towards its application.

Caching

The fact that partitioned memory may present an abstract interface may be exploited to provide features internally which are not visible externally to client objects. Such additional features may be used to improve performance whilst maintaining the same interface – a classic optimisation step.

The intention is to use the same scheme as mentioned above but to hide it within the memory: elements may then be pre-fetched and cached at the component, and may be retrieved from the cache when requested.

The advantage of this approach is that it requires no modification to client objects; the disadvantage is that it re-introduces the communications overheads which the use of external pre-fetching avoids.

The most attractive architecture, then, is to use the "chunked" data access routine to access a component which internally caches remote objects to avoid resolution requests.

In the general case, such caching may be extremely unsafe due to the actions of other clients. There is then a need for a cache-coherence protocol which, in the worst case, degenerates into a form of DSVM implemented entirely in software. However, in many cases, the requirements on consistency are less strict: in the current case, for example, the value of a point forms a history trace which is only added to, never changed. Caching of the most recent part of the trace may thus be performed safely. This is a good example of the ability of the partitioned model's ability to absorb the programmer's knowledge of the application in situations where the details would be difficult to extract automatically.

Rectangular Distribution of Components

A cellular automaton has a very regular pattern, following the arrangement of the space which it models; moreover, the requests for remote elements will only ever be made to neighbouring locales. A further optimisation is to ensure that neighbouring locales are always mapped onto neighbouring processors; or, more precisely, match the distribution of components to the underlying mesh-structured hardware. Such a distribution may be implemented by making use of a novel distribution manager (*i.e.* partition sub-class) which performs this mapping, and may be performed without affecting the shapes of the components[26].

[26]Actually this is not always entirely true. Care must be taken to ensure that no hidden assumptions about the shape of components are introduced into an application. With care, however, this is possible: the code fragments given will work for any distribution.

6.4.3. Example Three: an Inference System

Inference systems are typified by Prolog interpreters. The idea is to solve a query deductively by examining a database containing logical assertions and inference rules.

The main part of the inference system is the database of clauses against which queries are solved. The database is a single conceptual structure which may grow to be very large, and which may be accessed in parallel if required.

We shall represent the database as a large associative memory. We shall make the assertion, however, that clauses are never removed from the database, so it will only grow in size.

Resolving queries uses the unification algorithm and may proceed in parallel: several different processes may attempt to unify different parts of the database, with the results being combined to give the query's final result.

Clauses and Bindings

A clause is the unit of storage within the database. A clause may contain actual values and variables, which the unification process will bind onto appropriate values. Clauses are generated by parsing strings describing the structure of the assertion:

```
struct Binding {
   string variable, value;
};

class Clause : public Object {
private:
   Binding **bindings;
   ...

public:
   Clause( string str );

   virtual HashKey Hash( void );
   boolean Unify( Clause *clause, Binding b[] );
};
```

Calling the `Unify()` method will attempt to match the free variables in the `Clause` supplied and the target `Clause` (the unification algorithm is described in detail most AI books, *e.g.* [35]). The result is `false` if the two clauses cannot be unified, or `true` together with a set of bindings describing the most general unification of the clauses.

The Clause Database

The clause database may be represented as a large associative memory containing clauses. As with the previous example, the basic partitioned structure – in this case a `Dictionary` rather than an `ObjectArray` – could be used: but we

shall bypass this step and develop an interface which is better suited to our application.

The database essentially consists of three operations: assert a fact, retract a fact, and deduce the answer to a query. The first two are simply variations on the theme of access to the associative memory; the last is a more complex operation involving operations on the clauses themselves.

```
class ClauseDatabase :
     public AssociativeCollection {
private:
   ...

public:
   ClauseDatabase( void );

   void Assert( string cl );
   void Retract( string cl );
   int Query( string cl, Binding b[][] );
};
```

The operations accept `string` arguments, creating `Clause` objects to represent them internally: thus the `Clause` object is never manipulated by clients of the database. All the operations must make use of the ability of `Clause` objects to generate a hash key based on their values. The `Hash()` member function must return a hash key from the contents of the `Clause`, and this key will then be used to store and access clauses.

The `Query()` method functions by unifying the query against all members of the database using the clauses' `Unify()` methods, and returns the set of sets of bindings representing all possible unifications across the database.

As an example, consider the following database containing the logic programmer's favourite, a family tree:

```
man(simon)
man(matthew)
man(chris)
woman(pamela)
man(frank)
woman(betty)

parent(chris, simon)
parent(pamela, simon)
parent(frank, matthew)
parent(betty, matthew)

father(?x, ?y) :- man(?x), parent(?x, ?y)
```

Creating this database involves first creating the ClauseDatabase structure, initialising it, and placing clauses into it

```
ClaseDatabase *cd;

cd=new ClauseDatabase();   cd->Initialise();
cd->Assert("man(simon)");
```

and so forth. The assertion of new facts and rules may result in the partitioning of the clause database memory structure, controlled according to properties from the property sheet. The most important property in this case is the size of each component, interms of the number of clauses which it may contain: exceeding this limit causes the component to split. Smaller component sizes result in more splits (which take more time) but allow more concurrent activity to be generated when queries are performed in parallel.

Once built, queries may be made of the database, for example:

```
Binding **bind;

cd->Query("father(chris, simon)", bind);
   true

cd->Query("father(?x, ?y)", bind);
   {?x=chris, ?y=simon}
   {?x=frank, ?y=matthew}
```

(where the lines in italics represent the results of the queries). This is an example of a sequential query regime being run on a distributed-memory system: a simple, but hardly efficient solution. For better performance, especially on large databases, it is necessary to use parallel evaluation of queries.

Parallel Querying

Within Phœnix, a parallel query will obviously involve the creation and attachment of `Activity` objects. For every sub-clause in a query, an activity is created on every component in the clause database to perform the unification. Each activity attempts unification against those clauses which are held by its local component. The result is that a query gives rise to a set of worker activities which will eventually return the set of all possible unifications to the creator of the query.

The activity takes the usual form, but is extended with functions to allow the results of a unification to be acquired:

```
class SubGoalProcessor : public Activity {
private:
   void SetBindings( int b, Binding b[][] );
   ...

public:
   SubGoalProcessor( Clause *gl );

   int GetBindings( Binding b[][] );
};
```

We shall assume that a single function `LocalQuery()` has been added to the definition of `ClauseDatabase` which behaves in exactly the same way as `Query()` but only attempts to unify the query with the clauses held locally. The activity's `Body()` code may be implemented as follows:

```
void SubGoalProcessor::Body( void ) {
   ClauseDatabase *cd =(ClauseDatabase *)
                          GetCollection();
   Clause *goal =GetGoal();
   Binding **b;
   int uni;

   uni=LocalQuery(goal, b);
   SetBindings(uni, b);
}
```

For simplicity, we shall consider the case of a query composed of a single clause, which may be answered using the following algorithm:.

```
Clause *query;
Group *g;
SubGoalProcessor *sgp;
ClauseDatabase *cd;
Binding b[][], c[][];

sgp=new SubGoalProcessor(query);
g=cd->AttachAndStart(sgp);
g->Await();

for(i=0; i<g->Members(); i++) {
   g->Member(i)->GetBindings(c);
   CollectBindings(b, c);
}
delete g;
```

The operation makes use of the Phœnix `Group` object to manipulate a set of activities *en masse*: in this case, a `Group` is created for the activities evaluating all the sub-goals, and the routine waits for all processing to complete before collecting the results together into a single set of bindings.

It is interesting to compare this code fragment against the example presented in §2.4. The structure of the algorithm is exactly the same: an activity is created and attached to a memory. The creator awaits the completon of the query and then amalgamates all the partial answers to form a single result (a set of sets of bindings). There is no reference to the exact amount of concurrency generated, or the location of elements required in processing queries.

Refinements and Optimisations

The refinements which may be appropriate in this application are less obvious than those of the previous example, and it may be better to refer to someone whose studies of parallel logic programming are an end in themselves rather than the current means to an end. – a good example is the work of Wise[121]. A few alternatives suggest themselves, however.

The first is that a predictable hash function, coupled with the predictable nature of clauses, may be exploited. If, for example, all clauses of the form `man(?x)` are hashed with the same prefix, they can be guaranteed to all fall into the same portion of the partition tree and it will only be necessary to create sub-goal processors on a small portion of the collection's components.

At present, evaluating a new query requires the attachment of a new set of activities. An alternative would be to create the evaluating activities along with the components of the database, and to send queries directly to these activities. Specialising the memory interface would allow parallel processing to occur in this fashion without the user's involvement.

6.5. Scalable Memory, Partitioning and Phœnix: a Judgement

The preceding sections have evaluated to work presented in the earlier chapters from four main perspectives:

- how good is the abstraction of scalable memory when creating scalable applications?;
- how good an implementation of scalable memory is the partitioned model?;
- how well does the Phœnix prototype perform?; and
- how easy is it using the model, as embodied in Phœnix, to create scalable applications?

We shall here draw together the conclusions reached in each of these evaluations to form an overall value judgement on the system.

The idea of a scalable memory which may be distributed transparently across the nodes of a multicomputer system is a very powerful one. When compared with other

similar systems it offers advantages over them all, and suffers from few of their disadvantages. The ability of a data structure – a large, user-defined, strongly-typed, scalable collection of elements accessed using meaningful names – to be used as a programming metaphor is a major simplification over other parallel processing paradigms.

The partitioned object model is a good implementation of the scalable memory abstraction. Again, it offers several large advantages over other possible implementations: it is potentially more flexible and tailorable than either Linda or DSVM, and would allow these systems to be implemented in itself if required. There is little or no loss of abstraction when moving to a partitioned system; moreover, a partitioned data structure may be configured very finely to take account of any application-domain knowledge. The collections may also be used to regulate concurrency in a scalable manner.

However, the partitioned model suffers from a handicap when compared to recent Linda and DSVM implementations, in that these systems make use of dedicated hardware. No software-only system can hope to compete against a rival which uses hardware assistance to improve its performance. Partitioning is less susceptible to enhancement *via* hardware, although a machine designed exclusively to support its object model would be a major step in this direction.

The scalability of the system is closely tied to its use of "hints" supplying important parameters. Currently the best values for these hints must be determined experimentally: it would be better if this process were automated to some extent. Although the use of hints is an improvement over the re-compilation required by other systems, it falls short of the goal of truly transparent scalability.

Phœnix is very much a prototype system, and lacks several features which would be needed in a "real" programming system. Its host language's type system and object model are not ideal – although this illustrates that a partitioned system can be implemented in a variety of host languages. The lack of optimisation, in both the host and the RPC system, coupled with the fine decomposition of the Phœnix class hierarchy, means that performance suffers as a result of the large number of method calls made. Flexibility, in this sense, is a drawback.

The model is, however, extremely easy to program in. If applications are built around large data structures – and a sizeable fraction are – then the partitioned model and Phœnix may be used to create a working application in a remarkably short time. This basic applications may then be refined to remove some overheads – effectively performing the reverse of an object-oriented decomposition – in order to improve its performance. There is no reason in principle why such refinement might not result in an application as efficient as one written using a lower-level programming system: in practice, it is doubtful that refinement would be carried so far.

The partitioned model offers good potential for the creation of applications able to tolerate faults in the underlying hardware. Although we did not examine fault tolerance experimentally – due to constraints of time and of available hardware – there is reason to suppose that a partitioned system might be built which was extremely resilient to faults. An experimental verification of this would be interesting future work.

6.6. Résumé

This chapter has sought to evaluate the work set down in the preceding chapters, with a view to deciding whether the ideas presented really constitute a viable approach to scalable parallel programming.

The scalable memory abstract was examined. The abstraction was seen to hide some of the more troublesome characteristics of multicomputer systems, whilst providing a means of regulating the distribution, and hence the concurrency, used in solving problems.

The partitioned object model was seen to be a good implementation of the abstract memory model, with advantages over all the possible alternative implementations – although it also suffers from disadvantages, notably its use of software-based routing of requests.

The Phœnix prototype was examined and was seen to be deficient in several respects. The use of C++ as a host language outlawed some desirable language features – notably true polymorphism and the creation of activities from first-class functions – which were seen to be advantageous to a partitioned system. The system's performance, from a theoretical standpoint, was seen to be free from bottlenecks as long as the application exhibited locality of reference, but its practical performance left much to be desired. As a programming system, however, Phœnix was seen to offer great scope for fast prototyping of applications, followed by stepwise refinement to improve performance as an optimisation step. The need to consider the exact distribution of applications from the start of development was thus removed.

Chapter 7.

Conclusions and Further Work

To live only for some future goal is shallow. It's the sides of the mountain which sustain life, not the top. Here's where things grow. But, of course, without the top you can't have any sides. It's the top which defines the sides. So on we go ... we have a long way to go ... no hurry.

Robert M. Persig, *Zen and the art of motorcycle maintenance*

This thesis has examined some aspects of programming on machines whose hardware and software resources are dynamically variable. Such *scalable* machines offer considerable hope for "future-proof" computing, as their capabilities may be incrementally increased as required to support a larger user base, more computationally complex applications *et cetera*. The central theme has been the development of an abstract programming model, an implementation architecture and a programming environment for creating scalable parallel applications. Such applications are capable of taking advantage of whatever resources are available in the machine at run-time, without re-compilation.

7.1. Réprise

Chapter one explored the concept of scalability in all its forms, concluding that the essence of something's being scalable was its ability to cope gracefully with changes in its fine structure whilst maintaining its gross structure. It then reviewed the existing literature in three areas: machine architectures, operating systems and

programming environments for parallel distributed machines. The focus of this review was on the scalability of the various systems described.

Of all the available multicomputer architectures, only those architectures maintaining a constant number of links per node were seen to be scalable in our sense. Although hypercubic architectures are scalable in some respects, adding additional nodes requires that all the nodes in the system are upgraded with extra links. The design of operating systems for these machines was a challenge, as it requires a substantial degree of information hiding to shield applications from changing number of processing nodes.

Many of the programming environments described in the literature were seen to offer the possibility for creating truly scalable applications, but few procedural or object-oriented systems provided a sufficient degree of architecture-independence for scalable applications to be constructed easily.

Given this, chapter two developed a more abstract view of scalable computing. It began by examining the nature of programming systems in general, concluding that all such systems are built using layers of abstraction. This allowed us to take the view that toolkits built on top of programming languages nevertheless provide the programmer with an abstract machine on which to write applications. For a scalable system, such an abstract machine could be seen as implementing a model of memory and processing which was highly divorced from the underlying hardware and software base: this allows applications to be constructed using a shared-memory model whilst retaining a large degree of scalability.

Chapter three developed an object-oriented implementation of the abstract model of scalable memory, built around the notion of object communities constructed to implement highly scalable memory modules. These communities appear to the programmer as common data structures. They are composed of many objects which interact to present the illusion of a single logical entity. This means that a data structure may be created which is as large as required by an application, unconstrained by architectural features such as the size of individual node memories.

A collection of techniques were developed for the creation of such memory modules. These techniques included the design of a novel hashing algorithm with highly scalable and distributed characteristics. The complexities and access characteristics of the various approaches were analysed, as was the ease with which the structures might be extended to provide application-specific functions. Consideration was also given to the effects of failures within the machine, and the ability of the partitioned model to degrade gracefully.

Chapter four addressed the problems implied for concurrency control and regulation by the introduction of scalable processing. Concurrency control was first dealt with: while no new model or algorithm was presented for concurrency control in a scalable environment, a suitable system was devised from a fusion of the deontic concurrency control logic of DRAGOON with the auxiliary control objects of Arjuna. This fusion is extremely flexible, allowing applications to specify complex concurrency control constraints simply.

Concurrency regulation was introduced by the observation that a multiple-worker approach is very well-suited to a system with shared memory. The number of tasks may be controlled automatically by the partitioned collection itself, with all

co-location and replication occurring transparently. Support may be provide for termination detection, speculative concurrency *et cetera*.

No new programming system is complete without an experimental implementation, however simple. Chapter five described *Phœnix*, an implementation of the partitioned object model in C++. Phœnix follows closely the descriptions of chapters three and four, with some restrictions introduced by the syntax and semantics of C++. The implementation is basically a virtual object space within which partitioned collections may be built: all the features of the partitioned model are supported. Considerable emphasis was placed on the construction of a "clean" class hierarchy with features for easy extension.

Chapter six presented the results of the evaluation performed on both the partitioned model and Phœnix. The use of scalable memory was judged to be a good abstraction for scalable computation, and the partitioned object model showed many advantages over the other possible implementation strategies. The class of algorithms suited to partitioned computation was identified.

The Phœnix prototype was analysed in detail. It was found to be a good implementation of the partitioned model in terms of its theoretical complexity and the support which it provided for the creation of parallel applications. Some case studies were used to demonstrate the ease with which a Phœnix application may be first prototyped and then refined into a more suitable form. The practical performance of Phœnix, however, was shown to be prohibitive, as too much overhead was introduced by the underlying system and by the fine level of decomposition performed to maximise the system's flexibility: ironically, a good object-oriented decomposition is what destroys the system's performance. Identifying these causes, however, lead to methods by which the overheads might be eliminated.

Some of the deficiencies of Phœnix were also seen to derive from the use of C++ as a host language. The major problems with C++ in this context were identified.

Contributions

The research described has, it is believed, made the following contributions to the field of programming language design for scalable systems:

- a thorough discussion of the concept of *scalability* in all its manifestations – from hardware, through algorithms, to applications;
- a novel parallel programming model which views *memory*, rather than *processing*, as the central component of a parallel system, around which processing may be centred in a scalable and flexible manner;
- a collection of implementation techniques which allow true shared-memory computing to take place in a distributed-memory environment, using the abstract programming model;

- a novel algorithm for building large hashed data structures which is scalable, distributed and free from bottlenecks; and
- some insights on language and operating system design for the next generation of scalable systems, in the light of the experience gained in the course of the research.

7.2. Further Work

The work presented in this thesis is, of course, not an end in itself. It has suggested several course of study for the future.

A Language for Partitioned-model Programming

The design of the partitioned object model has been an experiment in the design of a programming *system*, differing from a programming *language* only in as much as that it may be implemented in any one of a number of possible host languages. By identifying the features in the host which most act to the benefit (or detriment) of the partitioned model (§6.2.4), it is possible to draw some conclusions about the form of an "ideal" scalable parallel programming language, in which the ideas of scalable memory are embedded.

The model rests on the initial contention that the distributed nature of multicomputer memory should not propagate to the programmer, in the sense that it should not make the construction of applications more complicated unnecessarily. The way in which it *should* propagate is by allowing more efficient, more parallel, more scalable applications to be constructed. This allows the power of multicomputer architectures to be harnessed whilst hiding the difficulties which they introduce.

A language based on the partitioned model presents memory as a collection of typed abstract memory modules, to which activities are attached. This means that the methods by which data is structured within the language must be very flexible. In particular, it is important that data structures may be introduced using type parameterisation or polymorphism. This greatly increases the generality of memory modules.

A more important contribution comes from the provision of first-class functions. The ability to build new concurrent activities *ex nihilo* at run-time is something which is seriously missed in Phœnix: although providing no essential advantages in terms of expressibility, first-class functions make the creation of general-purpose tools very much simpler. (Interestingly, first-class code is a feature of Smalltalk – the first object-oriented language – but not of many of its descendents.)

Once we have the ability to create worker tasks "on the fly," we reduce the number of classes required by an application. The code which an activity is to execute is simply a parameter – especially when the types of elements contained by the activity are also parameterised.

Indeed, the ability to pass code to objects as parameters raises some questions about the nature of objects. What is the distinction between a method and a function

passed as a parameter? Does it make sense to change the text of a method dynamically? These are interesting directions for the future.

A Better Implementation

The most pressing need is for an implementation of the partitioned model which is better than Phœnix in terms both of performance and of language design.

The factors contributing most to Phœnix' poor performance were identified in §6.3.3: the poor implementation of remote procedure call, coupled with the overheads involved in dynamically creating class servers. Optimising RPC is a difficult and time-consuming task, although experience on other projects indicates that it is feasible. The size of code servers may be reduced by allowing some portions of the server – the RPC management especially – to be shared by all the class servers on a node. Both these improvements would have a considerable effect.

Another alternative is to implement a virtual machine interpreter running on each node. The code for the various objects is then simply passed to this interpreter. Such an organisation is more portable, but incurs a penalty through the use of interpretation. Such a strategy interacts well with the ideas of first-class functions and other high-level constructs, however.

Applications Experience

It is difficult to evaluate the success or usability of a new programming approach without developing a significant amount of code with it, and naturally it is difficult to obtain such experience in a limited time. There is also a natural reluctance on the part of third parties to invest time programming an experimental system.

Phœnix is not a practical vehicle with which to perform large-scale programming, but with a better implementation it would be possible – and eminently desirable – to obtain more practical experience with the construction of realistic partitioned-model applications.

In particular, there is a need to experiment with the configuration of scalable applications. There area great many factors which must be balanced to achieve an optimal configuration and, although some small experiments in this line were performed, meaningful results could only come from prolonged use of the system.

Some form of tool support for generating configurations, or for assessing the effects of changes, would also be beneficial. The latter case could take the form of additional instrumentation of partitioned collections (activated by a property) which analysed resolution traffic. It is then possible to determine the effects of different property values by examining (for example) the different number of remote requests generated by activities

Fault Tolerance

Fault tolerance in the partitioned model was discussed briefly in §3.5, although it was not embodied into Phœnix; nor is the Transputer a suitable testbed.

Implementing a partitioned system on a different architecture would make it possible to test the ability of the partitioned model to withstand faults in its hardware and software. The techniques outlined in §3.5, although quite minor, offer the potential for a highly fault-tolerant memory architecture to be constructed.

7.3. Conclusion

The partitioned object model has been shown to be a possible programming environment for creating highly parallel, highly distributed, highly scalable applications. It has several advantages over other systems.

The most notable advantages are in the areas of abstraction and refinement. The model provides a very good abstraction over a scalable machine, hiding the issues of data distribution and concurrency regulation from the programmer. The programmer is presented with a system based on large scalable memory modules which manage distribution of data elements automatically.

In contrast to many other systems, the partitioned model still allows the programmer to exert a marked degree of control over data distribution. This control comes from two sides: the ability to supply run-time parameters, and the ability to provide totally new distribution controller. The former allows factors such as component size, tree structure *et cetera* to be controlled at run-time, without re-compiling the application and possibly making use of automatic tools; the latter means that new distribution algorthms may be introduced.

The abstraction over the hardware view of memory allows programmers to create applications very quickly using the partitioned model, without being immediately concerned about distribution. The exact distribution of an application may be left until it is debugged and working, at which point different distribution patterns may be applied *a posteriori* if required. The distribution of elements has no semantic relevance, and so need only be considered where performance is an issue. This is a major simplification over other parallel programming systems.

Some of the techniques developed may have uses outside the current work. The best example of this is the scalable hashing algorithm, which might form the basis for a distributed name server or a database engine. The algorithm might also be used to manage data stored on disc: by decomposing both the data and index table of a hashed data structure, it would allow better control over what parts of the structure are brought into memory.

Concurrency concerns the programmer in two ways: *controlling* concurrent activity to avoid interference and *regulating* it to determine how many concurrent activities to deploy as part of an application. The first is tackled on a per-object basis by using concurrency control objects coupled with a deontic logic for the specification of constraints; the latter is addressed using the memory infrastructure.

The partitioned model contains support for the multiple-worker paradigm for concurrency control. Concurrent activities may be created which access a scalable memory to transform it in some way. These activities are replicated according the memory's size and distribution, with the scalable memory itself determining how many replicas to create and where to locate them. An application may thus treat parallelism as indeterminate: it specifies *when* parallel activity is to occur, but does

not specify *how many* activities should be created or *where* they should reside. Memory organisation controls concurrency, regulating the number of activities automatically. This allows concurrency to scale alongside memory.

Although not considered at length, partitioned data structures might form a good basis for a fault-tolerant memory architecture. Their abstract nature means that fault tolerance may be added, to some extent, internally, without being visible to clients.

Applications may be written using a very simple abstract model – that of scalable memory and automatic concurrency regulation – and this is a major simplification over other systems. The penalty is increased execution times due to the amount of communication involved in resolving requests for data.

If flexibility is the great strength of the partitioned model, then performance is its major weakness. Although all the structures and algorithms used in the system are completely scalable, the fine-grained decomposition of applications into objects means that efficient performance in practice is governed by the efficiency of the low-level communications system; nor is the model as amenable as other systems to the use of hardware accelerators.

The ability to incrementally refine applications' distribution patterns, however, may be used to reduce unnecessary communications. In the limit, this refinement might result in a system with exactly the same properties as one constructed in (for example) Occam, with manual data placement and concurrency regulation. An application might be prototyped using the partitioned model and then refined into another, more efficient form.

A long-term aim is to investigate the creation of a practical programming system based around the ideas in this thesis, with the appropriate language support and including support for fault tolerance and hardware assistance. This would lead to a high-level, practical approach to scalable parallel programming.

References

[1] Gul Agha, "*Actors: a model of concurrent computation in distributed systems,*" MIT Press (1986).

[2] Sudhir Ahuja, N. Carriero and D. Gelernter, "*Linda and friends,*" IEEE Computer 19(**8**) (August 1986) pp.26-34.

[3] R.J. Allan, "*Numerical algorithm libraries for multicomputers,*" Advanced research computing group, SERC Daresbury nuclear physics laboratory (1990).

[4] Tom Anderson and Pete Lee, "*Fault tolerance: principles and practice,*" Prentice-Hall (1981).

[5] Artificial Intelligence Ltd., "*Strand-88 technical descripion,*" (August 1989). Admiralty release.

[6] Martin C. Atkins, "*Implementation techniques for object.oriented systems,*" YCST 90/01, Department of computer science, University of York (June 1989). D.Phil. dissertation.

[7] Colin Atkinson, "*An object-oriented language for software re-use and distribution,*" Department of computing, Imperial College of Science and Technology (February 1990). Ph.D. dissertation.

[8] Paul B. Austin, Kevin A. Murray and Andy J. Wellings, "*The design of a scalable parallel operating system,*" YCS 129, Department of Computer Science, University of York (November 1989).

[9] Paul B. Austin, Kevin A. Murray and Andy J. Wellings, "*File system caching in large point-to-point networks,*" YCS 139, Department of computer science, University of York (September 1990).

[10] Paul B. Austin, Kevin A. Murray and Andy J. Wellings, "*Early experiences with the construction of a scalable parallel operating system,*" YCS 153, Depertment of computer science, University of York (November 1990).

[11] Paul B. Austin, "*Towards a file system for a scalable parallel computing engine,*" YCST 92/01, Department of computer science, University of York (March 1992). D.Phil. dissertation.

[12] Maurice J. Bach, "*The design of the Unix operating system,*" Prentice-Hall (1986).

[13] John Backus, "*Can programming be liberated from the Von Neumann style?,*" Communications of the ACM 21(**8**) (August 1978) pp.613-641. 1977 Turing Award Lecture.

[14] Henri Bal, Andrew S. Tanenbaum and M. Frans Kaashoek, "*Orca: a language for distributed programming,*" ACM SIGPLAN Notices 25(**5**) (May 1990) pp.17-24.

[15] Bell Labs, "*Plan 9 programmer's manual,*" (1992). Unpublished document, supplied with the pre-release of Plan 9.

[16] J.K. Bennett, J.B. Carter and W. Zwaenpoel, "*Munin: distributed shared memory based on type-specific memory coherence,*" ACM SIGPLAN Notices 25(**3**) (March 1990) pp.168-176. Proceedings of the 2nd ACM symposium on principles and practice of parallel programming.

[17] B.N. Bershad *et alia*, "*An open system for building parallel programming systems,*" ACM SIGPLAN Notices 23(**9**) (September 1988) pp.1-9. Proceedings of the SIGPLAN '88 conference on parallel programming: experience with applications, languages and systems.

[18] A.D. Birrell and B.J. Nelson, "*Implementing remote procedure call,*" ACM Transactions on Computer Systems 2(**1**) (February 1984) pp.39-59.

[19] R. Bjornsen, N. Carriero, D. Gelernter and J. Leichter, "*Linda, the portable parallel,*" YALE/DCS/RR-520, Yale University (1988).

[20] A, Black, N. Hutchinson, E. Jul and H. Levy, "*Object structure in the Emerald system,*" ACM SIGPLAN Notices 21(**11**) (November 1986) pp.78-86. OOPSLA '86.

[21] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, "*Distribution and abstract types in Emerald,*" IEEE Transactions on software engineering 13(**1**) (January 1987) pp.65-76.

[22] H. Boehm, A. Demers and J. Donahue, "*An informal description of Russell,*" 80-430, Department of Computer Science, Cornell University (1980).

[23] Grady Booch, "*Software engineering with Ada,*" Addison-Wesley (1987).

[24] Grady Booch and Michael Vilot, "*The design of the C++ Booch components,*" ACM SIGPLAN Notices 25(**10**) (October 1990) pp.1-11. OOPSLA '90.

[25] Steve Bourne, "*The Unix system,*" Addison-Wesley (1983).

[26] Per Brinch Hansen, "*The programming language Concurrent Pascal,*" IEEE Transactions on Software Engineering 1(**2**) (June 1975) pp.199-207.

[27] Per Brinch Hansen, "*Joyce: a programming language for distributed systems,*" Software - practice and experience 17(**1**) (January 1987) pp.29-50.

[28] David Bruce, "*A strongly-typed approach to parallel systems,*", pp.57-59 in Collected position papers of the BCS workshop on abstract machine models for highly parallel computers, volume 2, University of Leeds (25-27th March, 1991).

[29] Tim Budd, "*A Little Smalltalk,*" Addison-Wesley (1987).

[30] Paul Butcher, "*Lucinda: an overview,*" ACM SIGPLAN Notices 26(**8**) (August 1992) pp.90-100.

[31] David Catton (June 1990). Personal communication.

[32] Luca Cardelli and Peter Wegner, "*On understanding types, data abstraction and polymorphism,*" ACM Computing surveys 17(**4**) (December 1985) pp.471-522.

[33] N. Carriero and D. Gelernter, "*Applications experience with Linda,*" ACM SIGPLAN Notices 23(**9**) (September 1988) pp.173-187. Proceedings of the ACM SIGPLAN conference on parallel programming: experience with applications, languages and systems.

[34] N. Carriero and D. Gelernter, "*How to write parallel programs: a guide for the perplexed,*" ACM Computing surveys 21(**3**) (September 1989).

[35] E. Charniak and D.M. McDermott, "*Introduction to artificial intelligence,*" Addison-Wesley (1985).

[36] A.A. Chien and W.J. Dally, "*Concurrent Aggregates,*" ACM SIGPLAN Notices 25(**3**) (March 1990) pp.187-196. Proceedings of the 2nd ACM symposium on principles and practice of parallel programming.

[37] Alonzo Church, "*Calculi of lambda-conversion,*" Princeton University Press (1941).

[38] L. Clarke and G. Wilson, "*Tiny: an efficient routing harness for the Inmos Transputer,*" Concurrency – practice and experience 3(**3**) (July 1991) pp.221-245.

[39] W.F. Clocksin and C.S. Mellish, "*Programming in Prolog, 2e,*" Springer-Verlag (1984).

[40] Scott Danforth and Chris Tomlinson, "*Type theories and object-oriented programming,*" ACM Computing surveys 20(**1**) (March 1988) pp.29-72.

[41] E.W. Dijkstra, "*Co-operating sequential processes,*" in Programming languages, ed. F. Genuys, Academic Press (1968).

[42] Graeme N. Dixon, "*Object management for persistence and reliability,*" TR 276, Computing laboratory, University of Newcastle upon Tyne (December 1988). Ph.D. dissertation.

[43] G.D. Dixon, S.K. Shrivastava, F. Hedayati, G.D. Partington and S.M. Wheater, "*A technical overview of Arjuna: a system for reliable distributed computing,*" TR 262, Computing laboratory, University of Newcastle upon Tyne (July 1988).

[44] G.N. Dixon, G.D. Parrington, S.K. Shrivastava and S.M. Wheater, "*The treatment of persistent objects in Arjuna,*" TR 283, Computing laboratory, University of Newcastle upon Tyne (June 1989).

[45] R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, "*Extendible hashing a fast access method for dynamic files,*" ACM Transactions of database systems 4(**3**) (September 1979) pp.315-344.

[46] M.J. Flynn, "*Very high-speed computing systems,*" Proceedings of the IEEE 54 (1966) pp.1901-1909.

[47] James D. Foley, Andries van Dam, Steven K. Feiner and John F. Hughes, "*Computer graphics: principles and practice,*" Addison-Wesley (1990).

[48] Ian Foster and Stephen Taylor, "*Strand: new concepts in parallel programming,*" Prentice-Hall (1990).

[49] Steve Frank, "*Virtual memory to ALLCACHE memory,*" in Proceedings of the virtual shared memory symposium, Centre for novel computing, University of Manchester (17-18 September 1992).

[50] E. Fredkin, "*Trie memory,*" Communications of the ACM 3(**9**) (September 1960) pp.439-499.

[51] N.H. Gehani and W.D. Roome, "*Concurrent C,*" Software - practice and experience 16(**9**) (September 1986) pp.821-844.

[52] D. Gelernter, "*Generative communication in Linda,*" ACM Transactions on Programming Languages and Systems 7(**1**) (January 1985) pp.80-112.

[53] D. Gelernter, "*Getting the job done,*" Byte 13(**12**) (November 1988).

[54] Adele Goldberg and David Robson, "*Smalltalk-80: the language and its implementation,*" Addison-Wesley (1985).

[55] J. Gustafson, G.R. Montry and R.E. Benner, "*Development of parallel methods for a 1024-processor hypercube,*" SIAM Journal of scientific and statistical computing 9(**4**) (July 1988) pp.609-638.

[56] R.H. Halstead, "*MultiLisp: a language for concurrent symbolic computation,*" ACM Transactions on Programming Languages and Systems 7(**4**) (October 1985) pp.501-538.

[57] High Performance Fortran Forum, "*High Performance Fortran language specification, Draft version 0.4,*" (November 1992).

[58] Daniel Hillis, "*The Connection Machine,*" MIT Press (1985).

[59] C.A.R. Hoare, "*Communicating Sequential Processes,*" Prentice-Hall (1985).

[60] C.A.R. Hoare, "*Monitors: an operating system structuring concept,*" Communications of the ACM 17(**10**) (October 1974) pp.549-.

[61] P. Hudak, "*Conception, evolution and application of functional programming languages,*" ACM Computing surveys 21(**3**) (June 1989) pp.359-411.

[62] A.D. Hutcheon and A.J. Wellings, "*The virtual node approach to designing distributed Ada programs,*" Ada User 9(**Supplement**) (December 1988) pp.35-42.

[63] N.C. Hutchinson, "*Emerald: an object-based language for distributed programming,*" 87-01-01, University of Washington at Seattle (1987). Ph.D. dissertation.

[64] Leah H. Jamieson, "*Characterizing parallel algorithms,*", pp.65-100 in The characteristics of parallel algorithms, ed. L.H. Jamieson, D.B. Gannon, R.J. Douglass, MIT Press (1987).

[65] Michael Jones and Richard Rashid, "*Mach and Matchmaker: kernel and language support for object-oriented distributed systems,*" ACM SIGPLAN Notices 21(**11**) (November 1986). OOPSLA '86.

[66] David Jourdan, John McDermid and Ian Toyn, "*CADiZ – computer aided design in Z,*", pp.93-104 in Proceedings of the Z user workshop, Oxford 1990, ed. J.E. Nicholls, Springer-Verlag (1991).

[67] E. Jul, "*Object mobility in a distributed object-oriented system,*" 88-12-06, University of Washington at Seattle (1988). Ph.D. dissertation.

[68] E. Jul, H. Levy, N. Hutchinson and A. Black, "*Fine-grained mobility in the Emerald system,*" ACM Transactions on Computer Systems 6(**1**) (February 1988) pp.109-133.

[69] Samuel N. Kamin, "*Programming languages: an interpreter-based approach,*" Addison-Wesley (1990).

[70] Donald Knuth, "*The art of computer programming,*" Addison-Wesley (1973). (3 volumes).

[71] C. Koelbel, P. Mehrotra and J.V. Rosendale, "*Supporting shared data structures on distributed memory architectures,*" ACM SIGPLAN Notices 25(**3**) (March 1990) pp.177-186. Proceedings of the 2nd ACM SIGPLAN symposium on principles and practice of parallel programming.

[72] Glenn Krasner, "*Smalltalk-80: bits of history, words of advice,*" Addison-Wesley (1983).

[73] Per-Åke Larson, "*Dynamic hashing,*" BIT 18 (1978) pp.184-201.

[74] R.M. Lea, "*ASP: a cost-effective parallel microcomputer,*" IEEE Micro 8(**5**) (October 1989) pp.10-29.

[75] William Leler, "*Linda meets Unix,*" IEEE Computer 23(**2**) (February 1990) pp.43-54.

[76] Kai Li, "*Shared virtual memory on loosely coupled multiprocessors,*" Research report 492, Department of computer science, Yale University (September 1986). Ph.D. dissertation.

[77] K. Li and P. Hudak, "*Memory coherence in shared virtual memory systems,*" ACM Transactions of computer systems 7(**4**) (November 1989) pp.243-271.

[78] K. Li and R. Schaefer, "*A hypercube shared virtual memory,*", pp.123-131 in Proceedings of the international parallel processing conference, volume 1 (August 1989).

[79] B. Liskov and L. Shira, "*Promises: linguistic support for efficient asynchronous procedure calls,*" ACM SIGPLAN Notices 23(**7**) (July 1988). Proceedings of the ACM conference on programming language design and implementation.

[80] Witold Litwin, "*Virtual hashing: a dynamically changing hashing,*", pp.517-523 in Proceedings of the 4th international conference on very large data bases, ed. S. Bing Yao (1978).

[81] S.E. Lucco, "*Parallel programming in a virtual object space,*" ACM SIGPLAN Notices 22(**12**) (December 1987) pp.26-34. OOPSLA '87.

[82] Craig Lund, "*Goal of a new machine*," Parallelogram **46** (July/August 1992) pp.8-10.

[83] E. Lusk, D.H.D. Warren and S. Haridi *et alia*, "*The Aurora or-parallel Prolog system,*" New generation computing 7(**2,3**) (1990) pp.243-271. Special issue on parallel logic programming.

[84] Jeff Magee and Naranker Dulay, "*MP: a programming environment for multicomputers,*", pp.1-16 in Programming environments for parallel computing, ed. Nigel Topham, Roland Ibbett and Thomas Bemmerl, North Holland Elsevier (1992).

[85] David P. Mallon and Peter M. Dew, "*Communicating through shared objects,*", pp.181-191 in Programming environments for parallel computing, ed. Nigel Topham, Roland Ibbett and Thomas Bemmerl, North Holland Elsevier (1992).

[86] David May, "*Occam-2 language definition,*" Technical note, Inmos (13 February 1987).

[87] Isi Mitrani, "*Modelling of computer and communications systems,*" Cambridge University Press (1987).

[88] Sape J. Mullender and Andrew S. Tanenbaum, "*The design of a capability-based operating system,*" Computer Journal 29(**4**) (April 1986).

[89] Sape J. Mullender, "*Amoeba – high-performance distributed computing,*", pp.17-26 in European Unix systems user group Spring conference (April 1989).

[90] K.A. Murray and A.J. Wellings, "*Issues in the design and implementation of a distributed operating system for a network of Transputers*," in Proceedings of EUROMICRO '88.

[91] Kevin A. Murray and Andy J. Wellings, "*Wisdom: a distributed operating system for Transputers,*" Computer Systems Science and Engineering 5(**1**) (January 1990) pp.13-20.

[92] Kevin A. Murray, "*Wisdom: the foundation of a scalable parallel operating system,*" YCST 90/02, Department of Computer Science, University of York (February 1990). Ph.D. dissertation.

[93] T. Nakajima, Y. Yokote, M. Tokoro, S. Ochiai and T. Nagamatsu, "*DistributedConcurrentSmalltalk: a language and system for the interpersonal environment,*" ACM SIGPLAN Notices 24(**4**) (April 1989)

pp.43-45. Proceedings of the SIGPLAN workshop on object-based concurrent programming.

[94] B.J. Nelson, "*Remote procedure call,*" CMU-CS-81-119, Carnegie-Mellon University (1981). Ph.D. dissertation.

[95] Cherri M. Pancake and Donna Bergmark, "*Do parallel languages respond to the needs of scientific programmers?,*" IEEE Computer 23(**12**) (December 1990) pp.13-23.

[96] Graham D. Parrington, "*Management of concurrency in a reliable object-oriented computing system,*" TR 277, Computing laboratory, University of Newcastle upon Tyne (July 1988). Ph.D. dissertation.

[97] Rob Pike, Dave Presotto, Ken Thompson and Howard Trickey, "*Plan 9 from Bell Labs,*", pp.1-9 in Proceedings of the Summer UKUUG conference (1990).

[98] Dick Pountain, "*Parallelizing Prolog,*" Byte 13(**12**) (November 1988) pp.387-394.

[99] Dave Presotto, Rob Pike, Ken Thompson and Howard Trickey, "*Plan 9: a distributed system,*", pp.43-50 in Proceedings of the Spring EurOpen conference (May 1991).

[100] Sanjay Raina, David H.D. Warren and James Cownie, "*Shared virtual memory on the Computing Surface* via *the data diffusion machine,*", pp.137-141 in Proceedings of the 13th technical meeting of the Occam User Group – extended abstract of papers (18-20th September 1990).

[101] John H. Reppy, "*Concurrent programming with events – the Concurrent ML manual, v.0.9,*" Department of computer science, Cornell University (November 1990).

[102] R.D. Rettberg, W.R. Crowther, P.P. Carvey and R.S. Tomlinson, "*The Monarch parallel processor hardware design,*" IEEE Computer 23(**4**) (April 1990) pp.18-30.

[103] Robert Schlieffer and James Gettys, "*The X Window system,*" Digital press (1990).

[104] Karsten Schwan and Win Bo, "*Topologies – distributed objects on multicomputers,*" ACM Transactions on computer systems 8(**2**) (May 1990) pp.111-157.

[105] J.M. Spivey, "*The Z notation: a reference manual,*" Prentice-Hall (1989).

[106] P.D. Stotts, "*A comparative survey of concurrent programming languages,*" ACM SIGPLAN Notices 17(**9**) (September 1982) pp.76-87.

[107] R.F. Stone, "*Reliable computer systems – a review,*" YCS 110, Department of Computer Science, University of York (January 1989).

[108] Bjarne Stroustrup, "*The C++ programming language,*" Addison-Wesley (1987).

[109] Andrew S. Tanenbaum and Robbert van Renesse, "*Distributed operating systems,*" ACM Computing Surveys 17(**4**) (December 1985).

[110] Andrew S. Tanenbaum, "*Operating systems: design and implementation,*" Prentice-Hall (1987).

[111] Andrew S. Tanenbaum, "*Modern operating systems,*" Prentice-Hall (1992).

[112] Andrew S. Tanenbaum, M. Frans Kaashoek and Henri Bal, "*Parallel programming using shared objects and broadcasting,*" IEEE Computer 25(**8**) (August 1992) pp.10-19.

[113] L.G. Valiant, "*Bulk synchronous parallel computers,*" TR-08-89, Aiken computation laboratory, Harvard University (1989).

[114] Chris Wadsworth, "*Virtual shared memory: the good, the bad and the unknown,*" Parallel processing group note 85, Informatics department, SERC Rutherford Appleton Laboratory (September 1992).

[115] David Walker, "*pi-calculus semantics of object-oriented programming languages,*" ECS-LFCS-90-122, Laboratory for foundations of computer science, Department of computer science, University of Edinburgh (October 1990).

[116] Peter Wegner, "*Dimensions in object-based language design,*" ACM SIGPLAN Notices 22(**12**) (December 1987) pp.168-182. OOPSLA '87.

[117] Åke Wikström, "*Functional programming in Standard ML,*" Prentice-Hall (1987).

[118] Greg Wilson, "*The life and times of cellular automata,*" New Scientist (8th October 1988) pp.44-47.

[119] Greg Wilson, "*Improving the performance of generative communication systems by using application-specific mapping functions,*", pp.117-130 in Proceedings of the workshop on Linda-like systems and their implementation, University of Edinburgh (24 June, 1991).

[120] Niklaus Wirth, "*Algorithms + data structures = programs,*" Prentice-Hall (1976).

[121] Michael J. Wise, "*Prolog multiprocessors,*" Prentice-Hall (1987).

[122] Yasuhiko Yokote and Mario Tokoro, "*The design and implementation of ConcurrentSmalltalk,*" ACM SIGPLAN Notices 21(**11**) (November 1986) pp.331-340. OOPSLA '86.

[123] Pamela Zave, "*A compositional approach to multiparadigm programming,*" IEEE Software (September 1989) pp.15-25.

Appendix A.

A Formal Treatment of Partitioning

Any new technique benefits from a formal treatment: it allows the properties and interactions of a system to be presented unambiguously. In this appendix we present a formal treatment of the partitioning technique introduced in chapter 3, using the Z notation[105]. The specification was generated using the University of York's CADiZ type checker and formatter for Z[66].

Objects and Collections

We shall begin by defining a rudimentary object space. Values stored in collections are identified by abstract identifiers.

$$[VALUE\_ID]$$

There is a single identifier which represents the "null" object.

$$NULLVALUE\_ID : VALUE\_ID$$

Values are identified within collections by names – array index tuples, hash keys, edge labels *et cetera*.

$$[VALUE\_NAME]$$

A collection is a community of component and partition objects. These objects may also be represented in the system using abstract identifying values.

$$[COMPONENT\_ID, PARTITION\_ID]$$

Similarly, there exist privileged objects identifiers representing the "null" component and partition object.

$$NULLCOMPONENT\_ID : COMPONENT\_ID$$
$$NULLPARTITION\_ID : PARTITION\_ID$$

Local storage within a component, where values are stored, may be seen as a (finite partial) function from a values names to value identifiers.

$$STORAGE == VALUE\_NAME \pfun VALUE\_ID$$

A component object is an instance of an abstract type having a system-unique identifier, a partition to which it is attached, a (possibly infinite) set of names which it may store locally, and a binding from some or all of those names to values.

COMPONENT

$$cid : COMPONENT\_ID$$
$$part : PARTITION\_ID$$
$$localNames : \mathbb{P}\ VALUE\_NAME$$
$$local : STORAGE$$

$$\mathrm{dom}\ local \subseteq localNames$$

Partition, similarly, are named instances of another abstract type. One element of this type is a disjoint union of possible descendents of a partition, which

may be components or other partitions.

$$
\begin{aligned}
&\mathrm{TREENODE\_ID} ::= \\
&\quad \mathrm{ComponentTreeNodeID} \ll \mathrm{COMPONENT\_ID} \gg \mid \\
&\quad \mathrm{PartitionTreeNodeID} \ll \mathrm{PARTITION\_ID} \gg \mid \mathrm{NullTreeNodeID}
\end{aligned}
$$

A partition itself is composed of a unique identifier, a parent partition's identifier, and a sub-mapping table of this disjoint union, accessed using value names.

PARTITION

$$
\begin{aligned}
&\mathrm{pid} : \mathrm{PARTITION\_ID} \\
&\mathrm{parent} : \mathrm{PARTITION\_ID} \\
&\mathrm{submap} : \mathrm{VALUE\_ID} \pfun \mathrm{TREENODE\_ID}
\end{aligned}
$$

The identifiers for the objects, held within the abstract values, act to all intents and purposes as object names (or pointers to) single shared instances of the abstract type. Such names may be dereferenced uniquely by ensuring that only at most one object has the given identifier. The "system" of partitioned collections is essentially a set of component and partition objects – for simplicity we shall consider stored values to lie outside the system being specified. Every object within the system has a unique name.

SystemObjects

$$
\begin{aligned}
&\mathrm{componentObjects} : \mathbb{F}\ \mathrm{COMPONENT} \\
&\mathrm{partitionObjects} : \mathbb{F}\ \mathrm{PARTITION} \\
&\hline
&\forall\, \mathrm{c1}, \mathrm{c2} : \mathrm{COMPONENT} \mid \\
&\quad \mathrm{c1} \in \mathrm{componentObjects} \land \mathrm{c2} \in \mathrm{componentObjects} \bullet \\
&\qquad \mathrm{c1.cid} = \mathrm{c2.cid} \Leftrightarrow \mathrm{c1} = \mathrm{c2} \\
&\forall\, \mathrm{p1}, \mathrm{p2} : \mathrm{PARTITION} \mid \\
&\quad \mathrm{p1} \in \mathrm{partitionObjects} \land \mathrm{p2} \in \mathrm{partitionObjects} \bullet \\
&\qquad \mathrm{p1.pid} = \mathrm{p2.pid} \Leftrightarrow \mathrm{p1} = \mathrm{p2}
\end{aligned}
$$

This completes the description of the object space.

Collections and the System

A collection may be seen as an abstract value composed of a number of components and partitions and having a set of value names which it may resolve.

COLLECTION

componentIDs : $\mathbb{F}$ COMPONENT_ID
partitionIDs : $\mathbb{F}$ PARTITION_ID
resolvable : $\mathbb{P}$ VALUE_NAME

The system is defined as a set of collections. The members of a collection possess a structural relationship to one another, and not to members of any other collection.

SystemCollections

collections : $\mathbb{F}$ COLLECTION

$$
\begin{aligned}
&\forall\, coll : COLLECTION \mid coll \in collections \bullet \\
&\quad \#coll.componentIDs = 1 \land coll.partitionIDs = \varnothing \lor \\
&\quad \#coll.componentIDs > 1 \land \\
&\quad (\forall\, c : COMPONENT \mid c.cid \in coll.componentIDs \bullet \\
&\qquad c.part \in coll.partitionIDs) \land \\
&\quad (\exists_1 r : PARTITION \mid r.pid \in coll.partitionIDs \bullet \\
&\qquad r.parent = NULLPARTITION\_ID) \land \\
&\quad (\forall\, p : PARTITION \mid p.pid \in coll.partitionIDs \bullet \\
&\qquad p.parent = NULLPARTITION\_ID \lor \\
&\qquad (\exists\, q : PARTITION \mid \\
&\qquad\quad q.pid \in coll.partitionIDs \land q \neq p \bullet \\
&\qquad\qquad p.parent = q.pid))
\end{aligned}
$$

For partitions, there is a notion of a partition being "above" another if it or one of its descendents holds a reference to that partition in its submap table.

_ above _ : PARTITION $\leftrightarrow$ PARTITION

$$
\begin{aligned}
&\forall\, p1, p2 : PARTITION \bullet \\
&\quad p1 \ above \ p2 \iff \\
&\qquad p2.parent = p1.pid \lor \\
&\qquad (\exists\, p3 : PARTITION \bullet \\
&\qquad\quad p3 \ above \ p2 \land p3.parent = p1.pid)
\end{aligned}
$$

Partitions and components are related by the fact that a given partition may contain in its submap table an entry relating to a local element of a component. This partition is said to "resolve" the component; moreover, by virtue of the partitioning technique, any partition which is below a partition which can resolve a particular component can itself resolve that component.

$$\_ \text{ resolves } \_ : \text{PARTITION} \leftrightarrow \text{COMPONENT}$$

$$\begin{aligned}
&\forall p : \text{PARTITION}; c : \text{COMPONENT} \bullet \\
&\quad p \text{ resolves } c \Leftrightarrow \\
&\qquad \text{ComponentTreeNodeID } c.\text{cid} \in \text{ran } p.\text{submap} \lor \\
&\qquad (\exists q : \text{PARTITION} \bullet \\
&\qquad\quad p \text{ above } q \land \\
&\qquad\quad \text{ComponentTreeNodeID } c.\text{cid} \in \text{ran } q.\text{submap})
\end{aligned}$$

The complete system may be defined as the synthesis of objects and collections together with the additional constraint that every resolvable value name must be held locally by exactly one component.

System

$$\begin{aligned}
&\text{SystemObjects} \\
&\text{SystemCollections} \\
&\hline
&\forall coll : \text{COLLECTION} \mid coll \in \text{collections} \bullet \\
&\quad \forall vn : \text{VALUE_NAME} \mid vn \in coll.\text{resolvable} \bullet \\
&\qquad \exists_1 c : \text{COMPONENT} \mid c.\text{cid} \in coll.\text{componentIDs} \bullet \\
&\qquad\quad vn \in c.\text{localNames}
\end{aligned}$$

Resolution

"Resolution" is the process by which value names are mapped onto components by traversal of the partition tree. The resolution operation accepts the value name being sought and a target component, and returns the name and the component which holds the name locally. The target and servicing component will always be in the same collection.

RESOLVE

$$
\begin{array}{l}
id? : VALUE\_NAME \\
target? : COMPONENT\_ID \\
id! : VALUE\_NAME \\
service! : COMPONENT\_ID \\
\Xi System \\
\hline
\exists\, coll : COLLECTION \mid \\
\quad coll \in collections \land target? \in coll.componentIDs \bullet \\
\qquad id? \in coll.resolvable \land service! \in coll.componentIDs \land \\
\qquad (\exists\, c : COMPONENT \mid c.cid = service! \bullet \\
\qquad\quad id? \in c.localNames) \\
id! = id?
\end{array}
$$

The resolution operation relies on the fact that, from any component in a collection, there is a path to a partition which can resolve the component holding the required data item. If we represent the parameters to a resolution operation as the receiving and servicing components, dereferenced from their identifiers and ignoring the value name for the present:

ResolutionParameters

$$
\begin{array}{l}
rec? : COMPONENT \\
ser! : COMPONENT
\end{array}
$$

then this property may be stated as follows:

$$
\begin{array}{l}
ResolutionParameters;\ System \\
\vdash \\
\quad \forall\, coll : COLLECTION \mid coll \in collections \bullet \\
\qquad rec? = ser! \lor rp\ resolves\ ser! \lor \\
\qquad (\exists\, p : PARTITION \mid p\ above\ rp \bullet p\ resolves\ ser!) \\
\quad where \\
\qquad\quad rp : PARTITION \\
\qquad\quad \hline \\
\qquad\quad rp \in partitionObjects \\
\qquad\quad rp.pid = rec?.part
\end{array}
$$

This property may be proved trivially as the partitions form a tree with references at every node to all components below: therefore there exist either no partitions or a single root partition able to partially route any request. In other words,

$$
\begin{array}{l}
\text{System} \\
\vdash \\
\quad \forall\, coll : COLLECTION \mid coll \in collections \bullet \\
\qquad coll.partitionIDs = \varnothing \lor \\
\qquad (\exists_1\, p : PARTITION \mid \\
\qquad\quad p \in partitionObjects \land p.pid \in coll.partitionIDs \bullet \\
\qquad\quad\ \forall\, c : COMPONENT \mid \\
\qquad\quad\quad c \in componentObjects \land \\
\qquad\quad\quad c.cid \in coll.componentIDs \bullet p\ resolves\ c)
\end{array}
$$

It is this property which allows a partitioned collection to behave as a single resource, with all components acting as pseudonyms for each other, whilst still maintaining an essentially distributed nature.

Creating and Manipulating Collections

If a component is judged to be full, it may be split. Splitting has an important property: it is value-preserving across the set of mapped names, even though the contents of components and the object population may change.

SPLIT

$$
\begin{array}{l}
id? : VALUE\_NAME \\
service? : COMPONENT\_ID \\
id! : VALUE\_NAME \\
service! : COMPONENT\_ID \\
\Delta System \\
\hline
\quad collections' = collections \\
\lor \\
\quad (\exists\, oldcoll, newcoll : COLLECTION \mid \\
\qquad oldcoll \in collections \land \\
\qquad collections' = collections \setminus \{oldcoll\} \cup \{newcoll\} \bullet \\
\qquad\quad oldcoll.resolvable = newcoll.resolvable \land \\
\qquad\quad oldcoll.componentIDs \subset newcoll.componentIDs \land \\
\qquad\quad oldcoll.partitionIDs \subseteq newcoll.partitionIDs)
\end{array}
$$

This operation will be used in defining user-level access operations.

The overall collection-creation function must simply generate some collection which can contain all the value names specified.

CREATECOLLECTION

$$
\begin{array}{l}
global? : \mathbb{P}\ VALUE\_NAME \\
root! : COMPONENT\_ID \\
\Delta System \\
\hline
\nexists c : COMPONENT \mid c \in componentObjects \bullet c.cid = root! \\
componentObjects \subset componentObjects' \\
partitionObjects \subseteq partitionObjects' \\
\exists coll : COLLECTION \bullet \\
\quad coll \notin collections \land coll \in collections' \land \\
\quad root! \in coll.componentIDs
\end{array}
$$

Operations

We shall define two operations on collections to illustrate the process: getting a value and assigning to a value. All operations share a common framework: the supplied ideitifier is resolved to the correct servicing component, at which the activity specified by the operation occurs.

Getting the value of a name from a collection simply involves accessing the local storage function of the servicing component. The result of the operation is the value associated with the supplied name, or the null value if no such association exists.

RETURNVALUE

$$
\begin{array}{l}
id? : VALUE\_NAME \\
service? : COMPONENT\_ID \\
value! : VALUE\_ID \\
\hline
id? \in c.localNames \land \\
(id? \in \mathrm{dom}\ c.local \iff value! = c.local\ id?) \land \\
(id? \notin \mathrm{dom}\ c.local \iff value! = NULLVALUE\_ID) \\
\text{where} \\
\quad \begin{array}{l}
c : COMPONENT \\
\hline
c.cid = service?
\end{array}
\end{array}
$$

The entire get operation may be represented as the composition of resolution and storage access.

$$GET \mathrel{\widehat=} RESOLVE \gg RETURNVALUE$$

Altering a value – assignment into a storage function – is slightly more complex. The alteration of the servicing component involves modifying the

component selected, over-riding its storage function to reflect the assignment.

ALTERVALUE

$$
\begin{array}{l}
id? : VALUE\_NAME \\
service? : COMPONENT\_ID \\
value? : VALUE\_ID \\
\Delta System \\
\hline
id? \in c.localNames \land \\
(\exists\, newc : COMPONENT \bullet \\
\quad newc.cid = c.cid \land newc.part = c.part \land \\
\quad newc.localNames = c.localNames \land \\
\quad newc.local = c.local \oplus \{id? \mapsto value?\} \land \\
\quad componentObjects' = \\
\quad componentObjects \setminus \{c\} \cup \{newc\}) \land \\
partitionObjects' = partitionObjects \land collections' = collections \\
\text{where} \\
\quad \begin{array}{l} c : COMPONENT \\ \hline c.cid = service? \end{array}
\end{array}
$$

Since assignment may cause a component to be split, the full operation involves resolution, (possibly) splitting, and storage over-riding.

$$PUT \triangleq RESOLVE \gg SPLIT \gg ALTERVALUE$$

One might similarly define a value-removal operation by subtracting a pair from the appropriate local storage function, and so on: the point is that *all* operations are composed as resolution-plus-action. Similarly, high-level user-defined operations may be defined as compositions of the basic operations.

Appendix B.

Wisdom

The Wisdom project investigated the design and implementation of a general-purpose scalable parallel computing engine (SPCE) based on an extensible mesh of processing nodes. The principal product of the project is an operating system nucleus which provides the minimum support necessary for running communicating tasks on such a processor mesh. At the time of writing, a prototype of the Wisdom nucleus has been implemented for a mesh of Inmos T800 Transputers.

B.1. The Wisdom Nucleus

Wisdom creates a virtual machine in which a (potentially vary large) number of communicating tasks are executed in parallel. Inter-task communication is performed through a capability-based message-passing system, not through a shared primary memory, so communicating tasks can be executing on different processing nodes. Although Wisdom distributes tasks amongst the available nodes, tasks are not aware (unless they want to be) of their location, or the location of tasks with which they are communicating. This *location transparency* means that the user can think of a Wisdom system as being the same as a traditional time-shared computer, except that tasks really execute in parallel rather than having their executions interleaved. It is hoped that this abstraction will encourage programmers to produce applications composed of a large number of communicating tasks. Such applications would execute on a Wisdom system with any number of processors, but should run more quickly on a Wisdom system with a large number of processing nodes since each task could be executed on a separate node.

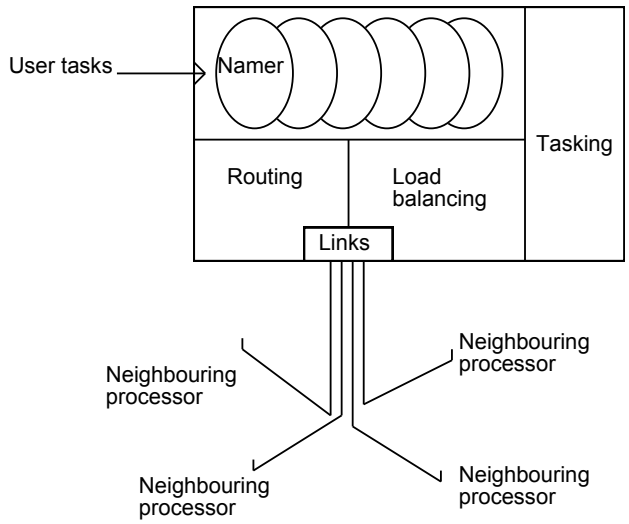


Figure 25: The Wisdom nucleus

The Wisdom nucleus runs on each processing node in the mesh, and is composed of four principal components (or modules) (figure 25). The modules are:

- *tasking module* – manages the allocation of memory and processor time between the tasks running on the node;
- *load balancing module* – manages the creation and distribution of new tasks amongst the nodes. When a new task is created the load balancing module examines the load of its own processor and of its four immediate neighbours. The new task is then created on the processing node with the lowest load. (There is a threshold function used to avoid conspired thrashing: see Murray[92] for details.). This style of load balancing results in tasks spreading-out in a "ink-blot" pattern (see figure 26);
- *routing module* – manages the transfer of messages between tasks on the same or different processing nodes. A *capability*[88] abstraction is used to allow tasks to identify the destination of messages without having to know the targeted task's location. A capability is a user-space object with a single reader (its creator) but may be written to by any task which has a copy of it. Capabilities may be freely exchanged in messages. At present the routing modules on all nodes co-operate to move message around the network using a *store and forward* routing mechanism: this is a consequence of the lack of routing hardware on the Transputer, and could be rectified by a different processor design or a routing co-processor;

- *naming module* – provides a way for tasks to associate a textual name with a capability. Other tasks may then obtain a capability to a task if they know its textual name. Although naming module is considered to be part of the nucleus, it runs as a user task.

In addition to the nucleus, other prototype software has been developed to provide a primitive user interface to the Wisdom system. This includes a simple file system that allows programs to access the department's NFS-based file server network and Unix services, a shell, and small number of utilities.

B.2. The Filing Systems

The Wisdom nucleus, by itself, does not constitute a working operating system: there is, at the very least, a need to be able to access files. Two file systems have been provided for Wisdom.

The first is a simple attachment of the Wisdom system to an external NFS file server. A library of file system calls interact with a process which converts file system requests into TCP/IP requests, which are then sent to another process running on the host system: this host then places these packets onto the EtherNet to which the NFS servers are attached. Returning data is handled in a similar way. This structure allows Wisdom to access the filing system of its host, using the most widely-available network file system standard. As a bonus, Wisdom applications can also use the TCP/IP libraries to communicate with any process running on the host *via* sockets[12], and the system can also provide a proper log-in suite using the host's password protection scheme.

The more interesting file system is an experimental project called Sage[11]. This project involved the creation of a filing system explicitly designed for use on a scalable parallel computing engine. It makes extensive use of caching: files are cached as near as possible to the site at which they are used[9], with processes sharing caches for common files. The system also identifies commonly-used, infrequently-updated files (such as system binaries, font descriptions and the like) for optimised handling.

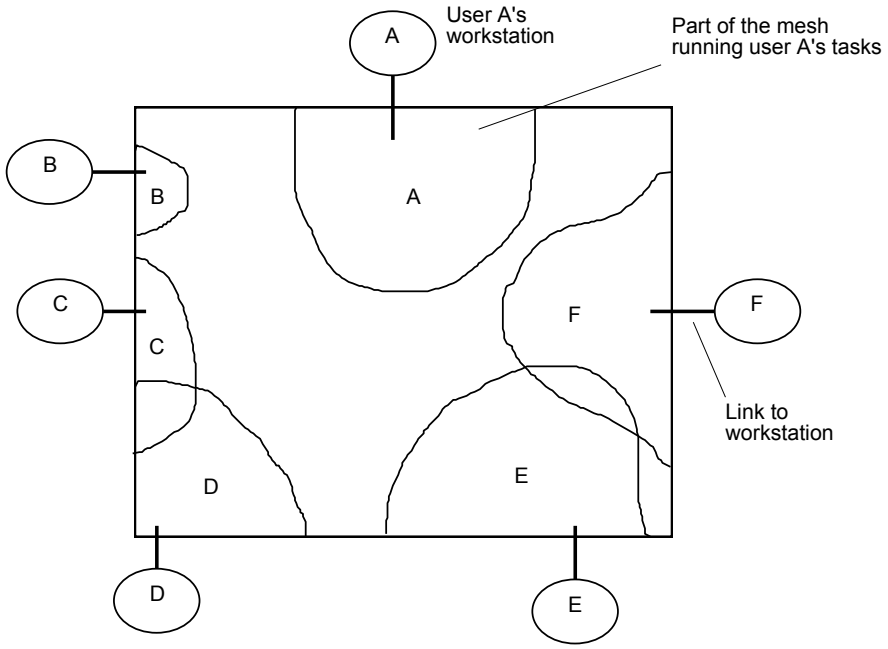


Figure 26: A Wisdom system in use (showing load balancing)

B.3. Wisdom in Use

The intention is that a Wisdom system would be used to serve a group of users in the same way as a traditional time-shared computer. One Wisdom system with an appropriate number of processing nodes, some secondary storage devices and other peripherals would have terminals or workstations connected to it by high-bandwidth links. The connection points of the terminals would be distributed evenly throughout the mesh so that each user would start programs off at a separate point in the mesh and his (or her) tasks would flood-out from this point (figure 26). (In a Wisdom system using Transputers the connection points must be at the edges of the mesh since only the Transputers at the periphery have a free communications link.) Users would be allowed to use any terminal connected to the mesh, regardless of the terminal's connection point.

The current Wisdom nucleus presents the user with an environment very similar to that of Unix. A full-function shell allows files to be examined and executed direct from the host filing system, and most of the standard Unix tools are available. This is a marked contrast to most parallel systems, which must access filed data using some other, less convenient mechanism.