OBJECT-ORIENTED DIVIDE-AND-CONQUER FOR PARALLEL PROCESSING

Andrew James Piper

Emmanuel College
Cambridge

July 1994

This dissertation is submitted for consideration towards the degree of Doctor of Philosophy at the University of Cambridge

Summary

The quest for cost-effective supercomputing performance has been hindered largely by the complexity and lack of usability of the software involved.

Techniques have been devised to address these issues. On the one hand, compilers have been constructed that attempt to parallelize serial programs. On the other hand, languages have been devised that enable parallelism to be extracted easily. However, smart compilers have proved relatively bad at generating efficient parallel programs and parallel languages have proved difficult to use.

More recently, various software techniques have been proposed for reducing the implementation complexity of parallel software. Among these techniques are those of object-oriented programming and the divide-and-conquer algorithm. However, both these techniques fall short of the ideal. Parallel object-oriented languages are easy to use but do not necessarily yield high performance. The divide-and-conquer algorithm yields ready parallelism but is syntactically poor.

The objective of this thesis is to use object-oriented techniques to express parallelism in a way that enables divide-and-conquer evaluation to be performed easily, but which is also easy to program. The result is accessible, efficient parallelism.

To achieve this, a system is designed for implementing problems on parallel computer hardware using these techniques. This system is composed of two parts. The first part is a run-time system that evaluates divide-and-conquer problems which have been expressed using an object-oriented framework. The second part is a library of simple divide-and-conquer problems.

Object-oriented technology is a way of writing programs that reflect the structure of the real world. This thesis demonstrates that the use of this technology enables divide-and-conquer problems to be implemented easily, and enables complex problems to be implemented through the combination of simpler divide-and-conquer expressions.

The experimental work of this thesis focuses on the implementation of real-world problems. This enables a more objective assessment of the success of the system to be made, in terms of both parallel performance and ease of programming. The implementation of many of these problems is both original from a parallel perspective as well as a divide-and-conquer perspective. The problems addressed include: a neural network algorithm where the limiting factor is the size of the model; a neural network algorithm where the limiting factor is the amount of training data; and a speech recognition algorithm for making word predictions based on their context.

As well as contributing a general parallel-programming system, this thesis contributes novel object-oriented techniques and implementation enhancements for the divide-and-conquer algorithm. One technique enables maximum computation to be achieved in a divide-and-conquer evaluation. Another enables divide-and-conquer implementations to be scaled effectively.

This thesis concludes that the techniques presented do enable problems to be implemented efficiently, in parallel, with a minimum of programmer effort. The techniques employed mean that each implementation benefits its successor, and this is reflected in the implementations discussed in this thesis.

Keywords: Parallel processing, object-oriented programming, divide-and-conquer.

Acknowledgements

I would like to thank Richard Prager, my supervisor, for his direction and comments over the years; the Trollius team: Greg Burns, Raja Doud and James Vaigel, for a superb operating system and helpful support; countless others who have provided me with ideas, help, financial support and encouragement; and Libby, my wife, for putting up with a penniless student.

This dissertation was funded by the Science and Engineering Research Council.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration with any other party.

Contents

Chapter 1

Introduction

There is only one basic way of dealing with complexity: Divide and conquer.
B. Stroustrup

Parallel processing has the potential to deliver cost-effective supercomputing power. However, if the benefits of parallel processing are to be experienced by applications programmers of all abilities, then the difficulties in implementing parallel programs need to be tackled. Two common approaches to the problems of achieving this *accessible* parallelism have emerged. The first approach involves producing a custom language for which language constructs can be easily implemented in parallel – for instance functional languages. The second approach involves taking a serial program and extracting parallelism from it using a compiler. These two approaches both fail to some degree because of their extremity. Custom languages that are easily parallelized are not generally suitable for tackling complexity. The compiler benefits, the programmer does not. Parallel feature extraction is difficult and inefficient at best, and limited by the number of parallel features present in a program. At worst it is impossible. The programmer benefits at the expense of parallelism.

Divide-and-conquer (D&C) [3, 38] in conjunction with object-oriented programming [102] may offer a middle ground which achieves the best of both worlds. D&C is a well-studied computational paradigm that can be easily and efficiently parallelized. Object-oriented programming allows the complexity of the parallelism provided by D&C to be hidden from the programmer. At the same time the programmer can benefit from the features of object-oriented programming in general. This thesis presents an object-oriented framework for D&C, describes the benefits of the object-oriented approach and develops a number of applications within this framework.

1.0.1 Thesis organization

In chapter 1 we give an overview of relevant parallel processing research, object-oriented techniques and the D&C algorithm. In chapter 2 we describe the general ethos behind D&C in an object-oriented environment, and demonstrate that the idea is practically viable. In chapter 3 we describe a more sophisticated design for an object-oriented, D&C system. We go on to introduce classes based on this system, and outline its practical implementation. In chapter 4 we describe some further D&C classes, and analyse theoretically the D&C algorithm. We also describe the implementation of a practical problem. In chapter 5 we examine a D&C implementation of the Kanerva model and, using this model as a basis, develop some additional D&C classes. Finally in chapter 6 we summarize the thesis and its contribution.

1.0.2 Background

Research into parallel processing has been motivated in the main by its potential to deliver cost-effective computational power. It allows, or should allow, complex problems to be solved faster and more cheaply than ever before and creates the opportunity for solving problems of previously unmanageable size. However, parallel processing and parallel programming have turned out to be hard problems; problems that have consumed much in terms of man hours and computational resources. Solutions to this 'hardness' have emerged which generally compromise usability whilst maintaining efficiency, or, compromise efficiency whilst maintaining usability. This thesis argues that both these approaches are not ideal in producing accessible parallelism and discusses some previous solutions which represent 'half-way houses'. It then argues that some of these solutions are themselves not ideal in solving real problems and presents an alternative solution. This solution combines the D&C paradigm [3] to utilize parallelism, and object-oriented programming [102] to express this parallelism and abstract it from the D&C internals. The characteristics of D&C and object-oriented programming are discussed and the combination of the two in this context outlined. Results are then presented to demonstrate the efficiency of the system in terms of programmer productivity and processing speed.

1.0.3 Approaches to parallelism

Two general approaches to the general problems of parallelism have emerged. The first approach involves taking an individual problem and trying to find the most efficient parallel solution for it; systolic [75] algorithms for instance. The second approach involves taking any solution specification and transforming it in such a way as to achieve maximum possible parallelism from it - the holy grail of parallel processing; vectorizing FORTRAN compilers for example [69].

The first approach yields solutions that are fast and efficient but unapplicable, in any but the most general sense, to most other problems. This approach generally requires a high-level of expertise both in the theory of the problem concerned and in the problems of parallel processing. Additionally, it exacts a high development cost for each problem solved.

The second approach is, in general, not possible as it requires parallelizable features to be present in a given solution - *do* loops, independent function calls, parallel data etc. If these features are not present then the solution is not parallelizable. Solutions that are in some way, parallelizable are generally realized with a high degree of inefficiency. However, the solution gaining process can be applied to many different types of problem. This approach initially requires an extremely high-level of expertise in parallel processing and problem solving in general and exacts an extremely high development cost. However, each problem subsequently solved with this system, requires little knowledge of parallel processing and a relatively low development cost.

Neither solution can really be justified as cost-effective in a global sense. If cost-effective means acceptable development time *and* acceptable performance then it can only be achieved through accessible parallelism. The need for accessible parallelism has motivated many efforts which fall into an area between the solutions referred to above. These efforts involve placing constraints on the solution method which allow inherent parallelism to be extracted easily. This approach commonly involves constraints on the input specification, i.e. language, and thus requires some degree of expertise in using such an input specification, but requires little intimacy with parallel processing / programming concepts. Functional languages are a common example of this approach - problems solved using a pure functional language may be easily parallelized using parallel graph-reduction.

1.0.4 Hybrid input specification

The problem with the constraints mentioned in section 1.0.3 is that they are usually applied across the board, and this leads to unnecessary inconvenience and effort in developing aspects of a solution that do not require parallelism. It leads subsequently to difficulties in designing non-trivial systems - witness the fact that there are few functional language based, application oriented systems. There already exist many sophisticated tools and languages which allow most computational problems to be solved effectively. These tools and languages have a large installed user base, and it would therefore seem wasteful to replace this sophistication with something that only favours parallelism. What is required is a hybrid of the two representations - the freedom to use current serial programming tools and techniques *and*, at the same time, easily incorporate and express inherent parallelism.

1.1 Literature survey

Parallel processing is not a new concept; while computing was still in its infancy, tackling a computational task in a concurrent fashion had been postulated. The rapid evolution of parallel processing, though, has happened relatively recently, as the realization of cheap parallel machines has only taken place in the last 10 or so years. This evolution has been largely dictated by advances in microprocessor technology, and the shortcomings of various approaches to parallel processing, that these advances have highlighted.

The scope of parallel processing is enormous. Hwang & Briggs have defined parallel processing as,

" ... exploitation of concurrent events in the computing process." [60, p6]

which leaves great scope for different approaches.

1.1.1 Classification

As with serial computing, parallel processing has many aspects, each at a different level of abstraction. These areas can be broadly classified as *hardware architecture, operating system* and *language*.

Hardware Architecture. Within this area the main types of architecture are SIMD and MIMD machines, according to Flynn's classification [37].

SIMD machines have a single instruction stream, but operate on multiple data streams. In essence they are array processors, and as such present a rather inflexible, but efficient, approach to parallel processing. Examples include Crays and the DAP. Many new microprocessors, especially DSP chips, are SIMD machines themselves.

MIMD machines have concurrent threads of execution operating on distinct data streams.

Operating System. This performs a similar purpose to normal computer operating systems, but with features that express concurrency and process communication well, for instance Helios and Trollius^{TM} [79]. Many parallel operating systems have close inter-dependency with a particular language, e.g. Emerald [27] and Concert [26].

Language. Ben-Ari's book gives a good overview of parallel languages [10], see also Gehani [39].

Languages take many forms:

- New languages designed to exploit parallelism. However, end-users tend to be loath to learn new languages and, while a language may be good at expressing parallelism, it may not be good at expressing functionality. e.g. Occam [57] based on Hoare's CSP [56].
- Old languages with concurrent extensions. These have the advantages of large user-bases and small learning difficulties. However, they may be poor at expressing parallelism. e.g. Concurrent C.
- Old programs transformed into parallel programs. While this approach presents no learning difficulties, it is extremely poor at utilizing parallelism. e.g. the ASPAR system [61].

Somewhere in the middle of these areas, lies problem solving in a parallel fashion. This is where the programmer has to work within some regime less demanding than fully parallel program design. In return the system performs at least part of the exploitation of the available parallelism automatically. It is this area that this Thesis addresses.

1.1.2 Message passing vs shared memory

MIMD machines perform two functions; those of *computation* and *communication*. Frequently, it is communication that is the limiting factor on speed and scalability, and thus, communication is an important topic for any treatment of MIMD parallel processing.

Broadly speaking, current research lies in two areas of communication; those of *shared memory* and those of *message passing*. Hwang & Briggs' treatment of parallel architectures is rather biased towards shared memory and SIMD machines as message passing machines have only become popular relatively recently. This popularity has been largely precipitated by Hoare's pioneering work on CSP [56], by the advent of the transputer by Inmos [57], and by the realization that shared memory architectures which utilize the shared memory for inter-process communication (IPC) with large numbers of processors, have fallen prey to the bus contention problem[1]. More recently still, virtual shared memory machines that rely on message passing for communication have become popular [50], although it is not clear that this yields maximum efficiency [68].

Message passing architectures are those where IPC takes place through some sort of serial pipe. This method greatly reduces the IPC allowed as a pipe is usually dedicated to two processes (exceptions do exist [40]), but eliminates bus-contention. Hoare [56] used message passing as a general computational paradigm and his work was both innovative and brilliant. Languages, like OCCAM [57], based on this message passing paradigm do exist. However, the implementation of OCCAM at least is insufficient for designing real

[1] This is where two or more processors must a share a hardware bus in order to share address spaces. Accesses to the bus must be mutually exclusive, limiting overall performance.

systems. Message passing is thus, more commonly used as an extension to normal, serial, languages. Message passing forms the basis of many current MIMD machines. A very readable survey of MIMD / message passing parallel processing and programming may be found in Fox [38].

A lot of interest still lies in shared memory approaches as these have many advantages, communication time is low for instance, and a lot of concurrent computational theory is based on shared memory approaches e.g. the EREW PRAM model [32]. However, this thesis is directed at message passing approaches as it is felt that only these can yield machines that are *scalable* and thus potentially *massively parallel.*

1.1.3 Scalability and efficiency

It is difficult to define what represents an ideal parallel system, but two desirable attributes are *scalability* and *efficiency.* The efficiency of an architecture is defined in terms of the speedup gained with respect to the number of processors (or *nodes*) used. For example, if a task takes n seconds to complete on a single node and $\frac{\alpha n}{k}$ seconds to complete on a parallel machine with k nodes, then the speedup would be $n/(\frac{\alpha n}{k}) = k\alpha^{-1}$ and the efficiency would be $k\alpha^{-1}/k = \alpha^{-1}$. For ideal systems $\alpha = 1$ and the speedup is said to be *linear*, for good systems $\alpha > 1$ and is constant[2], for poor systems $\alpha = f(k)$ and the speedup tails off rapidly[3]. In some circumstances, $\alpha < 1$ and the speedup is said to be *superlinear*[4].

We should note that speedup is measured relative to the same algorithm on one processor, and that this gives us a realistic idea of how good the processor utilization of an approach is. It is also necessary, in terms of real performance, to assess speedup on a particular problem relative to the corresponding serial algorithm.

Scalable architectures are those for which their computational principles hold, whatever the dimensionality of the machine and for which their efficiency is close to ideal. For instance a machine in which the nodes are hypercube connected is *not* scalable, as the number of inter-node connections is infeasible for moderate to large numbers of nodes. Scalability is desirable because it provides a gateway to powerful machines with very large numbers of nodes which would otherwise be impossible to design. Much research is directed at machine models that will lead to these massively parallel machines [8].

It is interesting to note that Fox et al. [38] are optimistic about scalability of MIMD systems, whereas Hwang & Briggs [60, pp27–29] are much more pessimistic giving *Amdahl's law* as a bound on speedup. However, the results of Fox et al [38][5] justify their optimism and show that different approaches have vastly different characteristics.

More recently Gustaffason [52] showed that speedup was dependent on the size of problem being executed. If a problem is scaled in size then the impact of serial overhead can be reduced, allowing programs that effectively break Amdahl's law.

---

[2]This usually corresponds to some fixed overhead per node.

[3]See [38, 60] for a more detailed treatment.

[4]This can happen when the basic computational algorithm used for exploiting parallelism is actually more efficient than the common serial solution.

[5]They obtained linear speedup for many problems.

1.1.4 Drawbacks

Parallel processing has a few major drawbacks. Firstly, it is conceptually complex. Some would disagree with this [62] and have argued that this complexity is because we have been conditioned to think serially by existing, serial based, computer languages. Others argue that because the brain is parallel in nature, parallel concepts should come easily to us. Whatever the reason, parallel processing and programming *are* complex [49] and are thus not readily accessible. Much research is currently directed at improving this accessibility through advanced languages [44], abstract machine concepts and production tools [61].

Secondly, it is physically complex. This complexity arises from the enormous variety of possible relationships between processors, memory and I / O, and the variety of possible processor topologies. Much research is directed at parallel concepts which unify, to a degree, the different possibilities [80].

Thirdly, current serial techniques do not lend themselves well to parallel processing and there is no recognized, standard, parallel alternative. Parallel techniques are still emerging; each with its advantages, disadvantages and proponents. Most of them have the drawback that, unlike serial methods, they are only applicable to, or at least efficient for, a small set of problem classes - no one technique can solve all problems. Because of this, much research is directed at solving different classes of problems using one particular technique and to improving the performance of a given technique for a more generic problem.

Lastly, parallel programs are often *non-deterministic*. In a sequential program, a task cannot start until its predecessor has completed. In a parallel program, the concurrency implies that a child could finish *before* its ancestor. It is situations like this that mean poorly coded parallel programs can yield differing solutions, depending on the time particular tasks take. This makes parallel programs difficult to debug; simply the introduction of debugging messages can cure or create a problem!

1.1.5 Solution classes

In order to discuss the many different classes of solution, we have to introduce a number of parallel processing concepts.

In general, problem data has a source and a destination. Usually, with MIMD architectures, this data is sourced from a location outside of the parallel machine. The I / O between the two locations, usually takes place through a single hardware route. Thus, data needs to be *broadcast*, from this point source, to processors within the parallel machine. At the end of processing, the data has to be returned, or *multi-reeled*. It is obvious that this point source represents a bottleneck to dataflow. Multi-I / O solutions have been tried [38], but these are expensive and difficult to manage.

Problem data is usually not uniform, for instance sparse matrices. This means that an even division of data between processors does *not* imply an even distribution of the load. Thus *load balancing* is an important part of parallel processing solutions if these solutions are to be efficient. It can be static [89], but is more generally dynamic [66, 51] as the run-time progress of a task is difficult to predict before its execution.

A number of general solution classes have emerged:[6]

[6] These classes have gained popularity because of their applicability to MIMD architectures. It should be noted that many other solution classes exist, but have lost prominence because their applicability is tied to increasingly outdated parallel architectures.

Data-parallel programming. Here, the problem classes do not have data dependencies and, given a local piece of the data, a local piece of the result can be found. Thus data can be distributed over a processor array, the local solutions found, and the results returned. The advantage is that the solution does not depend on processor communication apart from in the broadcast and reeling phases. This gives a relatively straightforward processor topology and a linear speedup excluding the data transmission. Unfortunately, many problems are not data-parallel in nature. Also, data tends to be very localized and this often leads to poor processor utilization - load balancing often has to be an integral part of this technique for it to be efficient.

Nevertheless data-parallel machines dominate the commercial parallel world because of their cheapness and suitability to some important problems, e.g. seismography and image processing. Data-parallelism is embodied by the connection machines [55]. These machines are data-parallel in operation and are programmed using the data-parallel language C* [105].

Graph reduction. This is an example of *demand-driven data-flow*. In this scheme reducible expressions are replaced by their computed results. Computation is only limited by the availability of data. Reductions in the same expression are independent, and can therefore be performed concurrently. Graph reduction is best known for its application to functional programming.

Functional languages in their purest form are languages where all primitives, data structures etc. are represented as functions. This implies that there are no globally visible data structures and thus, any two functions are independent as long as one is not an ancestor / descendent of the other. This makes programs written in functional languages ideal for parallelism and much research is directed towards functional approaches to parallelism, for example [59, 5, 88], and graph reduction systems. Functional languages also have the nice property that they map well into mathematics. Much research has been done into the mathematics of functional languages and new mathematics has been designed to accommodate functional approaches e.g. the lambda calculus [9] and serial combinators [59].

Graph reduction has many advantages, not least the fact that parallelism is implicit to the system, and easily extracted. There are no programmer-visible parallel constructs.

Functional languages have the disadvantage that they are difficult, and constricting, to program in sensibly. Indeed, there are very few examples of real applications using functional languages.

Systolic arrays. Definitions vary but the general idea is that of dataflow with results being obtained at a fixed frequency, hence the name systolic. Data passes through an array of processors which can be trivially linked (e.g. pipelining) or linked in a more complex fashion. The advantage of this approach is that once the array is full, results can be obtained with high frequency. Also, the approach is readily scalable as processor interconnections tend to be uniform across the array. The disadvantage is that usually the processor interconnection is only applicable to a small set of tasks and thus the problem is very "built-in" to the solution. See [75] for example.

Explicit parallelism. Here, each parallel program is hand-coded with program interactions explicit, and the overall solution tied to the particular problem being solved.

> Applications programmed in this way tend to be very fast and efficient, but difficult and costly to create and maintain.

Of these particular classes we concentrate on the area of graph-reduction, since this appears to offer some degree of generality, whilst maintaining efficiency. In particular we look at D&C, which is a higher-level graph-reduction paradigm that has been used successfully by a number of people. The next section describes various aspects of this paradigm.

1.2 Divide-and-conquer

In this section we describe the D&C algorithm and its parallel implementation.

D&C is well known for its simple parallel evaluation properties and has been widely used in a functional language evaluation context [88] as well as a programming paradigm in its own right [5, 6, 76, 20].

1.2.1 D&C explored

Divide-and-conquer (D&C) is a computational paradigm [3] which is very similar to graph reduction in a parallel context [58]. The dividing line between D&C and graph reduction is very blurred, but essentially it involves solving the *physical problem*, rather than in solving the *parallel problem*. Obviously this constrains the problem class, but at the same time relaxes the constraints on implementation, from which functional programs suffer[7].

A general D&C algorithm can be expressed as follows: First, test the data to see if it is so small that a result can be obtained from it immediately, without needing to sub-divide the data. This is done by a function we shall call *simple()*. If a result can be obtained then the function which does this we shall call *evaluate()*. If the data is not simple, it must be divided into parts using a function we shall call *divide()*. Each part may then be further divided, and so on. This subdivision of the data ends when all the parts are so simple that no further subdivision is needed. The results obtained from each simple part are then combined together, using a function which we shall call *combine()* to give the final result [7]. It should be noted that the bulk of computation can lie in any of *divide()*, *function()* and *combine()* [4].

The D&C algorithm can be represented in pseudocode as follows:

```
FUNCTION divacon (data)
BEGIN
        IF simple (data)
        THEN RETURN evaluate (data)
        ELSE combine (map(divacon,divide (data)))
END
```

It should be noted that the most common form of D&C is binary D&C where division yields two sub-tasks.

[7]For example, we could *solve* matrix multiplication by a D&C method and the method, rather than the program, would be parallelizable. Alternatively, we could *write* our program in *Haskell*, a functional language, and the whole program would be parallelizable, rather than our particular matrix multiplication paradigm.

D&C forms the basis of many serial algorithms, algorithms which are often more efficient than traditional approaches. For example, matrix multiplication can be performed in $O(n^{2.81})$ operations by a D&C method, rather than the normal $O(n^3)$ operations [100]. Indeed, normal integer multiplication by bit shifting and adding can be performed more efficiently by a D&C method [3]. Additionally, techniques have been devised for specifying problems in D&C terms [97].

1.2.2 D&C as a means to parallelism

Interest in D&C as a programming paradigm lies in the potential for parallelism in computing partial results for divided data. Axford [5] describes it as follows:

Suppose there are P processors available for parallel computation, then $y = divacon\ (d)$ can be computed by:

1. Compute a set of data values $d_1, d_2, d_3, ..., d_P$ by repeated application of *divide()* to d until the data is subdivided into P parts.

2. For each of the data values d_i, compute a partial result by sequential computation on the i-th processor.

3. By repeated application of *combine()* to pairs of partial results, compute the final result y from the set of partial results $y_1, y_2, y_3, ..., y_P$.

These three stages can each be implemented in parallel, although the greatest parallelism is possible for stage 2.

This property of D&C makes the solution of general computational problems by a D&C method desirable. Axford [5] gives some examples of these. In addition to simple computational problems, more computationally demanding ones have now been solved using D&C methods because of its potential for parallelism and thus decreased computational time [42].

The ZAPP project [19, 72, 73] has shown that excellent parallel performance can be achieved using the D&C paradigm. Clare extended this work to be more general [29]. Many others have investigated D&C architectures with varying degrees of success.

1.3 Parallel D&C

In order to address the implementation of a parallel D&C system, there are a number of issues that need to be considered.

As discussed previously, partial results can be obtained, in parallel, from sub-divided data. Thus, each data item potentially represents a parallel process and always represents a task to be completed. The ensuing discussion will refer to each data item as a task, but the two are essentially interchangeable.

A general strategy for implementation, used by others, has been that for each processor:

- Perform a *depth-first*, D&C, expansion of the task tree. This implies always completely expanding the left task first before expanding the right task (or vice versa). Alternatively, get a task from a neighbour if none are held locally. This is referred to as the *single-steal* rule.

- Put each unexpanded right-hand task into a FIFO queue and mark as *PENDING*. This implies that the largest tasks, representing the largest portions of data, are at the head of the queue, the size usually decreasing towards the back.
- If requested to by adjacent, unloaded processors, pop tasks from the head of the queue and pass them on to these processors. Mark the popped tasks as *FIXED*.
- When queued right-hand tasks need to be expanded, dequeue them. Note that because of the order in which tasks are queued, and the depth-first nature of the search, the task required for computation will *always* be at the back of the queue.
- If the task required is *FIXED*, then wait until it becomes *COMPLETE*.
- If the task tree has completed, then pass the result back to the parent processor from which the task was taken OR return the final result if there is no parent.

Note that the task tree is *not* expanded concurrently, only one task is ever being processed on the same processor.

It can be seen that many of these operations can be performed concurrently, in particular the *scheduling* of work between processors with the actual computation.

1.3.1 Scheduling strategies and load balance

In the general approach given above, it can be seen that the way work is scheduled yields an *automatic* load balance of the system. This is because if a processor is lightly loaded, it will finish quickly and thus be ready for more work. If a processor is heavily loaded, the workload should *diffuse* outwards from this processor, balancing the load. This is an important property of D&C systems.

In discussing scheduling strategies, it is necessary to introduce the concepts of *bidding* and *drafting*.

Bidding. In a bidding strategy, processors requiring work ask their neighbours for work. Bidding has the advantage that once the workload has been distributed, the scheduling network traffic is zero. It has the disadvantages that free processors do not acquire work as soon as it is available, and the scheduling network traffic is heavy while the workload is being distributed.

Drafting. In a drafting strategy, loaded processors tell surrounding processors of their desire to offload. Drafting has the advantage that work is scheduled very quickly, but has the disadvantage that it ties loaded processors up with even more processing.

Variations on the two do exist. Many approaches to load balancing in general attempt to utilize the diffusive nature of a computational load. Approaches that consider this are given in [51, 92].

In general, to be efficient, scheduling strategies need to be performed concurrently with computation. The single-steal scheduling of section 1.3 is bidding based.

1.3.2 Partition control

Within D&C, *partition* control is a means of changing the run-time characteristics of a D&C system. Partition control is based on the *simple()* functionality of D&C; what determines a simple task? In the case of matrix multiplication, the sub-problems involved are thinner and thinner columns, multiplied by shorter and shorter rows to yield a smaller and smaller result. At some stage it is more cost-effective to actually perform this multiplication, than to subdivide any further; ultimately, to a single row multiplied by a single column. The point at which this cost-effectiveness is maximized is the optimal partition.

Rabhi [87, 88] showed that, for balanced computation, the optimal partition is that which produces just enough sub-tasks to utilize available processors.

It should be noted that the optimal partition is only optimal with respect to potential parallelism. For most normal, serial problems, it is always more cost effective to perform the *evaluate()* immediately, with no sub-division at all.

1.3.2.1 Multi-processing

Another run-time change is multi-processing. In a system involving several dissimilar sub-problems, each potentially solvable by a D&C method, there may well be some advantage in performing some or all of these concurrently, each on a subset of the available processors [18].

1.3.2.2 Breadth-first vs depth-first search

A common variation on the implementation of D&C is breadth-first search. Burton [18] has shown that, space-wise, this is a more optimal method of expansion. He suggests that D&C systems should expand their task trees depth-first until space (memory) is short and then continue the expansion breadth-first to avoid deadlock.

1.3.3 Drawbacks and benefits

The major problems with D&C systems are those of semantics and use. Like functional languages, the syntax of D&C is fairly restrictive, and solving a real problem with this paradigm would appear to be a matter of some complexity. Mou [77] has developed a mathematical model for D&C and implemented his own language based on it [76]. However, this language is in essence a functional language and is also only applicable to fine-grain parallelism. We surmise that although it provides some syntactic convenience, it does not go far enough in yielding a system suitable for solving real-world problems.

The thrust of this thesis is to use proven software tools as the basis for implementing a D&C system. We aim to implement real-world, large-grained problems using D&C, and use *object-oriented* software technology to aid implementation. We describe object-oriented technology in section 1.4.

1.3.3.1 Input / output

D&C has the property that at the beginning of the computational process, all data resides on one node. This has the advantage that single-channel I/O, which is a "feature" of many parallel machines, is built into the problem solution. Thus, the single-channel I/O does not represent a bottle-neck to processing, as it does in other parallel techniques. The drawback is that this limits the problem size to that which will fit on one processor. As large problem size is desirable, if a parallel machine is to be utilized effectively using a D&C system, this drawback is one which must be worked around.

1.4 A tour of object-oriented programming

Object-oriented languages and techniques have a long and varied history, and have followed many different threads. Object-oriented features can be defined in general terms, however any meaningful survey must take into account the variety of approaches that exist. All these approaches are valid, although the religiousness that abounds in the subject tends to cause a bias concerning what object-oriented programming really is!

The term “object-oriented programming” tends to encompass a plethora of programming techniques, most of which, in one way or another, will be provided in an object-oriented language. However, to isolate which of these techniques lies at the heart of object-orientation is to fall between the cracks in the pavement. Thus one can describe object-orientation as an abstract concept, but to define it requires a close look at the techniques that make it possible.

In essence, the object paradigm takes a view of the problem domain in which objects are the basic abstraction. Object-oriented programming is an implementation technology that allows the structure of these abstractions to be encapsulated in an implementation. The beauty of this approach is that it shifts the design focus from the solution domain to the application domain. Thus the structure of an implementation will mirror that of the real world, the structure of which is, or should be, immutable. The manipulation of these abstractions in a familiar framework yields the ability to manage complexity; their immutability provides a basis for graceful system evolution.

The following sections describe the implementation techniques that are the enabling technology for the object paradigm.

1.4.1 Objects, classes, methods and members

“An object models some entity of concern in an application by encapsulating its structure and behaviour” [35]. A class is a description of what structure and behaviour its instances (objects) exhibit. Some object-oriented languages use cloning to create new objects rather than instantiation [106] from classes.

The structure of an object is composed of data *members*. The behaviour of an object is composed of *methods* or *member functions* i.e. procedures to manipulate the members of an object.

1.4.2 Message passing

A key element of object-oriented programming is the separation of procedure call from procedure invocation. This gives rise to the notion of message passing. In order to invoke an object method, a message is sent to the required object, whereupon that object has to lookup the appropriate method for that message. The relationship between message passing and method invocation can be determined at compile-time - *static binding* - or at run-time - *dynamic binding*. Most object-oriented languages require some sort of dynamic binding in order to implement polymorphism (see below), however, the relative use of the two binding types generally depends on whether the language is *strongly* or *weakly typed*. In a strongly typed language (e.g. C++ [102]), an instantiated object *cannot change its* *type*. The type is fixed at compile-time and thus a large amount of binding and type checking can be performed at compile-time. In a weakly typed language (e.g. Smalltalk [43]) an object can be defined as untyped, thus dynamic-binding must be used, as the type of the object - and so the relationship between message and method - can only be determined at run-time.

Static binding generally results in increased efficiency. Dynamic binding results in increased flexibility. However, it is possible to use sophisticated compiler techniques to improve efficiency in a weakly typed context [107]. It is also possible, as shall be described later, to imitate weak typing in a strongly typed language to gain flexibility.

1.4.3 Class hierarchy

In class-based object-oriented languages, classes categorize entities. However, different categories are not necessarily distinct. Different classes can be related just as different objects can be related by their class. The class relationship of primary importance is the "IS-A" relationship - a cow IS-A mammal - and is implemented through *inheritance*. Inheritance is a corner-stone of object-oriented programming. Through inheritance a *subclass* inherits the features of its *superclass* and then defines additional ones (*extension*) or refines existing ones (*specialization*). Inheritance mechanisms and sophistication vary, but the general concept is the same, a subclass has an IS-A relationship with its superclass.

Inheritance need not be a single-path relationship, after all a cow IS-A herbivore and a cow IS-A mammal but there is no natural inheritance relationship between herbivores and mammals, instead a cow inherits from both categories. This is generally called *multiple inheritance*.

Other class relationships exist; a class can be described as 'having' another class using the HAS-A relationship. For instance a car HAS-A wheel. Now although this relationship can be expressed through inheritance, it is not actually a *subtype* relationship. A subtype relationship is defined as:

> "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T" [71].

and inheritances mechanisms *should* be used to express this, but a HAS-A relationship should be defined using a class member. If class A HAS-A class B then A can be defined with a *member* of type B.

1.4.4 Polymorphism

The notion of subtyping leads on to the concept of *polymorphism*. A subtype relationship between classes indicates a shared *interface*: all employees, say, could have a method for updating their salaries. However, the shared interface does not imply a shared *implementation*: the C.E.O. would have a different salary update mechanism to the cleaner. Polymorphism allows us to capture this relationship. It implies subtyping; the C.E.O. IS-A employee; and also dynamic binding; in presenting a salary update message to an employee interface, the actual type of the object must be determined at run-time and then the appropriate method invoked.

The interface can be defined as *abstract*. We cannot instantiate a general employee because the implementation of salary update is not defined for employees. Subclasses would then implement the salary update mechanism, or, if a superclass had already defined it, it could be *overridden* through redefinition.

Polymorphism is crucial to object-oriented programming. Where inheritance allows us to capture class relationships, polymorphism allows us to manipulate abstractions in a general way.

1.4.5 Data hiding

Interface manipulation leads on to interface specification. An object may have features that are to be used internally and need not be a part of the object's interface. Object-oriented languages formalize this idea with the concept of *data hiding*. Generally an object will have: features that are part of its global or *public* interface; features that can be used by it and any subclasses derived from it, its *protected* interface; and features that only need be accessed internally, its *private* interface. Data hiding gives the programmer the scope to force the use of an object in a particular way. A benefit of this is that the internal implementation of an object can be changed without the changes affecting programs that use the object in any way.

1.4.6 Delegation

Some object-oriented systems treat everything as an object. This means that messages are as much objects as state entities [23, 13, pp63–66]. Some of these systems provide delegation as a way of invoking methods. This means that when an object receives a message it delegates it to *all* of its members (both methods and variables). The delegation message holds the name of its originator as an argument. The value of the first object to accept the message is then returned.

The power of this system is apparent when we consider *extension objects*. If an object has as a member another first-class object, instead of a method or simple state variable, then delegated messages can be passed on to the *methods of this object*. One of its methods then accepts the message and returns a value. To the user of the original interface, it appears as if a normal method invocation has occurred. By adding extension objects in this way, it is possible to extend the functionality of an object dynamically, *without* using inheritance.

1.4.7 Overloading

Another feature of some object-oriented languages is that of function overloading. Overloading allows functions taking different arguments to have the same name. The compiler arranges for the right function to be called according to its calling context. At first glance this appears to have nothing to do with object-oriented programming. However, overloading is essential in strongly-typed object-oriented languages, in order to allow new classes of objects to fit homogeneously into the existing type system. This is especially true for infix, prefix and postfix operators: an object which represents a complex number clearly needs to understand the arithmetic operations which might naturally be used with it, even though these operations are already defined for other types.

Dynamically-typed object-oriented languages have less need for function overloading since functions can be defined for typeless objects and the appropriate invocation found at run-time.

1.4.8 Guarded methods

A feature of some object-oriented languages, Eiffel [74] for example, is that of guarded methods. This allows a programmer to specify entry and exit conditions for the invocation of a particular object method. This provides a sort of run-time equivalent to type-safety. Method invocations not meeting the guard conditions can be flagged as errors and dealt with by an error handler.

1.4.9 Parameterization

Often an IS-A relationship expressed through inheritance is not quite what is required. We might want to define a type which has variants that depend only on what the types of its features are. For instance an array might be of integers or floating point numbers. In both cases the object is the same apart from its *parameterization*. Most object-oriented languages have support for this feature and the classes created are termed *generic* or *parameterized* types. C++ in particular calls these types *templates* [101]. C++ templates accept parameters much the same as functions do. The parameters refer to the template type and are termed the *template formals*.

1.5 C++ and object-oriented programming

The object-oriented language of choice for this thesis is C++. Although C++ has been on the receiving end of jokes doubting its object-oriented heritage, we can assert that it is an object-oriented language since it provides language support for class-based object description, inheritance and polymorphism. C++ lacks some features of other object-oriented languages; most notably dynamic typing and garbage collection of objects, such as are present in Eiffel [74], Smalltalk [43] and others. However, C++ is our language of choice because:

- C++ is efficient, and since we are concerned with parallel performance, efficiency is of great importance.
- The features that C++ lacks can be simulated through *meta-features* [31].
- C++ compilers are more widely available for parallel architectures.

The second item will be elaborated in subsequent sections.

C++ itself is an example of a strongly typed object-oriented language. Thus method invocations are mainly statically bound, with type-checking being performed at compile-time. Run-time bindings are made through the use of *virtual functions*, which involve table look-up through a pointer contained within an object. The pointer is arranged to point at the table which belongs to the object's real class, regardless of interface. This mechanism is relatively fast, only involving two extra memory references over a normal function invocation.

1.5.1 Meta-features

In this section we look at techniques for simulating, in C++, some of the desirable features of other object-oriented languages.

1.5.1.1 Reference counting

In C++, object memory-management is explicit. A new object is allocated space from free store, or the stack, and initialized through the use of its *constructor* method. An object is deleted by invoking its destructor and then freeing its store allocation. Objects are not garbage collected and therefore, by default, an object is copied by creating a new object and copying it on a member-by-member basis.

```
class Letter {
    friend class Envelope;
  protected:
    Objs data;
    int reference_count;
    void foo();
};

class Envelope {
  protected:
    Letter* rep;                                           // the contained letter
    void foo() { return rep→foo(); }   // a forwarded function
};
```

Figure 1.1: The Letter and Envelope idiom

The figure shows a letter/envelope class pair. The envelope class contains a pointer to a letter class. Any letter type functions applied to the envelope are passed on to the letter through this pointer. The letter also contains a reference count which is manipulated by the envelope's copy constructor and assignment operator (not shown).

Member-wise copying is compute intensive for large objects and not ideal from a performance point-of-view. In order to get around this, we can arrange for objects to be reference-counted [31, p58]. Thus, when an object is copied, all that is created is a new reference, with some counter in the actual object being updated to reflect the additional reference to it. When a reference is deleted, the reference count of the real object is decremented and the object removed if the count is zero. Obviously, some care has to be taken when an object really needs to be duplicated, but in the majority of cases this is not necessary.

Perhaps the most aesthetic way of defining reference counted objects in C++ is by using Coplien's envelope and letter idiom [31]. With this scheme the object holding the actual data, the *letter*, is referred to by the object that the user actually sees, the *envelope*. See figure 1.1. The letter holds the reference count, and the envelope does all reference manipulation. Class methods can be performed either by the envelope or by the letter through *delegation*, albeit explicit. In this case the envelope forwards messages to the letter.

1.5.1.2 Weak typing

In pure object-oriented languages, variables are run-time bindings to objects that act like labels [31, p134]. Assigning a variable to an object is analogous to putting a label on it. Assignment simply changes the labels. Variables have little or no type information associated with them, and little is done by the compiler to ensure compatibility between types. It is only at run-time that type incompatibilities show up through run-time checking. Obviously this has error elimination disadvantages but has the big advantage of flexibility.

```
class Number {
  protected:
    union {
        Number *rep;                                        // the letter
        int reference_count;
    };                                     // forwarded functions are virtual
    virtual int foo() { return rep→foo(); }
};

class Complex : public Number {                        // an example subclass
    double rpart, ipart;
    int foo();
};
```

Figure 1.2: An Envelope class with delegated polymorphism

The figure shows how a letter class can be its own envelope. This allows delegated functions to be virtual, and thus the letter polymorphic.

To achieve this flexibility in C++ we need to add a level of indirection which simulates this label and object separation. This can be achieved by using the envelope and letter idiom given in section 1.5.1.1, but by making the letter the same class as the envelope. The different classes that we require can be derived from this unified class, and letters can be interchanged without breaking the C++ type system. Figure 1.2 shows the scheme in outline but a fuller description can be found in Coplien [31, pp138–140].

The base class `Number` provides all the functionality that will be required by any subclass of `Number` and method invocations on `Number` are forwarded to the actual object. The scheme actually works because the forwarded functions are `virtual` and therefore the right function is selected at run-time.

This system cannot be an exact replica of untyped variables in languages such as Smalltalk. This is because objects defined in this way still have a generic type which defines the union of all messages that the object can possibly accept. Messages outside of this set will be rejected by the compiler as a type / method incompatibility: we cannot define new methods for an object which are not declared in its envelope. Thus we can only make a run-time type change to an object of the same generic type. This is actually a desirable feature - we do not require an object of Number class to accept messages that apply to Blancmange!

The flexibility we gain is run-time type changes not arbitrary run-time message bindings.

1.5.1.3 Virtual object construction

C++ provides virtual functions for dynamic binding of messages to methods. Calling a virtual function is like saying "apply this function to the object that X *really* is", where X is the interface to which a message has been sent. Destructors too can be virtual - "delete the object that X *really* is". Virtual constructors, however, make no sense; when an object is being constructed; it has no hidden type. However, type selection can be made on the basis of the arguments that are passed to the constructor call. Although C++ doesn't provide direct language support to do this, it is again possible to emulate the feature using the idioms outlined previously [31, ch. 8].

```
class Number {
  public:
/*
 * make a Double or Complex depending on context
 */
    Number(double d) { rep = new Double(d); }
    Number(double i, double j) { rep = new Complex(i,j); }

  protected:
    Number *rep;                                          // the letter
};

class Complex : public Number {                           // an example subclass
    double rpart, ipart;
  public:
    Complex (double i, double j) : rpart(i), ipart(j) {}
};

class Double : public Number {                            // another subclass
    double value;
  public:
    Double (double i) value(i) {}
};
```

Figure 1.3: Virtual object construction

> *The figure shows how the envelope/letter idiom can be used to construct objects, the type of which is selected at run-time. In the example there are two letter type objects,* `Complex` *and* `Double`*. The class* `Number` *will make its letter be one of these depending on its construction arguments. Since envelope functions will be delegated polymorphically to the letter, there will be no type conflict at run-time.*

Using the weak typing idiom we can arrange for an object under construction to make some sort of type selection decision and then to change its type accordingly. The type decision can be made through the use of overloaded constructors or through a type field. See figure 1.3.

The main problem with this approach is that of extensibility. What happens when a new type is required? Either the envelope interface has to be modified to accommodate a new constructor, or a new type must be added to the type field. In either case modifications are required which affect the interface, and this is unacceptable from an architecture and modularity point-of-view. The answer is to maintain a list of objects of the different types, into which new types can register. An object is constructed by presenting each object in the list with the constructor message and continuing in this way until the message has been accepted. The accepting object returns a new object of the correct type. Since it is impossible to determine beforehand the type of a new list object, the list is one of *interfaces*. In fact this is why the scheme works; dynamic bindings are made, but on *existing* objects. The immutability of the interface is preserved since registration merely means updating the list, which is a run-time action.

C++ is a *dual-hierarchy* language meaning that the stuff from which objects are made (classes) are distinct from the objects themselves. Creating objects from other objects, as in the case above, is a *single-hierarchy* principle and the prototypical objects - in this case the members of the list - are called *exemplars*.

1.5.2 Summary

Having examined both parallel programming and object-oriented languages we now look at the synthesis of the two. Concurrency in object-oriented languages is a concept almost as old as that of object-orientation. We examine some of the approaches to concurrency in object-oriented languages and, more specifically, concurrency techniques in C++.

1.6 Parallel object-oriented programming

Parallel object-oriented languages have been around for a long time. Their aim is not necessarily performance related. As described previously, object-oriented languages deal with entity relationships that exist in the application domain. One of the most common features of entities in the real world[8] is their action independence. Modelling this independence of actions naturally gives rise to multi-threaded, or concurrent, objects – objects which can undergo state transitions without recourse to a sequential thread of control. Concurrent objects are necessarily shared or distributed.

Using this approach to describe entity relationships is extremely powerful as it takes the solution domain ever closer to the application domain. There is a drawback however. Having multi-threaded objects amounts to object-based parallel processing, and thus suffers from all the usual problems of parallel processing – deadlock, mutual exclusion, etc.

Concurrent object-oriented languages generally provide several language enhancements to enable management of the complexity introduced by concurrent objects [15]. Some of the most widespread features are:

Active and passive objects. If objects are distributed and communicating with each other, there are two possible ways of managing the threads of control. Passive objects have an effectively sequential thread of control. Objects have separate threads of control but there is only ever *one* active thread. All message transfers are synchronous and base on remote procedure calls, so that a calling object will block until a result is returned. This type of concurrency is simple to manage and is employed by some object-oriented languages and operating systems [78].

[8] Note that not all application domains are in the real world

Active objects have multiple, independent threads of control. This means that true parallelism can occur in the system. The degree of sophistication varies greatly.

The next items generally only refer to active object systems. Chin gives a good overview of the different possibilities with active and passive systems in [27].

Monitor locks. Semaphores [34] ensure that critical sections of code[9] are protected from multiple accesses. In object-oriented programming, concurrent execution implies concurrent access, and we generally want to prevent multiple accesses to objects which involve state change. Thus we provide a special type of class - a *monitor* - which ensures that only one object method is operating at any one time.

Synchronization. Most explicitly concurrent object-oriented languages provide synchronization primitives. These primitives specify ways in which threads of control can synchronize. Primitives usually exist for *barrier synchronization* where processes block until all have rendezvoused with each other; *acceptance* which means a thread conditionally executes depending on the accepted message; *continuations* which means threads block until some condition is true; and *futures* which mean threads conditionally start some time in the future. Many languages support these [15, 23] but we note that parallelism is always explicit.

1.6.1 Implicit parallelism

In this thesis we are concerned with implicit parallelism. Thus the techniques above are of interest as implementation technologies, but not as user presentable features. Implicitly parallel object-oriented languages are generally extremely complex and are characterized by compilation and run-time techniques such as:

Dependency analysis. In an object-oriented program all method invocations are candidates for concurrent execution. However, a compiler must determine when methods are required, so that it only schedules concurrently those tasks which it would be advantageous to. It must also ensure that execution blocks when a result is required but has not yet been returned by another thread.

Locality analysis. When possible concurrency has been determined, a compiler must ensure that concurrent tasks which share data are scheduled on the same processor if possible. If this is not possible then the run-time system must ensure that shared-data is coherent between processors.

Type analysis. In order to determine whether objects share data it is necessary to perform detailed analysis of the object types in a program.

Scheduling. A run-time system must ensure that objects are distributed sensibly across a processor network, and make decisions about relative tradeoffs in scheduling tasks. All of the above points also have a run-time element which needs to be addressed by the system. The scheduler needs to take all these factors into account when placing objects.

These various techniques, and more, are addressed by the *Mentat* [44, 48, 47, 45, 46] and *Concert* [25, 22, 26, 84, 65] systems. The *Concert* system is particularly interesting

[9] i.e. those which not more than one process must be evaluating at any one time.

in that it attempts to optimize the concurrency grain size to improve performance. The forerunner to *Concert* was Chien's *Concurrent Aggregates* [23, 24]

1.6.2 C++ as a concurrent object-oriented language

C++ provides no language support for concurrency within its object paradigm. However, as with other possible object-oriented features, creating concurrent object support within the language is not an overwhelmingly complex task. Accept constructs are merely a language formalism for daemon-style programs; monitor support can be provided on a per-class basis, although entering and exiting the lock has to be done through explicit function calls; active and passive objects are simple to provide if the underlying operating system supports multi or parallel threading.

In fact some would argue that multi-threaded objects are purely a language formalism for a concept that belongs, and should stay, in the operating system domain. If an operating system does not provide support for these concepts then any concurrent object-oriented system will have to provide the support itself, essentially augmenting the operating system. If the operating system *does* provide support for threading then this can be incorporated into an object-oriented language through the creation of suitable libraries. The *Presto* system [11] amply demonstrates the viability of this approach.

1.6.2.1 Concurrent extensions to C++

Many concurrent extensions to C++ exist [15] and appear useful in the right context. However, proposals to extend the emerging ANSI C++ standard to include concurrency support have been dropped, and it is worth considering why.

Many people would like to see different extensions incorporated into the C++ language, and for every feature that C++ provides, there are many other variants of increasing sophistication. However, the features that are added to the language are those which will benefit a significant proportion of programmers on a significant range of applications without affecting those who do not wish to take advantage of the new feature. Concurrency mechanisms do not fulfil either of these goals since concurrency remains the domain of specialized applications, and concurrency implies a run-time system which will burden programmers who do not wish to take advantage of concurrency features. In addition concurrency features do not contribute a great deal more, apart from syntactic sugar, than library-based concurrent extensions, (excepting the Mentat approach).

If the language standard cannot justify an extension, then there can be little justification for creating a dialect to support one. Creating a dialect has a whole host of problems related to the need for writing a compiler to support the dialect; problems such as tracking the original language definition as well as the dialect.

1.6.2.2 Library-based concurrency mechanisms

In any parallel system we have to address two issues: processes and communication. Without different processes there is no parallel system. Without communication there is little point in having one. Parallel processes need to communicate with each other at some point in their lifetime. How can these issues be addressed in an object-oriented system without language support?

Processes. Most problems associated with processes are to do with creation, starting and stopping. As it is, the mechanics of this are very difficult without operating system support, although a co-routine style of programming can be used [104]. With operating system support, starting and stopping a process is relatively easy and fits neatly, conceptually, into the role of object constructors and destructors. These and other functions can be incorporated into a process superclass from which all classes needing process characteristics can be derived. In fact the concept of a process as an object is a direct application of object-oriented methodology rather than a concept needing specialized language attention.

Communication. Shared memory parallel processes usually communicate through shared memory. Thus all that is required to support this type of communication is some method of ensuring mutual-exclusion between critical sections of code. If an operating system provides support for parallel processes then it is most likely to provide mutual-exclusion mechanisms. All that is then required is for process classes to enforce use of these mechanisms for invocations of their methods. Unfortunately, C++ does not provide a method for generically enforcing guards (section 1.4.8) and thus mutual-exclusion calls must be explicit. This is perhaps one area where language support would be preferable, although not essential.

Processes in a message-passing parallel environment generally communicate in a stream-based fashion. This is merely an input / output operation, and so a classic candidate for using overloaded operators to ensure type-safety and give notational convenience [95]. Thus we can write:

```
processA << abyte;
```

meaning "transmit a byte of data to process A".

1.6.3 Object concurrency for performance

So far we have discussed concurrent object-oriented technology in general terms. We have argued that perhaps support for concurrency within object-oriented languages is not as essential as it first appears. Concurrency purports to bring object-oriented design closer to the application domain, but introduces additional problems with system design. On the rare occasions where concurrency is essential, library-based mechanisms can be used to provide the desired functionality albeit at the cost of convenience.

However, do language mechanisms add anything for applications which are performance directed, i.e. where concurrency is the object of the exercise? Certainly explicit parallel constructs provide syntactic sugar for writing explicitly parallel programs, but this does not make the hard problems of parallel programming go away. Sophisticated concurrent object-oriented systems such as Mentat and Concert take the view that all messages are naturally parallel, however, true as this may be it takes an extremely fine-grained approach to parallel processing, an approach that is not suited to stock hardware. Although much of the Concert project is concerned with increasing this granularity through sophisticated compiler and run-time techniques, there is still *no guarantee* of usable parallelism. Mentat takes a somewhat more coarse-grained approach but parallel objects are explicit.

Is there any other way that we can use object-oriented techniques, in a concurrent context, to *guarantee* efficient, implicit parallel performance? This thesis looks at a way of doing this, and the basis of the technique is outlined in the next chapter.

Chapter 2

Object-oriented divide-and-conquer

Veni, vidi, vici. *Julius Caesar*

Practical parallel computing has been a reality for an appreciable number of years. Research in this area has matured and significant advances have been made, especially with regard to hardware organisation and architecture. For MIMD computers, however, managing and programming for concurrent performance has proved, with the obvious exception of data-parallel programming, to be a difficult and largely intractable problem [70].

Object-oriented concurrent programming, in various forms and levels of sophistication, has been proposed as an answer to this intractability. The debate continues - passive objects, active objects, threaded message invocations, remote procedure calls [27] - though there is some agreement that concurrency mechanisms make object-oriented languages better at modelling real-world situations.

However, it has been pointed out that the heart of object-oriented programming is orthogonal to concurrency [70]. Although the general object-oriented programming model bears similarities to the concurrent programming model, programming with an object-oriented discipline does not necessarily generate the speedup that marks the holy grail of concurrent programming. If concurrency issues are addressed, most notably locality of reference [68], then object-oriented programming can yield the performance that one requires, but to constrain the programmer in this fashion is to contradict the supposed benefits of coupling object-oriented programming with concurrent programming.

An object-oriented program can be rather like a tangled mess of threads from a concurrency point-of-view. Although each thread exhibits significant concurrency with other threads, the whole is not separable. If the whole is not separable then large-grain evaluation cannot be performed in an efficient way, and even fine-grained evaluation has proven tricky [25]. As we discussed in chapter 1, significant research has produced compiler techniques for separating large grains. However, for good performance this still relies, to some degree, on the extant *structure* of the program concerned.

Many concurrent programming systems realize the need for the programmer to play a significant role in identifying concurrency and allowing their design decisions to be influenced by concurrency issues [45]. Yet, all too often, object-oriented concurrent programming is mooted as a panacea for the problems of concurrent programming, placing no constraints on the programmer apart from the discipline of object-oriented programming.

Experience shows us that taking advantage of the programmer's skill in grasping the structure of a problem invariably reveals levels of concurrency that would have been missed by an automatic system. However, allowing the programmer to make explicit parallelism decisions is not the only way of capitalising on this skill. Instead, constraints can be placed on the programmer which, if satisfied, are guaranteed to yield usable concurrency. Parallel functional languages operate in this way. In a similar manner object-oriented programming can be constrained through the use of the D&C algorithm [81].

2.0.4 Chapter organisation

In section 2.1 we discuss the basis of the object-oriented approach to D&C. In section 2.2 we discuss the advantages of this approach and how such a system would fit into an object-oriented programming system. We also discuss techniques for facilitating this integration. In section 2.3 we discuss the back-propagation algorithm and a naive implementation of it under our D&C regime. Finally in section 2.4 we look at how well this implementation performs on a parallel machine.

2.1 Object Oriented Design

In section 1.2 we examined the D&C algorithm and its parallel implementation. We saw that problems cast in a D&C framework readily exhibit parallelism. So, from a parallel point-of-view, D&C is a desirable method for solving problems.

However, the semantics of D&C are obviously not the last word in programming languages. Although Axford [5] and Mou [76] validly argue that D&C is a sufficiently powerful programming construct in its own right, to assert such a view with finality, is to ignore the primary purpose of programming systems. The role of programming systems is one of servitude. Programming languages, CASE tools, debuggers, compilers - all were created to bring the computer's world, of detailed, ordered logic, closer to the programmer's fuzzy notions of real world problems. Thus, in using D&C as our implementing technology for parallelism, we must not ignore the requirement of a programming system that meets the needs of the programmer, rather than those of the computer, operating system or whatever.

Mou's Divacon language [76] was a functional programming language in its own right, rather than purely a medium for writing D&C programs. However, D&C is, by definition, the source of parallelism. Thus, providing the programmer with additional facilities, that have their origins in computational convenience for parallelism, seems rather counter-intuitive. What is really needed are facilities that are directed towards programmer comfort.

Clare [29] attempted this by generalizing McBurney and Sleep's [72] work. By setting D&C in a C environment, Clare was, to a degree, addressing both the programmer's preferences, and the requirement for ready parallelism. However, using the C language resulted in a system that appears somewhat onerous and inflexible from a programmer's point-of-view. The system required the definition of many functions, and had no means of evaluating more than one D&C problem in any given program. Mou's system was at least not limited in the number of D&C problems it could evaluate. To implement this genericness in Clare's system, would probably involve a complicated system of pointers to functions, and be a fairly arduous affair given the limits of C in this area. However, let us consider what is implicitly being accomplished through this system of C functions, and what is required to extend it.

The C functions of Clare's system, purely provided an *interface*, between the run-time evaluator, and the programmer's D&C specification of the problem in question. To the run-time system these functions were an abstraction of the D&C problem. For each different problem the abstraction that the run-time system saw was constant. In order to evaluate different problems in a single program, this abstraction needs to be preserved for each problem specification.

And what of the problem itself? Each problem is likely to consist of state as well as function. Does the mere specification of functions yield a clear encapsulation of a problem? In fact some of Clare's functions represented *state*, since they were concerned with I/O of data related to the problem. Encapsulation is clearly a requirement, and so is the genericness of abstractions. But, these two requirements are *exactly* what object-oriented programming languages provide. Thus it would seem advisable to use an object-oriented language as the technology around which to base a D&C system. Also the design of object-oriented languages is motivated by programmer needs, and therefore meets our requirement for a system that is geared towards programmer comfort. Finally, object-oriented languages generally enable the seamless addition of new features, without recourse to compiler writing.

2.1.1 Basic D&C

In object-oriented languages, the object is the unit of abstraction and encapsulation. Thus, in order to encapsulate problems in well-defined entities, we want to represent individual D&C problems as *objects*. Additionally, we want to characterize each of these objects with the same D&C functionality. This means that each object presents the same public interface. Finally, we want the D&C functionality to be dynamically bound, so that D&C objects are *polymorphic* with respect to the D&C functionality. This means that each object can have a different internal structure, but that the manipulation of such objects is constant.

Thus, in C++ each D&C object is characterized by the virtual functions `simple()`, `divide()`, `combine()` and `evaluate()`, representing the D&C primary functions *simple()*, *divide()*, *combine()* and *evaluate()* of section 1.2.

Such a definition outlines, in general terms, the basis of the approach. However, a practical implementation requires a lot more detail, and many other issues need to be discussed and resolved. In the remainder of this chapter we will discuss some of these issues, and a naive implementation that addresses them. In chapter 3 we will describe a more sophisticated and elegant implementation, that resolves these issues satisfactorily.

2.1.2 Data Representation

In the evaluation of any D&C problem, there will be an underlying data model that will be the subject of the D&C primary functions. If we are to use more than one D&C object in the evaluation of a problem, then we require these objects to have the same characterization, so that, in our strongly typed environment, each may access the other's underlying data relatively easily. This is also true for the various individual D&C functions which need to be able to access the underlying data.

For balanced D&C where the computational demand is determined by the problem size, a vector characterization would appear most useful. With this structure, data sub-

```
template <class T> class Divacon {
  public:
    virtual T divideUpper (const T&) =0;
    virtual T divideLower (const T&) =0;
    virtual T combine (const T&, const T&) =0;
    virtual T evaluate (const T&) =0;
    virtual const boolean simple (const T&) const =0;
};
```

Figure 2.1: A basic D&C class structure

The figure shows a basic C++ interface for implementing D&C objects. The class is parameterized over the argument T which represents the structure being manipulated.
Note in these examples the use of `divideUpper()` *and* `divideLower()`. *These two functions perform a binary partitioning of the problem. In later examples we drop these in favour of a single* `divide()` *function which was found to be more flexible.*

```
struct MatrixData_t {
    unsigned int rows;
    unsigned int columns;
    unsigned int row_position;
    unsigned int column_position;
    double* data;
};

class MatrixDivacon : public Divacon<MatrixData_t> {
  public:
    virtual MatrixData_t divideUpper (const MatrixData_t&);
    virtual MatrixData_t divideLower (const MatrixData_t&);
    virtual MatrixData_t combine (const MatrixData_t&, const MatrixData_t&);
    virtual MatrixData_t evaluate (const MatrixData_t&);
    virtual const boolean simple (const MatrixData_t&);
};
```

Figure 2.2: A D&C class representing matrices

The figure shows the use of the generic D&C interface presented earlier. The example class uses `MatrixData_t`, *representing matrices, to hold the current state of data.*

units can be directly extracted rather than having to be traversed - which would be necessary for a list based structure. Additionally, larger sub-units could be constrained to be contiguous in memory so that I/O can be performed with a minimum of processor cycles[1].

```
template <class T> class Divacon {
  public:
    virtual const boolean simple (const T&) const =0;
    virtual T divideUpper (const T& ) =0 ;
    virtual T divideLower (const T& ) =0 ;
    virtual T combine (const T&, const T&) =0 ;
    virtual T evaluate (const T& ) =0 ;

    virtual void outputData (const T&, const Process&)=0;
    virtual void outputResults (const Process&)=0;
    virtual T inputData (const Process&) =0;
    virtual T inputResults (const T&, const Process&)=0;
};
```

Figure 2.3: A D&C class with input / output functionality

The figure illustrates the additional functions needed to make a D&C object usable by a parallel run-time system. There is no implicit language support for object I/O so this must be provided explicitly. Note that the D&C object is merely a functional provider for objects with structure T.

Alternative characterizations are linked lists or binary trees. Linked lists would be especially useful for complicated `combine()` or `divide()` functions, since the data sub-units could be rearranged very easily. However, the overhead, in traversing the list to extract sub-units and to perform I/O, is unnecessary for many problems.

McBurney and Sleep shared matrix structures between D&C operations. This meant that data was not duplicated in the `divide()` stage, and so memory was conserved. This is especially important for matrix multiplication and other algorithms which are not data-parallel, as the splitting up of the data involves a certain amount of duplication. In cases such as these, it would seem sensible to use, as much as possible, a description of the data, rather than the data itself.

Thus, as a first approach, we parameterize D&C objects with a representation of the underlying data structure. See figure 2.1.

This framework provides an interface to the problem through which the graph-reduction routines can operate. As such the interface can provide no information about the underlying problem specification, derived classes will build up this information.

For a problem involving matrices the class derivation would be as shown in figure 2.2. Whatever functionality is known at this stage can be added to the `MatrixDivacon` class, and the complete functionality added in derived classes e.g `MatrixMultDivacon`.

In order to communicate D&C objects between processors, we require additional functionality. The result is shown in figure 2.3.

[1]Recall that transputers can perform inter-processor data transfers via DMA, but only on contiguous data.

2.1.3 Run-time evaluation

We have outlined the structure of D&C problems in an object-oriented context. Such structures are provided by the applications programmer. However, such structures are only of any use when used with a run-time system for evaluation and scheduling.

For our naive first approach we structure the evaluation in the obvious fashion: as a binary tree. Since the scheduling benefits of D&C are to be found in depth-first expansion (section 1.3.1), we require a structure that is going to maintain the internal state of this type of recursive evaluation. Since the evaluation is tree-like in nature, actually adopting a tree structure will certainly provide the features we require. At each stage in the expansion of the tree, the D&C object being evaluated is used to manipulate the data representation of section 2.1.2. Each tree node will then contain a representation of the data and the current state of that data. Nodes that are not currently being processed are put into a queue, then nodes for parallel evaluation are taken from the front of this queue. This particular scheduling strategy has been discussed already in section 1.3.

Each system component can also be constructed in an object-oriented fashion, yielding a modular, easily maintainable system [98]. Where possible, predefined classes can be used for system construction.

In chapter 3 we will give a full system description.

2.2 The Object Oriented Approach

We have described object-oriented D&C objects for parallel evaluation. We have asserted that object-oriented programming forms a viable framework for expressing the D&C paradigm elegantly and flexibly, and for building a D&C system. However, object-oriented programming also provides the means for achieving an integrated approach to parallel processing.

2.2.1 Motivation

In most engineering disciplines, real problems are characterized by their complexity and their non-uniformity. Given a problem, there are usually many steps involved in solving that problem. Computationally, the requirements of a solution are usually diverse and multi-faceted. For instance, to recognize human speech usually requires some combination of acquisition, filtering, preprocessing, recognition, postprocessing and lexical-access. With today's problems, any computational system must be able to provide sufficient flexibility to accommodate steps such as these, whilst maintaining efficiency as much as possible. It is possible to be overly concerned with the efficient solution of very specific problems, and give insufficient thought to the cost of integration of these solutions into a real system. Efficiency must be measured in software life-cycle terms as well as actual algorithmic efficiency.

D&C provides the flexibility necessary to accommodate a large problem solution set. Object-oriented programming provides the means for integrating efficiently – in all senses of the word – these various solutions to make up a complete solution.

2.2.2 Integration using object-oriented programming

It has already been described in section 1.6.2, how data abstraction and operator overloading in object-oriented languages allow the complexity of parallel processing to be concealed

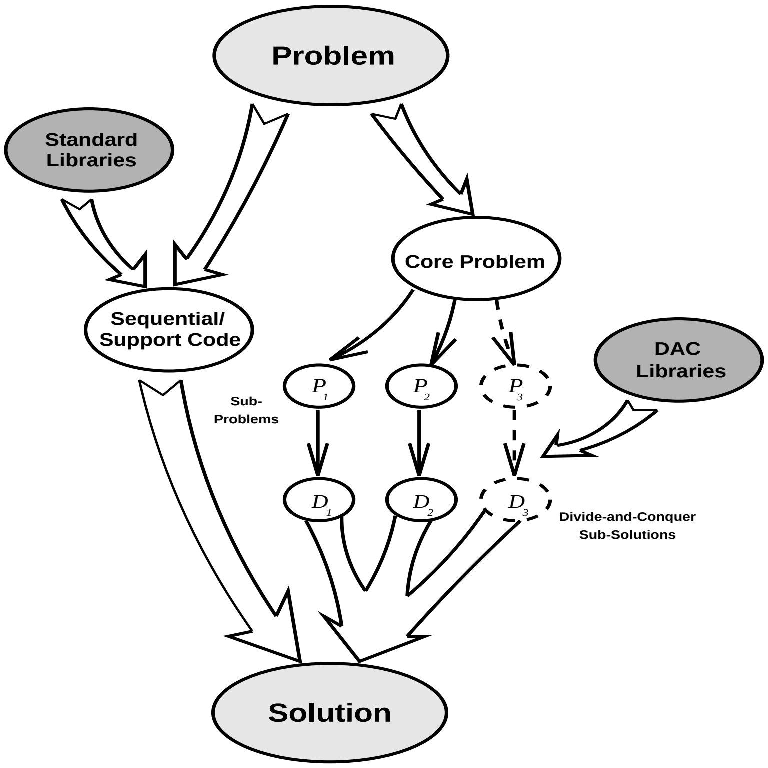


Figure 2.4: The Design Process

The figure shows a possible scheme for implementing parallel programs using D&C. A combination of D&C solutions, D&C libraries, standard C++ libraries and serial code is used. D&C only fulfils its potential when used in this context.

behind the application programmer's interface (API). Abstraction, together with genericity and encapsulation of functionality, allows libraries of objects to be efficiently created and easily used. However, as described above, the interaction between these various objects and methods is as important as the objects and methods themselves. If this interaction is poor then methods that are individually extremely efficient can be next to useless, leading to poor overall system efficiency.

With the parallel evaluation of D&C methods, it is crucial that the number of D&C passes[2] is reduced to a minimum so that the communication cost exacted by such a system is minimized. Data dependencies may mean that computation cannot all be performed in one pass, but the number of times this is necessary must be kept low.

2.2.3 Sequential-parallel and parallel-parallel integration

Parallelized object methods can be made transparent to the user using appropriate functions and overloaded infix operators; for instance `MatrixMultDivacon` can be integrated into the `operator*`. This allows the parallel and serial aspects of a problem to

[2] A D&C pass is D&C evaluation involving some form of data broadcast, with all the data being present on the root processor at the beginning and end of the evaluation.

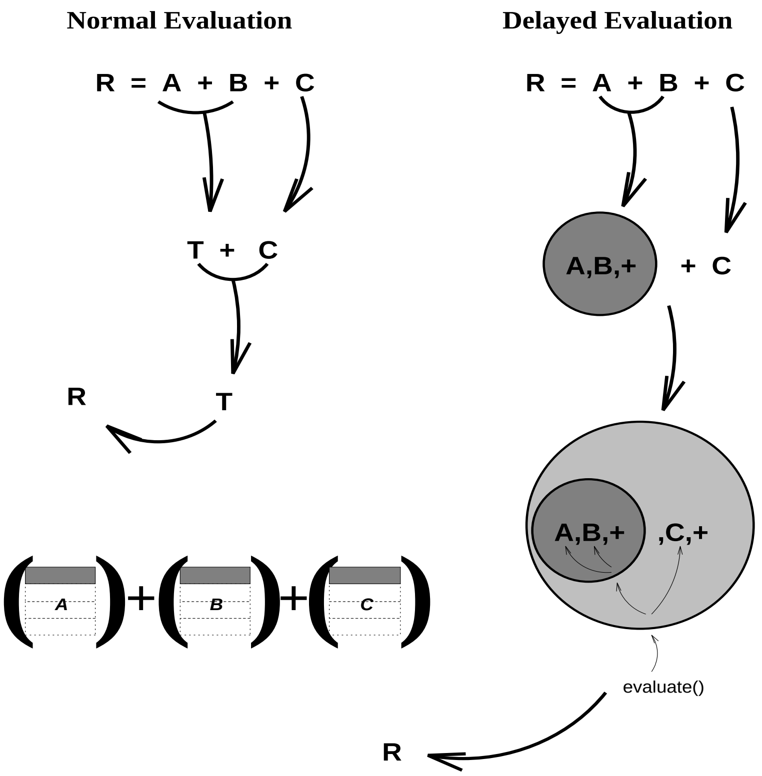


Figure 2.5: Delayed evaluation

The figure compares standard evaluation with delayed evaluation. The circles represent objects, and the letters data. In the example three matrices are being added together. Delayed evaluation ensures that in a parallel context data is distributed only once.The three shaded segments of the matrices will all be distributed to a single processor, rather than distributed, evaluated and returned in pairs.

be integrated seamlessly without the parallel aspects interfering with the overall solution strategy.

This also means that several parallel methods can be grouped together to yield a more complex method. The problem with small sub-units being combined to yield a larger solution is that this can lead to unnecessary inefficiencies. To take a trivial example: it is inappropriate to compute, using D&C, many pairs of matrix operations separately. $constant * A + B$ computed as $constant * A$ and then $result + B$ would involve a tremendous amount of unnecessary communication. The data would be broadcast or retrieved twice, when all that is needed is for all data to be broadcast, and each result element to be calculated as $constant * A_i + B_i$. As communication is the overriding overhead in MIMD parallel processing, we need to achieve the latter situation, but without explicit programmer intervention. This can be achieved, whilst still maintaining the notational convenience of operator overloading, by using delayed evaluation [33]:

This trivial example serves to illustrate the method of delayed evaluation.

```
class Data;
class AddedData;
typedef float type;

class Base {                          // base class - provides an evaluation interface
  public:
    virtual type evaluate()=0;
    Base& operator+ (Base& b1);
};

class Data : public Base {
  public:
    Data(const type t=0.0) : data(t) {}
    Base& operator= (Base& b1) {
        data = b1.evaluate();
        return b1;
    }
    type evaluate() { return data; };
  private:
    type data;
};
    // generic compund
class Compound : public Base {
    friend class Base;
  public:
    type evaluate() {                           // foo is some arbitrary function
        return foo( bp1→evaluate(), bp2→evaluate() );
    }
  protected:
    Compound(Base* b1, Base* b2) : bp1(b1), bp2(b2) {}

    Base* bp1;
    Base* bp2;
};
    // added data compund
class AddedData : public Compound {
    friend class Base;
  public:
    type evaluate() { return bp1→evaluate() + bp2→evaluate(); }
  protected:
    AddedData(Base* b1, Base* b2) : Compound(b1,b2) {}
};

Base& Base::operator+ (Base& b1) { return *new AddedData(this, &b1); }

main()
{
    Data a(1.0), b(2.0), c(3.0), r;
    r = a+b+c;
}
```

Figure 2.6: Example of delayed evaluation

The figure shows a working example of delayed evaluation. The three `Data` *objects a,b and c are added together by first forming a compound, using* `AddedData`*, of AddedData[AddedData[a,b],c]. This compound in then evaluated in* `Data`*'s* `operator=`*.*

2.2.4 Aggregate D&C objects

Delayed evaluation is characterized by aggregate objects. An aggregate is an object having additional objects as part of its internal structure. In delayed evaluation each aggregate usually represents an operation, the operands of which are contained within the aggregate. Operands themselves can be aggregates as well. In a sequential context, every object is characterized by its derivation from a single base object which specifies an interface to basic functionality. Compound objects have the basic form given in figure 2.6.

The function `foo()` will vary depending on the type of compound object. For instance if the object is an `AddedData` object then `foo()` will be the infix `operator+`. The infix `operator+` will then be overloaded for `Base` objects. Some appropriate garbage collection is used to release the `AddedData` object when it has been used. Objects that actually contain data are a special case and `evaluate()` simply returns the data itself.

This approach results in the construction of an evaluation tree. The tree can be pruned to eliminate temporaries and unnecessary operations. For example $A^T * B$ can be evaluated without calculating A^T. The tree is then evaluated by overloading the `operator=` so that it calls the root object's `evaluate()` function. This call recurses down the tree until `evaluate()` returns plain `Data`. Then as the recursion unwinds, evaluation takes place.

This is essentially how Davies' Newmat [33] package works.

2.2.4.1 D&C and delayed evaluation

For normal delayed evaluation, each object is characterized by the function `evaluate()`. Now, with our D&C objects, each is characterized by the D&C primary and auxiliary functions. Thus to extend delayed evaluation to D&C objects, we merely create aggregates with the same characterization. To evaluate the aggregates we recursively call the D&C functionality, instead of just `evaluate()`. Thus, in the addition of two matrices for example, `divide()` for `operator+` calls `divide()` for its two operands. The operands then divide as appropriate. A similar procedure applies for the other D&C functions. Evaluation is triggered by `operator=`, just as for the serial case.

Of course, this integration will only work within the constraints of one D&C pass, and it is essential that the maximum possible computation is performed within each pass. Unlike the sequential example, data dependencies will mean that the evaluation tree can rarely be evaluated in one pass. It is therefore possible to envisage an evaluation scheme in which nodes of the evaluation tree are identified, at which no further single pass evaluation can be performed. When these nodes are identified D&C processing takes place, rather than just in the `operator=`.

2.2.5 Problem partitioning with D&C objects

Rabhi [88] identified problem partitioning as crucial to achieving optimal speedups for balanced D&C systems. The compound object scheme as described above, means that partitions can be specified for each D&C operation[3]. For the compound `simple()` function, the overall partition can be specified as a suitable combination[4], of the individual operation partitions.

[3] For instance matrix multiplication

[4] Possibly the minimum.

2.2.6 Summary

The integration of parallel methods in this context can be achieved in an efficient and flexible way for some problems. These ideas are denoted schematically by Figure 2.4.

We now look at a typical problem, and how it can be implemented using object-oriented D&C.

2.3 The Back-Propagation Algorithm

Given a three-layer perceptron [90] with I input units, J hidden units and K output units, the output $\mathbf{o}^K$ is related to the input σ^I by:

$$\mathbf{o}^K = f_K((\mathbf{W}^{JK})^T f_J((\mathbf{W}^{IJ})^T \sigma^I)) \tag{2.1}$$

Where $f_J()$ and $f_K()$ are discriminant functions and $\mathbf{W}^{IJ}$ and $\mathbf{W}^{JK}$ are weight matrices. This is the feed forward equation. The back-propagation of the error gives weight updates:

$$\Delta\mathbf{W}^{JK} = \eta\mathbf{o}^J((\mathbf{t}^K - \mathbf{o}^K) \otimes f'_K(\sigma^K))^T \tag{2.2}$$

$$\Delta\mathbf{W}^{IJ} = \eta\mathbf{o}^I(\mathbf{W}^{JK}[(\mathbf{t}^K - \mathbf{o}^K) \otimes f'_K(\sigma^K)] \otimes f'_J(\sigma^J))^T \tag{2.3}$$

where $\otimes$ is elementwise multiply, $f'()$ is the derivative of $f()$ and η is the learning rate. $\mathbf{t}^K$ is the target output.

2.3.1 Approaches

It is now necessary to consider how this problem can be broken down modularly in a D&C fashion. There are two options:

- If the network is very large then the algorithm itself can be broken down using D&C and basic matrix operations. This is not necessarily highly efficient. This will be denoted as a vertical implementation.
- If the network is small then the whole algorithm can be kept on each node and different weight updates can be calculated on each node and the results combined. This is a common way to implement parallel back-propagation - to run the entire training set through the network and to sum the weight updates. This has the advantage of eliminating random noise. This will be denoted as a horizontal implementation.

Let us consider the second possibility first as this is likely to be the easiest one to implement.

2.3.2 Notation

Operations will be defined in terms of pseudo-mappings between sets. These pseudo-mappings will then be qualified. Thus

$$\begin{aligned} divide() : \mathbf{S} &\rightarrow \{\mathbf{S}_1, \mathbf{S}_2\}\ WHERE \\ &\quad < \text{define } \mathbf{S}_1 \text{ and } \mathbf{S}_2 \text{ in terms of } \mathbf{S} > \\ simple() : \mathbf{S} &\rightarrow < \text{boolean expression} > \end{aligned}$$

$$
\begin{array}{rcl}
evaluate(): \mathbf{S} & \rightarrow & \mathbf{S}' \ WHERE \\
 & & < \text{qualification of the mapping} > \\
combine(): \{\mathbf{S}_1, \mathbf{S}_2\} & \rightarrow & \mathbf{S} \ WHERE \\
 & & < \ \text{define } \mathbf{S} \text{ in terms of } \mathbf{S}_1 \text{ and } \mathbf{S}_2 >
\end{array}
$$

Representations and their equivalents are as follows:

$$
\begin{array}{rll}
\text{Vectors:} & \mathbf{v} & \equiv \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} \equiv (v_1, \ldots, v_n)^T \\
\text{Matrices:} & \mathbf{M} & \equiv \begin{pmatrix} \mathbf{M}_1 \\ \mathbf{M}_2 \end{pmatrix} \equiv \begin{pmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix} \equiv (\mathbf{v}_1, \ldots, \mathbf{v}_n) \\
\text{Scalars:} & a & \\
\text{Data sets:} & \mathbf{D_S} & \equiv \{\mathbf{M}, \mathbf{v}, a, \ldots\}
\end{array}
\tag{2.4}
$$

2.3.3 Horizontal Back-Propagation

Given a training set of n frames $\mathbf{T_S}(n) = \{f_1, \ldots, f_n\}$, this can be split recursively into groups of frames. Thus

$$
\begin{array}{rcl}
divide(): \mathbf{T_S}(n_s) & \rightarrow & \{\mathbf{T_{S_1}}(n_s/2), \mathbf{T_{S_2}}(n_s/2)\} \ WHERE \\
 & & \mathbf{T_{S_1}}(n_s/2) = \{f_1, \ldots, f_{n_s/2}\} \\
 & & \mathbf{T_{S_2}}(n_s/2) = \{f_{n_s/2+1}, \ldots, f_{n_s}\}
\end{array}
\tag{2.5}
$$

If n_s defines the number of frames in a given set then a straightforward $simple()$ function is when n_s is equal to some suitable $partition$.

$$
simple(): \ \mathbf{T_S}(n_s) \rightarrow n_s = partition \tag{2.6}
$$

The $evaluate()$ function then becomes a cycle of the feed-forward / error back-propagation equations.

The combination function is equally straightforward and involves an addition of the returned weight updates. Thus the result set $\mathbf{R_S}$ for each node is defined by:

$$
\begin{array}{rcl}
combine(): \{\mathbf{R_{S_1}}, \mathbf{R_{S_2}}\} & \rightarrow & \mathbf{R_S} \ WHERE \\
 & & \mathbf{R_S} = \{\Delta\mathbf{W}^{IJ}, \Delta\mathbf{W}^{JK}\} \ WHERE \\
 & & \Delta\mathbf{W}^{IJ} = \Delta\mathbf{W}_1^{IJ} + \Delta\mathbf{W}_2^{IJ} \\
 & & \Delta\mathbf{W}^{JK} = \Delta\mathbf{W}_1^{JK} + \Delta\mathbf{W}_2^{JK}
\end{array}
\tag{2.7}
$$

In addition to these primary functions it is also necessary to define the sets for input / output functions. These will be called $\mathbf{B_S}$ the Broadcast Set and $\mathbf{Re_S}$ the Reel Set. These are:

$$
\mathbf{B_S} = \{\mathbf{W}^{JK}, \mathbf{W}^{IJ}, \mathbf{o}^I, \mathbf{t}^K\} \tag{2.8}
$$

$$
\mathbf{Re_S} = \{\Delta\mathbf{W}^{IJ}, \Delta\mathbf{W}^{JK}\} \tag{2.9}
$$

Note that this approach runs into the problem of lack of memory. With a simplistic implementation the entire training set would need to be initially held on the root node together with all the other associated data. A more sophisticated approach might use some through routing technique so that data was not actually transmitted until needed. Additionally the idea of using the host as the root and initially employing D&C as a scheduling strategy would be helpful here. This would involve dividing the data until the sub-divided parts are small enough to run on the first node, and the evaluation function becomes that of scheduling this load onto the first node for computation.

2.3.4 Vertical Back-Propagation

The second approach is to break down the algorithm itself into D&C format, rather than the data. Only the feed-forward equation will be considered to illustrate the method. It is necessary at this stage to define some new terminology to specify the state of data structures, principally whether the leaf result of a D&C operation is in a column-wise segmented or row-wise segmented state. These will be denoted by the subscripts c and r respectively. Unsubscripted data structures are unsegmented.

By definition $f(\sigma)$ is a data parallel operation, and can thus be computed by D&C by simply splitting the vector along its length. This gives:

$$\begin{aligned}
divide() : \mathbf{v} &\rightarrow \{\mathbf{v}_{1_r}, \mathbf{v}_{2_r}\} \ WHERE \\
& \mathbf{v}_1 = (v_1, \ldots, v_{n/2})^T \\
& \mathbf{v}_1 = (v_{n/2+1}, \ldots, v_n)^T \\
simple() : \mathbf{v} &\rightarrow elems(\mathbf{v}) \leq const \\
evaluate() : \mathbf{v}_h &\rightarrow \mathbf{v}'_r \ WHERE \\
& v'_i = f(v_i) \\
combine() : \{\mathbf{v}_{1_h}, \mathbf{v}_{2_h}\} &\rightarrow \mathbf{v} \ WHERE \\
& \mathbf{v} = (\mathbf{v}_{1_r}^T, \mathbf{v}_{2_r}^T)^T
\end{aligned} \tag{2.10}$$

Note that functions of this nature can be recursively applied so that only one D&C pass is necessary.

Matrix-vector multiplications can be performed by segmenting the matrix row-wise and keeping the vector intact. Thus:

$$\begin{aligned}
divide() : \{\mathbf{M}, \mathbf{v}\} &\rightarrow \{\mathbf{M}_{1_r}, \mathbf{M}_{2_r}, \mathbf{v}\} \ WHERE \\
& \mathbf{M} = \begin{pmatrix} \mathbf{M}_{1_r} \\ \mathbf{M}_{2_r} \end{pmatrix} \\
simple() : \mathbf{M} &\rightarrow elems(\mathbf{M}) \leq partition \\
evaluate() : \{\mathbf{M}_r, \mathbf{v}\} &\rightarrow \mathbf{u}_r \ WHERE \\
& \mathbf{u}_r = \mathbf{M}_r\mathbf{v} \\
combine() : \{\mathbf{M}_{1_r}, \mathbf{M}_{2_r}, \mathbf{v}\} &\rightarrow \{\mathbf{M}, \mathbf{v}\} \ WHERE \\
& \mathbf{M} = \begin{pmatrix} \mathbf{M}_{1_r} \\ \mathbf{M}_{2_r} \end{pmatrix}
\end{aligned} \tag{2.11}$$

The vector $\mathbf{u}$ generated by this function is in the row-wise segmented state when *evaluate*() is applied. Thus, data-parallel vector operations can be applied to this result in the same D&C pass. For example:

$$f(\mathbf{M}_r\mathbf{v})$$

Since the resultant vector is row-wise segmented it is cannot be re-used by equations 2.11. Thus the calculation of $\mathbf{o}^K$ requires two D&C passes.

2.3.5 Implementation issues

We now come to actually implementing these expressions using D&C objects. For the vertical implementation, we can, as desired, write simple algebraic expressions using appropriately defined D&C objects. An example is given in figure 2.7. As discussed previously, `operator=` actually performs the D&C operations, with the other operators simply building up the compound D&C object.

```
main()
{
        const SZ = 128;
        VectorDac<float> A(SZ), B(SZ), C(SZ), R(SZ);
        MatrixDac<float> M(SZ,SZ);

        R = A + B * exp(M*C);
}
```

Figure 2.7: A simple algebraic expression using D&C classes

The figure shows the user's view of the manipulation of matrices and vectors using delayed evaluation. As can be seen all the techniques described so far are completely hidden.

What is immediately apparent, however, is that the structure representation *is not constant.* We can characterize many operations as vectors, but at some stage, as is the case for back propagation, the representation can change; perhaps to a scalar, as is the case for vector dot product. This means that the parameterization of section 2.1.2 is not sufficiently flexible. Although we can implement much of the back-propagation algorithm in this way, a complete solution is non-trivial. Similarly, it is not clear how multiple evaluations can be integrated if their representations are different. Since the structure of the evaluator is largely influenced by the interface of the D&C objects, different interfaces will naturally yield different evaluators. This is clearly far from ideal as it leads to an excess of template instantiations and means that the evaluation is still influenced by the problem in question. What is really needed is a *complete* separation of evaluator from problem, however, it is not clear how this can be achieved whilst still allowing problems to access their underlying data structure. In chapter 3 we will see how this can be achieved.

Another problem is also apparent. If we implement as much of the algorithm as we can, and run the whole, the execution time, for one frame on one processor, is so low ($<$ 1s) that parallel implementation yields very little performance improvement, and was, in some cases, slower! This is due to the parallelizing overhead being comparable to the actual computation being performed. Thus, if we are to promote delayed evaluation as a viable technique, we must pursue problems with a great deal more algorithmic overhead. We must also design a D&C framework that incorporates sufficient flexibility for variation in the underlying data structure.

2.3.5.1 Horizontal implementation

For the horizontal implementation, similar problems are encountered. Although the underlying structure is of a vector of training frames, the result - two matrices - must be combined through addition. To implement this, whilst still retaining the vector structure, requires rather a convoluted approach involving a vector of pointers to matrices. Although this is possible, and the results obtained bear witness to this, it is again obvious that a more flexible structure is required.

```
void main()
{
    // Setup data structures.

    const TRAINING_FRAMES = 512;
    const I = 32;                                                    // nn parameters
    const J = 136;
    const K = 10;

    VectorDac< Vector<float> > inputs(TRAINING_FRAMES),
                               outputs(TRAINING_FRAMES);

    inputs = Vector<float>(I);                                       // resize the data
    outputs = Vector<float>(K);

    Matrix<float> d_Wij(I,J), Wij(I,J),
                  d_Wjk(J,K), Wjk(J,K);
    CombVectorDac< Matrix<float> > delta_Wij(d_Wij, TRAINING_FRAMES),
                                   delta_Wjk(d_Wjk, TRAINING_FRAMES);

    // Initialization for root node.
    if (getnodeid() == 1){
        inputs = outputs = Wij = Wjk = Random;                       // randomize everything
    } // in lieu of actual data

    // Divide-and-Conquer operations.

    BackpropDac<float> bp(inputs, outputs, delta_Wij, delta_Wjk, Wij,Wjk);
    bp.evaluateExpression();

    // Post-processing for root node
    if (getnodeid() == 1) {
        Wij += d_Wij;                                                // update weights
        Wjk += d_Wjk;
    }
}
```

Figure 2.8: Horizontal back propagation

The figure shows a D&C implementation of the back-propagation algorithm. The essence of the algorithm is hidden in the `BackpropDac` *object. This structure is defined to use vectors of training frames (*`inputs` *and* `outputs`*) and return results in a CombVectorDac. This vector structure sums its elements during* `combine()`*.*

The essence of the horizontal scheme is to define a D&C object, for which `evaluate()`

performs the training algorithm on a vector of training frames. `divide()` merely splits this vector up, and, as indicated previously, `combine()` sums the resultant weight update matrices. The back-propagation object can be constructed relatively easily using standard C++ classes. The actual program is shown in figure 2.8. Looking at this we can see an additional problem; the threads of control on the root processor and on the farm processors are different. These differences are coded into the program that runs on the origin node, but actually need to be transparent to the user.

Having observed some of the problems inherent to our naive approach, we move on to performance figures for the horizontal algorithm.

2.4 Results

Figure 2.9 shows the speedup gained for the horizontal scheme with 32 input units, 136 hidden units, 10 output units and 512 Frames. The experiments were conducted on a 4x4

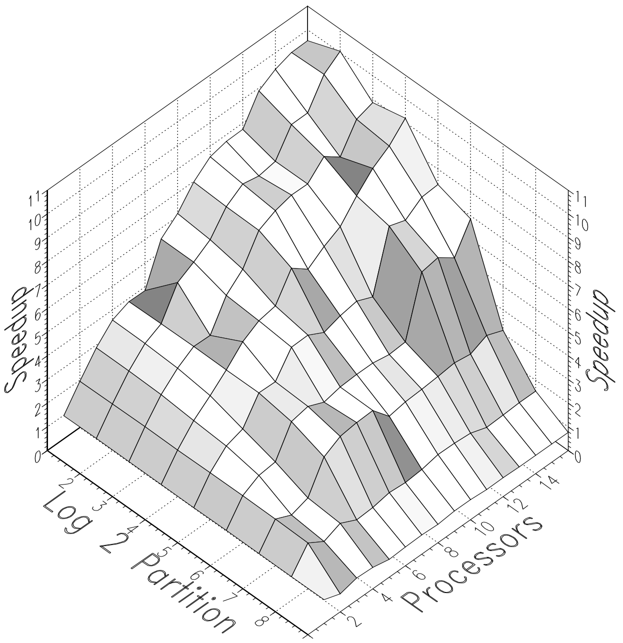


Figure 2.9: Results for Horizontal Back-Propagation

mesh of T800 transputers the structure of which is described in appendix B. The mesh was wrapped into a torus. The processor numbers used were increased rasterwise - left to right then top to bottom - the root node being at the beginning of the raster. The reference time was obtained by running the program on one processor with scheduling and partitioning switched off. The minimum divided size (partition) was varied, as well as the number of processors, to gain some perspective of how it affected speedup.

Examining the results, we see that the overall picture is fairly good with a speedup of 10 for 14 processors. A number of other features are worth noting.

- For large partitions, the number of available tasks decreases until it is less than the number of available processors. At this point performance tails off rapidly. In the limit, the speedup tends to that of the single processor case.
- When more than one processor is available, but the partition size constrains the system to use only one, the speedup gained is slightly less than the single node case. This is probably because of the overhead introduced by the bidding scheduling scheme - scheduling requests are still made to the root node, it just cannot satisfy any of them.
- Rhabi's [88] partitioning results - optimal speedups for partitions matching the sub-task size to the number of processors - are not observable here. This may be because of a difference in scheduling strategy between the two systems. The strategy employed here means that with more sub-tasks, a greater number of processors will be utilized towards the end of a processing pass. With less sub-tasks, towards the end of the computation the root processor will be the only processor left computing, when the sub-task queue has been exhausted.

2.4.1 Limitations in performance

There are two main observations we can make as factors that will inhibit performance. These two have been largely ignored in the literature, but are issues with considerable impact on performance.

2.4.1.1 Scheduling

The first factor is concerned with scheduling. In order to obtain load balance across the processor network, we employ depth-first expansion to produce sub-tasks for parallel evaluation. These tasks are pushed onto the execution stack and evaluated in parallel in FIFO order. Thus, for balanced evaluation, the first task evaluated in parallel will be 1/2 of the problem, the second 1/4. In fact the size of any task as a proportion of the whole problem will be:

$$size = \frac{1}{2^i} \tag{2.12}$$

where i is the index of the task scheduled, starting from 0. Now, because we employ the single-steal rule for scheduling, initially each of these sub-tasks will be scheduled on a different nearest neighbour. Once all the nearest neighbours have been exhausted, no further scheduling can take place until a neighbour has finished its sub-task. In the worst case, if we assume that no further scheduling *will* take place, then the execution time will be limited to that of the remaining task on the origin processor. The size of this task will be equal to the smallest scheduled task[5]. Since the number of scheduled tasks is equal to the number of links on the processor, the actual size will be:

$$size = \frac{1}{2^{n_{links}}} \tag{2.13}$$

[5] Remember we employ binary division.

and the maximum speedup will be:

$$\begin{aligned} S &= 1/\frac{1}{2^{n_{links}}} \\ &= 2^{n_{links}} \end{aligned} \tag{2.14}$$

For a transputer with 4 links, we expect the maximum speedup to be 16. For fewer links, we expect maximum speedups of 8,4 and 2. Obviously this is a worst case scenario, and the performance will be improved by the presence of non-nearest neighbours, and by nearest neighbours completing sub-tasks before the origin processor. Even so, we can observe discrepancies in our results at 4 and 8 processors. We will examine further evidence for this, and solutions, in chapter 4.

2.4.1.2 Problem size

The second observation we make is that the size of problem we can run is limited by the resources of the origin processor. Since the entire problem starts from a single source, the origin processor must have sufficient storage to be able to accommodate the whole problem. This is obviously important, firstly because we wish to be able to run problems of a reasonable size, and secondly because larger problems will tend to exhibit better performance characteristics [52].

When distributed evaluation takes place the data required for this evaluation is relatively small in comparison with the total data. Thus, as mentioned previously, some sort of virtual data transmission scheme would be necessary in order to implement larger problems, and make efficient use of the available resources. This would mean that data was only present on a node when it was actually needed and that a virtual representation was transmitted at other times.

We will study these problems in more detail in subsequent chapters.

2.5 Conclusions

We have examined a simple implementation of object-oriented D&C. We have seen that good performance can be obtained with such a system, and that we are able to manage parallel complexity effectively. We therefore conclude that the basic idea is sound, but the simplistic implementation has some fundamental flaws. If the system is to be really effective, then these flaws must be eliminated. There are also some parallel implementation issues that need to be addressed, if we are to increase the general applicability of the system.

Having examined in broad terms the concepts involved we now take a more detailed look at implementation. We have identified areas of concern and begin again with a top-down design, trying to solve these problems in a structured fashion.

Chapter 3

Design of an object-oriented D&C system

recursion 269 *The C Programming Language, p269*

We have examined the basic concepts of object-oriented D&C. We now look in more detail at the requirements for an object-oriented D&C system.

3.0.1 Chapter organisation

Section 3.1 describes the object-oriented D&C approach and its implementation together with some additional features. We also examine the approach in the light of the actor programming model. Section 3.2 describes a method of reducing a programmer's burden under object-oriented D&C. In section 3.3 we look at some enhancements to the basic system and their possible uses. In section 3.4 we take a more detailed look at the actual system implementation as well as its position in the overall software / hardware hierarchy. In section 3.5 we look at the design and implementation of some core object-oriented D&C classes, and finally in section 3.6 we look at their application to some simple problems.

3.1 Kernel Structure

In this section we describe desirable properties for an object-oriented D&C system. We also describe, in general terms, the implementation of these features in a C++ environment. Finally, we compare object-oriented D&C with the actor model of concurrent computation and describe how various features fit into this model.

3.1.1 General approach

The basis for object-oriented D&C is to incorporate the primary D&C functions into an object. This design consideration together with normal object-oriented features yields quite a powerful programming paradigm [81].

In chapter 2 we described [81] an object-oriented D&C structure that incorporated various features in a "bolted-on" fashion. In using this structure it became apparent that a more uniform, cleaner approach was needed, so that these features would then be an integral part of the design, and so that more advanced features could be incorporated. The goals of this new design are:

Full polymorphism. Our previous implementation consisted of two basic class structures. One represented the D&C functionality required, the other the data on which that functionality would work. The reason for this separation of data and functionality, which appears contrary to the goal of object-oriented programming, was three-fold. Firstly, the data-holding structures needed to be kept simple and small as the number of data objects produced during D&C evaluation was considerable. Secondly, there was no immediately apparent way of representing actions and data which were common to all D&C objects, if all actions and data were defined in a single structure. Hence, this global data was folded into the functional half of the design as this was in itself global to all D&C objects. Thirdly, parameterizing D&C objects with the data-holding structures, enabled the various D&C functions to access easily the underlying data-structure of the problem involved.

The problem with this approach is that the D&C functionality cannot be made fully polymorphic as it has to be typed by the data structure it uses – in other words this information has to be built into it. The D&C functionality requires a polymorphic interface because this interface is used by the D&C evaluation code, and we do not want to duplicate this code for every different type of D&C problem! In the end this evaluation code was parameterized upon the D&C data structure as well, using a class template facility, putting the onus on the compiler rather than the programmer.

However, this is not a particularly neat solution and runs into all sorts of problems if we want to evaluate more than one type of D&C problem at the same time. The obvious solution is to bundle the two halves into one type of object. A fully polymorphic interface can then be written and all the original problems go away.

"Virtual" object-creation. Once D&C objects are fully polymorphic they can be evaluated without any knowledge of their true type. However, for this anonymity to be transmitted coherently in a message passing environment we must have some way of transparently reconstructing these objects into their actual type.

Multiple evaluations. Provision of the features described above paves the way for concurrently evaluating multiple types of D&C objects. This would then allow programs to be written more flexibly, incorporating a greater degree of parallelism.

Mix-in-based program development. Ideally, most useful D&C functionality should be provided as a library if at all possible. This would ensure that a programmer would be able to build his program from predefined building blocks. However, each D&C object can vary in four possible ways, corresponding to variations in each of the four D&C primitives. A library containing all of the possible variations would tend to grow exponentially as more objects were added, and would eventually become unmanageable. What is needed is the ability to "mix in", or combine, single primitives together, as is allowed in Lisp Flavors, thus only requiring these single primitives, rather than their combinations, in a library [13, p54].

Delayed evaluation. We previously described [82] the advantages of delayed evaluation for creating efficient high-level algebraic operations. However, this feature was rather difficult to use in its previous incarnation, again mainly due to the structure not being totally polymorphic.

Having described desirable attributes for our object-oriented D&C system, we now discuss their implementation and an evaluation structure within which the overall scheme fits neatly.

3.1.2 Stack-based evaluation

Previous MIMD D&C implementations have generally adopted a tree-like evaluation structure. This has the advantage of maintaining the node orderings as well as being an easy structure to traverse. Another approach is to split the entire task completely and then gradually evaluate the pending sub-tasks. However, this is to perform a breadth-first evaluation which results in high memory overheads. Additionally, if division involves some computational overhead – and in many problems division represents the *only* computational overhead – then efficient parallelism is not possible. This approach can also suffer from incorrect task ordering.

D&C lends itself naturally to recursive evaluation. However, parallel evaluation requires that we have access to tasks that have not yet been evaluated, and recursive evaluation does not allow us to do this easily.

Sedgewick [96, p45], gives an algorithm for traversing a binary-tree using a stack rather than by a recursive method. By modifying this algorithm it proves possible to evaluate a D&C task, depth-first, using a stack. We can only do this by virtue of the fact that we have objects representing D&C problems. Thus at any stage of evaluation a D&C object holds *all* information necessary for further division. Once an object has been created it has no interdependency with its parent, and can be manipulated totally freely as a complete entity.

Two issues must be considered. Sedgewick's algorithm gives a way to traverse a tree in place[1], but of course the D&C algorithm necessitates *building* a tree and *ascending* it as well. The former is important to consider since as the tree is not in-place, the ordering of the nodes cannot be fixed rigidly. The latter is important because the original algorithm discards the tree information after it has visited nodes – so the information we require for division is present but that for combination is lost.

The combination problem can be solved by introducing a second stack onto which nodes are placed after they have passed through the divide stage of evaluation. If we do this correctly then we can pop successive pairs of nodes from this stack for combination (figure 3.1). Unfortunately, this works fine except for a few pathological cases which cause incorrect node ordering or evaluation. A trivial example is shown in 3.1.

If we have an unbalanced tree, where a node has less than two children, then the problem is more serious. If we modify the algorithm to cope with the case where a node only has a left child, then the right node only case fails, and vice versa.

The way we solve this problem is by introducing a *key* to the stack which we use to encode the *level* at which a particular object is in the tree. The root object is at level 0, its two children are at level 1 and so on. By doing this we can make sure that only pairs with the same level are combined. The modified algorithm is given in the program in figure 3.2.

3.1.2.1 Advantages of a stack

Stack-based evaluation of D&C problems provides more than just a simple, elegant algorithm. Most MIMD D&C implementations utilize a stack for holding pending tasks [72, 81]

[1] In other words a tree that already exists.

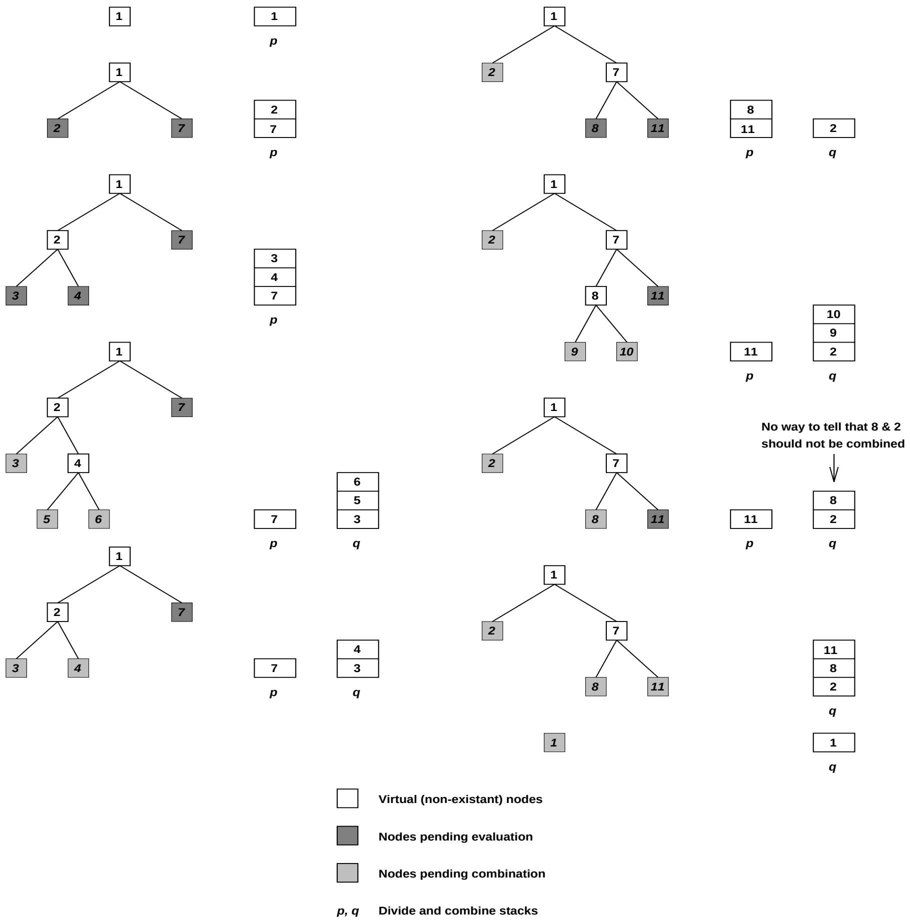


Figure 3.1: Stack-based evaluation

The diagram shows the depth-first, D&C expansion of the task labelled 1. *Execution proceeds by evaluating first left-nodes and then right-nodes. The evaluation is performed using a stack, and the diagram gives an example of where the evaluation is ambiguous. Normally, pairs of evaluated objects are combined from stack* q*, however, in the example objects* 8 *and* 2 *are a pair but should not be combined.*

because the tasks can be taken from the *bottom* of the stack for parallel evaluation. Since our whole evaluation scheme is stack based, tasks eligible for parallel evaluation are readily available. Further advantages we will consider in section 4.3.2.

3.1.3 Virtual object construction

Passing anonymous objects around a message-passing environment necessitates some means of reconstructing objects based on a message. Coplien [31, p290ff] describes a scheme for implementing exemplar-style programming in C++. The techniques he uses allow for arbitrary classes to register in an exemplar list. When an object is to be created, its construction parameters are presented to each registered exemplar in turn to see whether it can consume the input and return an object. By modifying this operation slightly we can

```
Dac Dac::run() const {
    Dac l,r,n;
    int s;                                                  // current level
/*
 * When the root node is simple then we simply evaluate it.
 */
    if (simple()) return evaluate();
/*
 * we now save the stack positions in case we recursively call this function.
 */
    int ppos = *p, qpos = *q;

    p→push(*this, DacStack::limbo, 1);                      // musn't spawn the first one

    while (!p→empty(ppos)) {                                // emptiness is relative to ppos
/*
 * we'll start to divide until we have something to combine.
 */
        while (!q→pair(qpos)) {

            s = p→key();                                    // get the level
            n = p→pop();                                    // get the object
/*
 * note that n might have suddenly become simple under our noses ...
 */
            if (!n.simple() && !(n = n.divide(l,r))) {
/*
 * If the two child nodes are simple then we evaluate them and push
 * them onto the combine stack. Otherwise we push both nodes onto the
 * evaluation stack. Note that we are going down a level.
 */
                if (!l.simple() || !r.simple()) {
                    p→push(r, DacStack::pending, s+1);
                    p→push(l, DacStack::limbo, s+1);
                }
                else {
                    q→push(l.evaluate(), s+1);
                    q→push(r.evaluate(), s+1);
                }
/*
 * if we couldn't divide then we evaluate the current node and push it
 * for combining. This should cope with the case when the current node
 * is simple.
 */
            } else {
                q→push(n.evaluate(), s);                    // level is constant
            }
        }
/*
 * combine objects if we have some.
 */
        while (q→pair(qpos)) {
            s = q→key();                                    // we are going up a level
            r = q→pop();
            l = q→pop();
            q→push(l.combine(r), s-1);
        }
    }
/*
 * The last object left is the one we want ...
 */
    return q→pop();
}
```

Figure 3.2: The stack evaluation algorithm

assign each D&C class a type code with which registered D&C exemplars can determine whether an incoming object is of its type and act accordingly.

The advantage of Coplien's approach is that the number of exemplars need not be known by the management code and new classes can be added to the exemplar list, simply by linking in the appropriate library, no code modification is necessary.

3.1.3.1 Nstreams

Coplien's exemplar example consumed characters from a static buffer. Although it would be possible to do this in a message-passing environment, this sort of input / output operation is far more simply performed using C++ streams (see section 1.5.1.3). These structures dynamically request input from a stream source in a manner transparent to the user [95] and are designed in such a way that the stream source can be modified relatively easily. Thus we have designed message-passing iostreams called nstreams based on the Trollius^{TM} message passing calls `nsend()` / `nrecv()`. These structures transparently send and receive network packets when sinking and sourcing information as per standard iostreams. One of the advantages of using this scheme is that short messages are buffered and transmitted as long messages. This reduces excessive numbers of communication messages.

Unfortunately, iostreams are designed to transmit information in ascii format, which is fine for writing data files to disk or scanning for user input, but leads to data expansion during message-passing! In order to overcome this, we create binary iostreams which are constructed from normal iostreams but which force binary reads and writes to be performed, thus optimizing the volume of data transmitted.

In addition to this optimization we want to make sure that long messages are transmitted directly rather than being intermediately buffered. This is not a problem except for the fact that these unbuffered messages could potentially overtake the buffered messages. For this reason we make sure that the unbuffered messages are only transmitted when the stream is flushed (implicitly or explicitly) at which point we can ensure that the buffered information is sent first.

3.1.4 Interface design

In this section we build up an object-oriented D&C interface, gradually incorporating desirable features.

3.1.4.1 The basic interface

To start with we require the D&C primary functions. In order for the interface to be fully polymorphic, all function arguments and return values must be expressed in terms of a generic D&C object `Dac`, see figure 3.3.

3.1.4.2 Reference counting and labelling

A little thought shows that this immediately creates a problem. All polymorphic behaviour in C++ is exhibited through operations on *references* or *pointers* to objects. If we are to return from functions, objects which exhibit polymorphic behaviour, then it is references or pointers that must be returned. Unfortunately, this creates a memory-management nightmare as all objects referenced in this manner would have to have global

```
class Dac {
  public:
    virtual const boolean simple() const;
    virtual Dac divide(Dac& l, Dac& r) const;
    virtual Dac evaluate() const;
    virtual Dac combine(const Dac& d) const;
};
```

Figure 3.3: Basic D&C interface

The figure shows the basic object interface for the new D&C design. Note that there is now no parameterization of the interface. All arguments and return values are expressed in terms of the generic object `Dac`.

lifetime. Fortunately there is a way around this problem; by using Coplien's [31, p133] envelope and letter idiom we can make the interface simply a label for a real object that can be changed readily without affecting the label. So in this instance our label is "D&C object" but what the real object might be is not pre-determined by the interface. The other advantage of this idiom is that we can incorporate reference counting for the actual objects so that memory-management is no longer a problem and object assignment and copying are cheap. Procedure calls made to the label can be forwarded to the actual object. Additionally, if this forwarding is done in-line then it has no performance penalty.

Coplien's original design had class member functions delegating their operation through a pointer to an object of the same class. However, we require that all actual D&C objects have some global information – like size – and we do not want to burden the "labels" with this information any more than necessary. For this reason we delegate the D&C interface's functions through a pointer to an object of a class *derived* from the interface class. See figure 3.4.

3.1.4.3 Mix-in support

We now wish to incorporate support for mix-ins. Mix-ins combine class functionality through the use of multiple-inheritance and a common base class. For example a base class X might declare the functions $a()$ and $b()$, and two other classes A and B, derived from X, might define one of these functions each. If we were then to require a class C that required the functionality of $a()$ or $b()$ or both, we could then derive C from X, to make it have the interface of X, and additionally make C "mix in", through inheritance, A or B depending on the functionality required.

In order for mix-ins to work, the base class defining the interface must be a virtual ancestor of all its derived classes. This means that all descendants of the base class share a single instance. The current D&C design makes this constraint easy to satisfy as we just make the envelope (`Dac`) a virtual ancestor of the letter (`DacRep`). Any mix-in definitions must then be virtually derived from `Dac`.

This raises one other issue. Using mix-ins can be made safe by making the base class functions all pure virtual. This means that the base class does not define these functions, and the compiler forces the programmer to define these functions in derived classes. Unfortunately, our self-referential design means that the D&C base class functions are already defined to delegate their operation to the derived letter. Thus if a programmer

```
class Dac {
    friend class DacRep;

  public:
    virtual const boolean simple() const {
        return d_rep→simple();
    }
    virtual Dac divide(Dac& l, Dac& r) const {
        return d_rep→divide(l,r);
    }
    virtual Dac evaluate() const {
        return d_rep→evaluate();
    }
    virtual Dac combine(const Dac& d) const {
        return d_rep→combine(d);
    }
        // reference counting
    Dac() {}
    Dac(const Dac& d) : d_rep(d.d_rep) {
        if (d_rep) d_rep→count++;
    }
    const Dac& operator= (const Dac& d) {
        if (d_rep ≠ d.d_rep) {
            if (d.d_rep) d.d_rep→count++;
            if (d_rep && --d_rep→count ≤ 0) delete d_rep;
            d_rep = d.d_rep;
        }
        return *this;
    }
    ~Dac() {
        if (d_rep && --d_rep→count ≤ 0) {
            delete d_rep;
        }
        d_rep = d;
    }

  protected:
    DacRep *d_rep;
};

class DacRep : public Dac {
    friend class Dac;

  public:
    DacRep() : count(1) {}
    unsigned char count;
};
```

Figure 3.4: Reference counting

The figure shows how the basic interface object, `Dac`*, can be reference counted to eliminate unnecessary copying. Note that all data is actually held in* `DacRep` *and all primary functions in* `Dac` *are forwarded to this object.*

```
class Dac {
    friend class DacRep;

  public:
    virtual const boolean simple() const {
        return d_rep->simple();
    }
    virtual Dac divide(Dac& l, Dac& r) const {
        return d_rep->divide(l,r);
    }
    virtual Dac evaluate() const {
        return d_rep->evaluate();
    }
    virtual Dac combine(const Dac& d) const {
        return d_rep->combine(d);
    }
        // reference counting
    Dac() {}
    Dac(const Dac& d);
    const Dac& operator= (const Dac& d);
    ~Dac();

  protected:
    DacRep *d_rep;
};

class DacPure : public Dac {
  public:
    const boolean simple() const=0;
    Dac divide(Dac& l, Dac& r) const=0;
    Dac evaluate() const=0;
    Dac combine(const Dac& d) const=0;
    DacPure() {}
};

class DacRep : virtual public DacPure {
    friend class Dac;

  public:
    DacRep() : count(1) {}
    unsigned char count;
};
```

Figure 3.5: Mix-in support

The figure shows an abstract interface inserted between the `Dac` *and* `DacRep` *classes. All mix-in classes inherit from this interface so that inheritance will yield a correctly constructed D&C class. The interface is abstract (functions ending in* `=0`*) so that derived classes are forced to define these functions.*

happens to forget to define one of these functions – and the compiler will allow this – any call will default to the base class definition. This obviously gives rise to a never-ending recursion that will only terminate when the process' stack space is exhausted. We can overcome this problem by introducing an intermediate "pure" interface that we place between the envelope and letter in the inheritance hierarchy. All mix-in definitions can then be derived from this pure interface and the compiler will then complain unless all functions are properly defined. See figure 3.5.

3.1.4.4 Exemplar support

Finally we must add exemplar support so that objects can be transmitted around a message passing environment. The functionality required by the interface is given in Coplien and comprises three functions: one to find a type match between a list of registered exemplars and an input stream which subsequently calls the relevant object constructor, another – simply a virtual placeholder – to output a type identifier and associated object to a stream and a third to register an exemplar in the exemplar list.

```
class ADacClass : public DacRep {
  protected:
    Obj anotherMember
    int aMember;
        // construction from an input stream
    ADacClass(ibstream& i) : DacRep() , anotherMember(i) {
        i >> aMember;
    }
        // exemplar registration constructor
    ADacClass(ExemplarConstructor e) : DacRep(e) {}
        // "virtual" construction from in input stream
    virtual DacRep* scan(ibstream& i) {
        return new ADacClass(i);
    }
        // output to a stream
    virtual void spawn(obstream& o) {
        o << anotherMember << aMember;
    }
        // registered exemplar
    static ADacClass* exemplar;
};

ADacClass* ADacClass::exemplar = ::new ADacClass(exemplarConstructor);
```

Figure 3.6: Exemplar support

The figure shows a hypothetical D&C class with exemplar support. Functions are defined to allow objects of this class be transmitted. The key to the exemplar support is the `exemplar` *member. This object holds type information (mainly a type identifier) for the class. This enables objects of the class can be recognized and constructed without explicit reference to the class.*

The exact mechanism of these three is not important here, however their effect upon derived classes is. A D&C class with input / output ability is characterized by three associated functions. One, a virtual function which returns an object of its type constructed from an input stream, another the associated input stream based constructor, and a third an output function for the object. In addition to these each class must have a static member of its own class which serves to register the class in the exemplar list. See figure 3.6.

Unfortunately, as can be seen, although flexible, this approach to object transmission is rather tedious from a programmer's point of view. We will examine the relief of this problem in section 3.2.

3.1.4.5 Dynamic type narrowing using exemplar support

One of the advantages of our simplistic implementation was the fact that D&C functions had direct access to the underlying data structure of the problem. This was made possible by the fact that each D&C object was parameterized by a representation of this structure. With D&C objects now being totally polymorphic we lose this type information. However, D&C functions still need to be able to access the actual type of the object involved. Fortunately, D&C functions will generally know the type of the object that they *need*. They just need to be able to convert the object that they have to this type[2].

Unfortunately, simple casting is not sufficient since objects may involve multiple inheritance lattices with virtual base classes. However, the exemplar structure we have incorporated in section 3.1.4.4 allows us to implement a version of the dynamic casting scheme given in Stroustrup [102]. This can be done by including a dynamically bound function `get_this_ptr()` in every class with an exemplar. Then a parameterized class `ptr_cast<Type>` can be used to cast a D&C object to type `Type`. `get_this_ptr()` compares the type of its class with that required by `ptr_cast<Type>`. If the comparison matches then a pointer is returned, otherwise the call is delegated to any parent classes. If no match is found then 0 is returned. This is similar to the run-time type information scheme adopted by the X3J16 ANSI committee [103]. See figure 3.7. This functionality can be extended as we will examine in section 4.2.3.1.

3.1.5 Object oriented D&C and the actor model

This section relates object-oriented D&C to the actor model and shows how delayed evaluation becomes a natural extension to object-oriented D&C in this framework.

The actor object-oriented programming model [2] represents programs as an interacting set of computational agents which map incoming communications to 3-tuples consisting of:

1. a finite set of communications sent to other actors;
2. a new behaviour; and,
3. a finite set of new actors created.

Now that the D&C primary functions (figure 3.3) are part of a D&C object rather than separate from it, we can see that they constrain D&C objects to a form of these rules. The rules are limited by the requirements of the D&C algorithm[3], namely that incoming

[2] Or signal an error if this is not possible.

[3] This is really what we would expect, the actor model being so powerful.

```
class DacRep : virtual public DacPure {
    ...
    virtual void* get_this_ptr(int) const;
    ...
};

template <class T> class ptr_cast {
    const DacRep* p_d;
  public:
    ptr_cast(const Dac& d) : p_d(d.rep()) {}
    ptr_cast(const DacRep* d) : p_d(d) {}
    operator T*()
    { return (T*)(p_d->get_this_ptr(int(T::exemplar->type))); }
    T* operator-> ()
    { return (T*)(p_d->get_this_ptr(int(T::exemplar->type))); }
};

class zClass : public zBase
{
  protected:
    void* get_this_ptr(int i) const {
        void *v=0;
        if (i == int(exemplar->type)) {
            v = (void*)this;
        }
        else {
            v = zBase::get_this_ptr(i);
        }
        return v;
    }

  private:
    static DacRep* exemplar;
};

example_usage(const Dac& d)
{
    zClass* = ptr_cast<zClass>(d);
}
```

Figure 3.7: Dynamic casting of D&C objects

The figure shows how generic D&C objects can be cast to their actual type using exemplar support. The class `ptr_cast<T>` *accesses the exemplar for its formal parameter* `T`*, and searches the desired object for a type match using* `get_this_ptr()`*. If a type match is found then a cast is performed, otherwise 0 is returned.*

communications are mapped to a finite set of new D&C objects created where the set is one of:

- a fixed number N created through division for N-ary D&C;
- a single object created through evaluation;
- a single object created through amortization of other objects, where the incoming communication contains one or more other objects.

Note that the second is a call-by-value form of (2) above. Although a D&C object could conceivably change its type internally – since its interface is purely a "label" – it is intentionally made difficult by the `const`-ness of the primary functions.

We will refer to the object that creates another, as the parent; and to the created object as its child.

One might think that not changing type would lead to a proliferation of D&C objects; and for the recursively evaluated and tree evaluated (section 2.1.3) cases this is so. However, with the stack based algorithm, the number of D&C objects is kept to a minimum by virtue of the fact that there are no used D&C objects serving as placeholders. Thus the rules above are all extended such that:

- no object changes its behaviour; and,
- an object's existence is terminated upon acceptance of a communication

Of course an object could return a copy of itself before termination. If the parent object is to be terminated then the copy could actually be the parent object itself. Since our object scheme is reference counted, the "new" child object can have a new label but the same interior. Thus the second item, above, can be adhered to conceptually *and* physically, without unduly increasing computation time.

The only disadvantage that can be envisaged is that of not being able to easily reuse object memory allocations. This problem can be overcome by using reference counting for all classes to be used by D&C objects. The advantages are many, not least the simple programming perspective that is realized.

In fact we could rewrite the rules so that changing an object's behaviour became integral to the operation; so that division, for example, would involve an object changing its state and creating $N-1$ new objects. However, this removes the uniformity of the approach by making a distinction between types of child object.

3.1.5.1 A homogeneous approach

In our previous implementation D&C objects had to be clumsily represented in terms of evaluation objects and result objects. Now the types of object are homogeneous: all are D&C objects but the type of D&C object can be transformed when a change of functionality is required. This eliminates the need for functional baggage that would clutter the definition of D&C objects. It also means that changing form can be accommodated easily.

For example if we have some operation that involves a matrix changing into a scalar, we can separate the two distinct types by defining a D&C object that deals with matrices, a D&C object that deals with scalars and a mapping between the two. If we then want to define an operation that involves only matrices we are not denied the possibility of using the matrix type object. Previously we would have had to define a single D&C object that knew about matrices and scalars and defined operations on both – and the object would be specific to that single operation. There are, however, some implementation difficulties that complicate this. We will look at these in more detail in chapter 5.

3.1.5.2 Delayed evaluation

In viewing object-oriented D&C as a specialized actor system we have made no reference to mapping (1) above for general actors. Allowing D&C objects to be replicated, amortized or transformed purely fits within the confines of creating a finite set of new actors (3). However, delayed D&C evaluation features, described in [81], operate by combining objects of *differing* types prior to D&C evaluation, and subsequently performing this evaluation upon the aggregate object using delegation [23]. This possibility fits neatly into mapping (1), as an aggregate D&C object would first create a finite set of children and then pass on the communication to its constituent members.

The application for this, discussed previously, is in evaluating arithmetic expressions where, for example, we might wish to evaluate the matrix expression $A+B+C$. Delaying the evaluation of this expression means producing an aggregate – in our case D&C – object, and delegating calls to the aggregate's interface to the individual object's functions. See figure 2.5.

3.1.5.3 Envelope / letter advantages for delayed evaluation

In using delayed evaluation we have two objectives:

- constructing the aggregate; and,
- evaluating the aggregate.

In the example given in section 2.2.3 the aggregate is constructed using overloaded arithmetic operators. However, previously there was no clear way in which to organize an object hierarchy that allowed the interactions of evaluation and construction to be separated. For example in the expression $A = B + C * D$ where $A - D$ are general objects. The evaluation sequence would have been:

$$\begin{aligned}
A &= B + C * D \\
A &= B + TimesObj[C, D] \\
A &= AddObj[B, TimesObj[C, D]]
\end{aligned}$$

where $O[P_1, \ldots, P_n]$ denotes an object O containing objects $P_1, \ldots, P_n$. Whereupon the compound $AddObj$ is evaluated and assigned to A. However, in order for this sequence of events to operate correctly, $AddObj$ and $TimesObj$ must be derived from some generic $ArithObj$ which defines the addition and multiplication operators; otherwise $TimesObj$ and $AddObj$ would need to define every possible operator combination. During subsequent D&C evaluation these functions are completely redundant. In addition, defining any new operators would necessitate recompilation of all the arithmetic classes.

With the envelope / letter idiom all of these problems disappear. The aggregate constructors can be defined within a wrapper derived from the envelope class `Dac`, while the aggregates themselves can be defined normally – derived from the letter class `DacRep`. The wrappers then serve purely to build an aggregate object, old wrappers being discarded when they are redundant:

$$\begin{aligned} ArithWrap[A] &= ArithWrap[B] + ArithWrap[C] * Arithwrap[D] \\ ArithWrap[A] &= ArithWrap[B] + ArithWrap[TimesObj[C, D]] \\ ArithWrap[A] &= ArithWrap[AddObj[B, [TimesObj[C, D]]]] \end{aligned}$$

This also means that the assignment operator can be completely generic and included in the definition of the wrapper. In fact the advantages of keeping the wrapper for each object are not insignificant. By keeping the wrappers we can take advantage of the reference counted relationship between wrappers and their letters. We will examine the actual definition of these types of object in subsequent chapters. We will also examine the implementation of delayed evaluation, in far more detail, in chapter 5.

Of course these features are a subset of the possibilities under the actor regime, and thus delayed evaluation represents a natural usage of object-oriented D&C rather than an added feature.

3.2 Translation versus code insertion

In this section we briefly describe an approach to overcome some of the tediousness of programming with D&C objects.

Many parallel object-oriented systems [44], with some exceptions [11], use a language translator to parse their particular flavour of the target language. For C++ at least, this is not particularly desirable, besides being difficult, as one has to have the translator as well as the target language compiler *and* ensure that the two are compatible. In addition the programmer has to learn the new constructs, and – as shown by the C++ standardization effort – these constructs may well just be unnecessary syntactic sugar.

Such is true of our object-oriented D&C additions to C++: we can incorporate these constructs using existing language features, so that they become meta-features. We could augment the C++ language definition in order to make the programming of these constructs easier, but with the portability problems given above. In addition it is not always clear what a programmer's intentions for a D&C object are, and enforcing a translator-based regime could yield an undesirable degree of inflexibility. Also, with C++, new language features are being accepted by the ANSI committee, features which will make the implementation of our exemplar system, for example, far less tedious. We describe some of these in appendix A.

However, making C++ objects into D&C objects *is* slightly cumbersome and some sort of automation would be desirable. If we are not going to write a translator then the obvious solution is to automatically insert the required code directly into the source – and we can achieve this by using the GNU emacs editor.

3.2.1 Dac-mode

GNU emacs differs from most editors in that it is almost infinitely reconfigurable through the use of its internal lisp interpreter. It also has a powerful regular expression library. These two features, used in conjunction, mean that quite complicated language constructs can be parsed by emacs. It is even possible to execute lisp commands non-interactively so that emacs can be used as a pseudo-translator if so desired.

Thus we have written an emacs "mode" – `dac-mode` – that allows a user to modify a C++ file by inserting text relevant to D&C evaluation. The lisp code parses C++ class definitions and determines what functionality is required to turn the class into a properly behaved D&C class. Different types of modification are available, depending on whether the resultant class is to be used as a mix-in, concrete class or in a library. Since the inserted text is editable, any wrong assumptions made by the lisp code can be corrected by the programmer. The lisp code will not attempt to update D&C-relevant constructs that are already in existence.

By providing this mode, the programmer is freed to consider only those functions that require actual design. All other programming is done automatically. Thus one might define a class consisting only of base classes, a constructor and `divide()`. `dac-mode` then provides the exemplar code, the type narrowing code, the object I/O code etc. One of the advantages of this system is that `dac-mode` can be made to provide functionality that needs to be visible, as well as functions related to behind-the-scenes working. For instance it can be made to provide constructors that are directly useful to the programmer, or prototypes for D&C primary functions. This sort of functionality is squarely in the programmer's domain, since it requires either additional information to be complete, or needs to be visible to the programmer. As such it would be foolish to try and do this with a translator, however, `dac-mode`, as a programmer's *tool*, is ideal.

Additionally, `dac-mode` could be extended to supply novice users with interactive design functionality. For instance, pull down menus of available parent classes, interactive prompts for relevant information. Emacs 19 and its derivatives (Lucid Emacs) provide the means to create what is essentially an integrated design system, yet at a fraction of the coding cost of designing such a system from scratch. Most D&C classes described in later sections have been defined using `dac-mode`, and it has proved to be an invaluable tool.

3.2.2 Conclusions

This whole environment manages to maintain programming flexibility whilst removing repetitious work. See appendix C for a description of `dac-mode` functionality.

3.3 Constructing D&C classes

In this section we describe the D&C interface in more detail and demonstrate additional features that make this interface more flexible and powerful. We also describe how some of these features fit into the ethos of designing D&C classes.

3.3.1 The D&C interface

We have described the design of the D&C interface and basic hierarchy. In actual use this design has proved robust and flexible, however, experience has shown that some additional refinements are desirable. We now look at some of these.

Conversion and ensuring type correctness. All concrete D&C objects are derived from `DacRep`. However, these classes must be *manipulated* through `Dac` wrappers. Since `DacRep` is derived from `Dac` it is currently possible for a concrete D&C object to be used where a wrapper is actually needed. If this is allowed to happen then anti-social program degradation will result. To prevent this we can *force* D&C objects to be automatically 'wrapped up' before being used. This is achieved by making the relationship between `Dac` and `DacRep` a *private* one. This makes all of `Dac`'s functions unavailable to derived classes. We then enable conversions from `DacRep`-type classes to `Dac`s by providing a constructor in `Dac`. C++ conversion rules [36, p270] make this conversion implicit.

```
class Dac {
    friend class DacRep;
  public:
    Dac(const DacRep& d) : d_rep((DacRep*)&r) {
        if (d_rep) d_rep→count++;
    }
    Dac(DacRep* r, BaseConstructor) : d_rep(r) {
        if (d_rep) d_rep→count++;
    }
    Dac(DacRep* r) : d_rep(r) { }
    ...

  protected:
    struct BaseConstructor { BaseConstructor() {} };
    DacRep *d_rep;
};

class DacPure : private Dac {
    DacPure() {}
    virtual Dac wrap() const=0;
};

class DacRep : virtual public DacPure {
    friend class Dac;
  protected:
    Dac wrap() const {
        return Dac((DacRep*)this, BaseConstructor());
    }
    ...
};
```

Figure 3.8: Implicit conversion of `DacRep` to `Dac`

The figure shows how `DacRep` *objects can be forced to be* `Dac` *objects when used. The* `private` *relationship between* `Dac` *and* `DacPure` *ensures that a* `DacRep` *object cannot simply be cast to a* `Dac`*. Instead the constructor in* `Dac` *must be invoked, correctly encapsulating the* `DacRep` *object inside a* `Dac`*.*

The same can be done for pointers to `DacRep`s. Since most `DacRep*`s will be created through the use of `new`, we make the constructor by default *not* increment the reference count. However, we also provide a member function `wrap()` that returns its parent object inside a wrapper. This also requires a `DacRep*` constructor, but this time the reference count must be incremented. We differentiate between the two constructors by adding a dummy argument `BaseConstructor` to the constructor. This class is local to `Dac` and so does not pollute the global namespace. This technique is described in Coplien [31]. See figure 3.8.

Comparison of D&C objects. We wish to be able to test D&C objects for equality, and whether letters exist or not (for example in `run()`). To this end we define the operators '==' and '!=' to compare the *letters* of wrappers. Thus we can write `if (Foo == 0)` where `Foo` is a D&C wrapper. Ideally we would define a conversion to `void*`, but this prevents the possibility of defining other, more useful, operators further down the inheritance hierarchy.

Cloning objects. Often we wish to define D&C classes that provide functionality for producing additional objects of their own type. Unfortunately, if the object being used is of a type derived from this functional class, then the functional class cannot know what type of object to create.

What is really required is for the functional class to *clone* itself, leaving it to the derived class to sort out what actual type of object to create. The created object can then be cast, using `ptr_cast<Type>`, to the type of the functional class, so that the functional class can make whatever modifications are necessary. Fortunately this is quite easy with our polymorphic system. We simply define a dynamically bound function `make()` that returns a D&C wrapper. Then all concrete classes can define this function to create an object of their own type. This usage is so common as to be a candidate for inclusion in `dac-mode`. In fact `make()` is crucial to the definition of generic D&C operation classes, for example the compound classes of section 5.3.3. See figure 3.9.

For example, say we wanted to create a class that defined `divide()` for `zVector<T>`s. We would use `make()` to create the two resultant objects. We would then use `ptr_cast<zVector<T>>` to allow us to initialize these objects with the correct vector information. Simply creating a `zVector<T>` using the normal constructor method is not sufficient, since this will lose the type information of the current, possibly derived, object, and subsequent operations will fail.

Printing. As we shall see in chapter 4, one of the strengths of object-oriented D&C lies in the creation of compound D&C objects. This has already been illustrated in the use of delayed evaluation. Such D&C collections are rather difficult to debug in terms of seeing the relationship between the different objects. Thus we provide printing facilities for D&C objects. Additionally, we keep track of the position of the current object within the compound. By displaying a proportional amount of whitespace before each object, we can view a compound's structure with ease. Printing also enables the actual type of polymorphic objects to displayed. Thus, a debugger might only reveal that we are operating on a D&C object, but printing will reveal the actual structure of that object.

Node numbering. Often it is desirable to have a unique identifier for each object in the D&C evaluation tree, for example the tree dumping of section 4.5.1.1. This can be achieved by labelling nodes in *level order*. Since this requirement is necessary in more than one situation, we provide it as a basic feature of D&C wrappers. See figure 3.10.

Object counting. For debugging debugging purposes we wish to keep track of the D&C objects currently in existence. This enables us to make sure that there are no

```
class Dac {
  public:
    Dac(DacRep*);
    Dac make() const {
        return d_rep->make();
    }
    ...

  protected:
    DacRep *d_rep;
};

class DacPure : private Dac {
  protected:
    virtual Dac make() const {
        ::error() << "make() not defined in most derived class" << fatal;
        return Dac(0);
    }
    ...
};

class DacRep : virtual public DacPure {
    ...
};

class AClass : public DacRep {
  protected:
    AClass();
    Dac make() const {
        return Dac( new AClass() );
    }
    ...
};
```

Figure 3.9: Cloning support for D&C objects

The figure shows how an object of type `AClass` *can be constructed polymorphically using* `make()`*. The usage of* `make()` *does not require any prior knowledge about the type of the object being constructed.* `make()` *can simply be applied to a generic* `Dac` *object.*

unreferenced objects at the end of execution. We do this by adding each *letter* to a linked list when each is created. Upon deletion, an object is removed from the list. At the end of execution, we can check the list to see if any objects remain in it. Note that, due to the conversion enforcement described above, the presence of unreferenced objects is extremely unlikely. Letter type objects are almost always converted to their reference counted counterparts.

Global data. In certain situations we wish to have objects that are shared across all objects of a particular class. Such objects are declared as class static. However, if these objects are to have access to run-time information, such as command-line arguments; or are to make use of D&C utilities, such as memory management; or are

```
class Dac {
  public:
    Dac divide(Dac& l, Dac& r) const {
        Dac x = d_rep→divide(l,r);
        l.d_rep→d_node = (d_rep→d_node ≪ 1);
        r.d_rep→d_node = (d_rep→d_node ≪ 1) + 1;
        return x;
    }
    int node() const {
        return d_rep→d_node;
    }
    ...

  protected:
    DacRep *d_rep;
};

class DacPure : private Dac {
  protected:
    virtual int node() const=0;
    ...
};

class DacRep : virtual public DacPure {
  protected:
    int node() const {
        return d_node;
    }
    DacRep() : d_node(1) { }

    int d_node;
    ...
};
```

Figure 3.10: Level-order numbering of D&C nodes

The figure shows how each D&C object in an evaluation can be given a unique identifier. Level-order numbering means that the node identifiers are consistent no matter what type of evaluation is employed.

to be destructed gracefully, then they cannot be *created* at global scope[4]. Instead, we again use the exemplar framework.

If we define dynamically bound functions `sti()`, for initialization, and `std()`, for de-initialization, then these functions can be called for each *exemplar*, after `main()`, but before the D&C program actually begins. See figure 3.11. This of course entails using a different main-line name, `mainCore()`, for the actual D&C program, but the flexibility gained outweighs this minor inconvenience.

Size determination. In evaluating D&C objects the actual size of these objects, real or imagined is very important. Only by having a size metric can the evaluator know when objects are simple. To this end we provide a data member for `DacRep`s which defines the size of the object. Originally, it was tried to make this member a reference

[4]This is because the initialization order of these features is undefined at global scope.

```
class Dac {
  protected:
    static DacRep *list;
    union {
        DacRep *d_rep;
        DacRep *next;
    };
};

class DacRep : virtual public DacPure {
  public:
    virtual void sti();                              // global initialisation
    virtual void std();                                   // global cleanup
    static struct ExemplarManager {
        ExemplarManager() {}
        ~ExemplarManager();
      public:
        static void sti();               // initialise all registered exemplars
    } exemplarManager;
};

void DacRep::ExemplarManager::sti()
{
    for (DacRep *n = DacRep::list; n; n = n→next) n→sti();
}

DacRep::ExemplarManager::~ExemplarManager() {
    for (DacRep *n = DacRep::list; n; n = n→next) n→std();
}
```

Figure 3.11: Static initialization for D&C objects

The figure shows how globally scoped objects can be initialized and still have access to command-line arguments. Any D&C object requiring this facility must define a `sti()` / `std()` *pair.* `sti()` *functions are then called by main.*

to some other parameter, however exceptions broke the rule. Instead a dynamically bound function `size()` is provided which can be redefined by derived classes. By default this function returns `DacRep`'s size parameter. To maintain encapsulation an overloaded function `size()` is provided which accepts an argument for modifying the size data member.

Each of these additions to the D&C interface is actually defined in `DacPure`. This means that derived classes are not forced to define the functions. However, the definitions in `DacPure` are such that an error is signalled if a function is actually called, but no derived definition exists.

These additions make easier the implementation of various problems. However, it is not an exhaustive list; we will describe further enhancements in later chapters.

3.3.2 Other types of evaluation

Thus far we have described enhancements to the D&C interface, however, we can manipulate this interface in other ways than simply depth-first expansion. We will now look at some of these.

3.3.2.1 Recursion

```
Dac Dac::recurse() const {
    Dac l, r, n, ret;

    if (!simple()) {
        if ((n = divide(l,r)) ≠ 0) {
            ret = n.evaluate();
        } else {
            l = l.recurse();
            r = r.recurse();
            ret = l.combine(r);
        }
    } else {
        ret = evaluate();
    }

    return ret;
}
```

Figure 3.12: Recursive D&C evaluation

The figure shows how D&C objects can be evaluated using standard recursion. Objects evaluated in this way cannot be run in parallel.

Depth-first expansion using a stack is simply a way of defining an essentially recursive algorithm in terms that can be parallelized. In some situations, for debugging purposes or where parallel evaluation is not desirable, a simple recursive implementation of the algorithm is preferable. To this end we provide `recurse()` as part of the D&C functionality. See figure 3.12.

3.3.2.2 Breadth-first evaluation

Depth-first evaluation has many desirable characteristics for a parallel D&C system.

Consider the evaluation of a very large problem, one that is larger than will fit on a compute node. In this instance it seems reasonable to allow the host computer, with its large resources, to start dividing the problem until it is small enough to fit onto a compute node. However, if the division is performed using depth-first evaluation then the smaller sub-tasks will only be available to the compute nodes in sizes of decreasing magnitude. Thus the first sub-task shipped to a compute node will be the maximum size that that node can cope with, the next sub-task shipped will be half that size and so on. The first sub-task shipped may well not be small enough to be scheduled. If this is the case then the load-balancing properties of this type of evaluation do not hold, and the potential for dividing in parallel is not realized either. This clearly does not make good use of the available compute power. Ideally it would be best to ship successive sub-tasks of a maximum possible size to compute nodes in the parallel machine. This can be achieved using *breadth-first* evaluation [18].

3.3.2.3 Queue / stack-based breadth-first evaluation

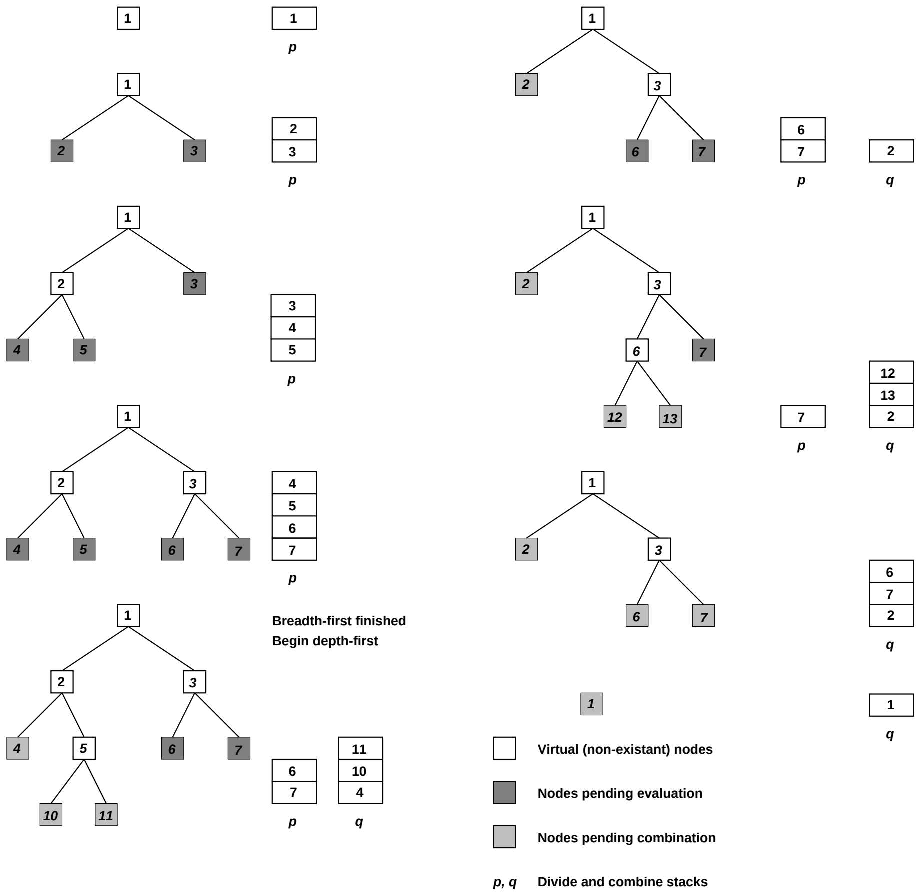


Figure 3.13: Queue / stack-based evaluation

The diagram shows how stack-based, breadth-first evaluation proceeds. The execution stack p *is initially used as a FIFO queue until the tree has been fully expanded. Operation is then switched to standard stack-based depth-first evaluation. Notice that* q *is unused until depth-first evaluation begins.*

Having developed a stack-based algorithm for depth-first evaluation, we would like to do the same for breadth-first evaluation. Sedgewick [96] describes such an algorithm for *traversing* a tree by using a *queue*. However, as before, because the tree is not in-place things are not quite so simple. Additionally, if we consider the operation of a queue it is obvious that we cannot dynamically schedule nodes from one end, since the *whole* queue is required for correct evaluation. See figure 3.13. Instead, we expand the tree until we have the desired number of nodes. At this point the ordering of the nodes from the *bottom* of the queue will be that of sequential left-to-right ordering of the tree. Thus, we can pop nodes from the bottom of the queue, like a stack, and evaluate these as for *depth-first*

```
Dac Dac::breadth(int depth) const
{
    Dac n,l,r;
    int s;
    DacStack::State t;
    static boolean nest=false;
/*
 * check that we are not already in a breadth evaluation
 */
    if (nest) error() << "nested breadth-first evaluation" << fatal;
    else nest=true;
/*
 * triviality check
 */
    if (simple()) {
        nest = false;
        return evaluate();
    }

    p->push(*this, DacStack::limbo, 1);
/*
 * we expand breadthwise while the level is less than the required depth
 */
    while ( (s = p->key()) < depth ) {
        n = p->pop();

        if ((n = n.divide(l,r)) != 0) error() << "unbalanced tree" << fatal;
/*
 * as soon as we come across something simple we stop expanding
 */
        if (l.simple() || r.simple()) depth = s+1;
/*
 * nothing will get offloaded until told to.
 */
        p->put(l, DacStack::limbo, s+1);
        p->put(r, DacStack::limbo, s+1);
    }
/*
 * breadth-first expansion finished - run and combine the rest
 */
    p->activate();

    while (!p->empty()) {

        s = p->key(); t = p->state(); n = p->pop();

        if (t & (DacStack::evaluated|DacStack::fixed)) q->push(n, s);
        else if (n.simple()) q->push(n.evaluate(), s);
        else q->push(n.run(), s);

        while (q->pair()) {
            s = q->key(); r = q->pop(); l = q->pop();
            l = l.combine(r);
            q->push(l, s-1);
        }
    }

    nest=false;                                    // we can run another evaluation now
    return q->pop();
}
```

Figure 3.14: Queue / stack-based evaluation

evaluation. The corollary to this is that we can pop nodes from the *front* of the queue for parallel evaluation. The queue is now acting in the same way as a stack. For combination we simply put evaluated nodes on a combine stack as we did for depth-first evaluation. We also label nodes with their depth so that combination is guaranteed to be correct as before. Thus, the only clever functionality we require is a queue that can be switched to stack-based operation, and some way of activating queued nodes for parallel evaluation. The algorithm is shown in figure 3.14.

3.4 System design

In this section we describe how D&C objects fit within an overall system. We describe the processes involved and the scheduling characteristics of the system.

3.4.1 Separation of origin from child processors

We have already described how an alternative main-line routine `mainCore()` can be used in place of `main()` so that system initialization can be performed in an *ordered* fashion. This also means that the generic operation of child processors can be separated from the specific operation of the root processor. Instead of requiring the programmer to mix operations that are specific to the root or child processors, we can specify all child oriented operations in `main()`, and only require the programmer to define `mainCore()`.

The operation of child processors can now be completely generic since a D&C object contains *all* information necessary to evaluate itself. Only D&C objects are evaluated by child processors, standard sequential code is evaluated on the origin processor. However, the program run on all processors is the same in content, even if not in operation. This is because the sequential program defines which D&C objects will actually be used, and so ensures that their definitions are linked in from libraries, or instantiated from templates. Unless the sequential code is very large, and this is unlikely since most functionality is encapsulated in D&C objects, the space overhead of having this information on child processors is minimal.

Since the evaluation of D&C objects is fully polymorphic, we can start the scheduling process at the beginning of a program and leave it running for the duration of the program. This process simply attempts to pull active objects from the bottom of the evaluation stack, and schedule them to other processors. If none are available then none are scheduled. Our overall system operation is then as shown in figure 3.15.

3.4.2 The scheduler

The scheduling process is designed to be as stateless as possible. Where state is necessary, communication messages are transmitted on different event identifiers. The scheduler simply answers requests, it initiates no communication exchanges. This is because all transfers – getting and returning objects – are initiated by the main routine on child processors. The scheduler answers three main requests: request for object to process, request to return processed object, request to terminate process. Other requests are introduced in section 5.5.1.2 for persistent objects. The object interactions are shown in figure 3.16.

```
<process command line arguments>
<execute sti() for all exemplars>
<start scheduling process>

IF <I am the root processor>
THEN
       mainCore()
       <send terminate message to other processors>
ELSE
       WHILE ( object = get_object() ) ≠ 0
              object.run()
       END
END

<stop scheduling process>
<execute std() for all exemplars>
```

Figure 3.15: Outline of D&C program execution

The diagram outlines the overall operation of the D&C run-time system. `main()` *is defined by the system, and it is up to the programmer to define* `mainCore()`. `main()` *processes command-line arguments, ensures that scheduling processes are started and stopped correctly and that global objects are initialized. Child processes continually process incoming D&C objects and return evaluated results.*

3.4.3 Memory management

In order to share objects between processes we need some sort of shared memory facilities. The main shared structure we have is the stack. To try and communicate data between the two processes using this structure, using message passing, would result in needless overhead and complexity.

The transputer, having a linear address space, has all of its memory accessible to processes running on it. Thus sharing the physical memory is easily accomplished. Less easily accomplished is shared management of this resource. Trollius^{TM} limits memory management on a per process basis. Memory descriptors are local to processes. Thus we need our own shared memory manager for D&C objects. We note that the blocks of memory we require are for D&C objects and are thus limited in size. They will be no smaller than the basic D&C class `DacRep`, and even the largest objects will be of the same order of size. Thus we choose Knuth's [67] *buddy* memory manager as a fast memory manager meeting these requirements. This system manages a single, contiguous block, allocating blocks in sizes of powers of two. We can tune the system so that the smallest block allocated is the size of `DacRep`.

The memory requirements will be largely dependent on the number of D&C objects created, rather than the individual sizes of these objects. This number is largely con-

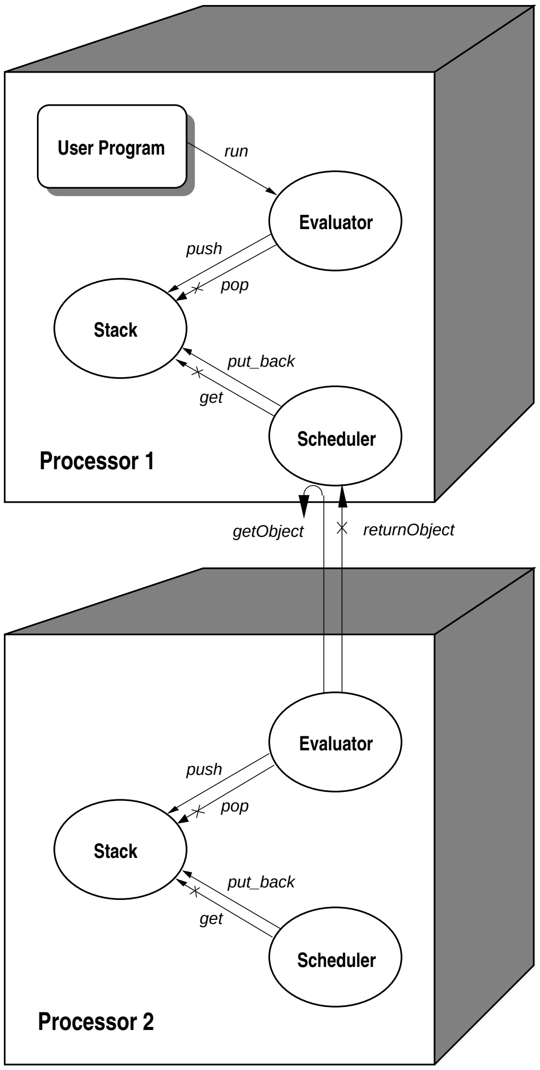


Figure 3.16: Scheduler communication exchange

The diagram shows the inter-processor and intra-processor interactions involved in D&C evaluation. The diagram uses Booch's notation which is given in figure 3.23. Notice that it is only the root processor that actually evaluates a user program. All other processors simply evaluate D&C objects.

stant for a given problem, and so it is feasible to select a managed block size just big enough to accommodate this number. However, between problems the requirements can vary tremendously, although only within a factor of ten or so. Thus we modify Knuth's algorithm so that if no free blocks are available, then another managed block is allocated of the same size as the first. This means that the total managed size will expand to meet the requirements of the problem in question, rather than starting with a block which may be far too big or too small.

Allocation and deallocation routines for the memory manager are semaphore locked, so that different processes using the system do not cause memory corruption. The manger is integrated into the D&C system by providing operators `new()` and `delete()` in `Dac` which call the manager's allocation routines.

The system design is similar to that used by Coplien's virtual constructor system [31].

3.4.4 System synopsis

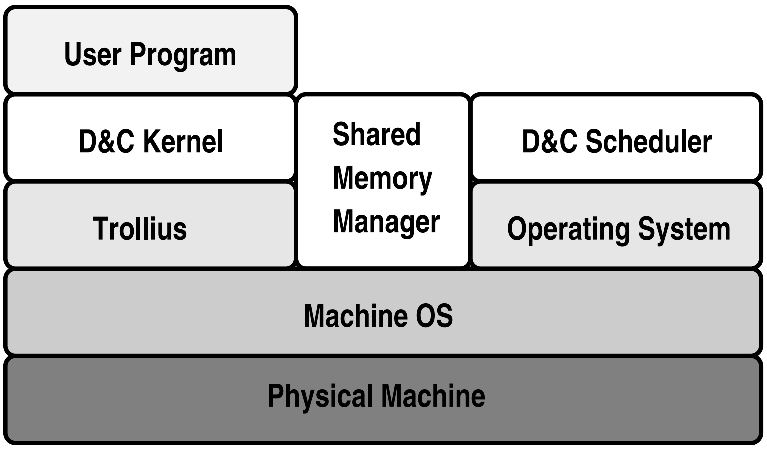


Figure 3.17: System design

The diagram shows a schematic representation of the D&C system in its hardware and software context. For transputer-based operation their is no shared-memory manager and no machine operating system. Note that the user program does not interact with the Trollius operating system. All run-time facilities are provided by the D&C kernel, and possibly the machine operating system.

The overall system involves two interacting processes layered on top of the Trollius^{TM} operating system, which in turn is in turn layered on top of either hardware or software depending on the target architecture. The relationships are shown schematically in figure 3.17.

Some system parameters – like problem size and partition – can be modified through command-line arguments. A full description of these is given in appendix C.

3.4.5 Summary

We have looked at the core definition of D&C classes. We have also looked at a system design that allows us to evaluate, simply and easily, objects of these classes. However, in any object-oriented system, design methodologies, interface definitions and evaluation principles are only half of the picture. These elements give a structure within which to work, but this structure is a *tool*, and we require basic *building blocks* in order to maximize the power of the overall system.

Thus most object-oriented systems have a comprehensive class library to provide these building blocks [74]. To this end we must provide such a library for our object-oriented D&C system. We now turn to the definition of some of these classes.

3.5 Core Beeblebrox classes

We look now at some members of the Beeblebrox library. The name Beeblebrox derives from the fictional character's first name *Zaphod* [1]. McBurney and Sleep's [72] original work was called the *ZAPP* project for Zero Assignment Parallel Processing. Thus we take this acronym and add Object-oriented Divide-and-conquer, throw in a random 'h', and take the surname just to be as obscure as possible.

Some of the classes we examine are concrete classes that can be used in their own right, but many are base classes from which more specific D&C classes can be defined. All these classes are documented more fully in appendix C.

3.5.1 Providing null functionality

The first classes we need are ones that fill in the blanks so to speak. The abstract interface `DacPure` requires, through pure virtual function definitions, that derived classes define most of the primary D&C functions. However, derived classes may not actually need to define this functionality; for instance an object that is discarded during `evaluate()` would not require `combine()`. In order to preserve the abstract interface, we provide null functionality in mix-in classes. Thus we define the mix-in `zCombine` which defines `combine()` to return itself. This and the other, equivalent classes are shown in figure 3.18.

3.5.2 Null D&C objects

There are certain instances when we want all of the D&C primary functions to be defined as null. For instance we might wish to have a D&C object that will operate conformingly in evaluation and transmission, yet does not actually do anything. We will see some more concrete examples of this in later sections. To this end we can define an object that simply mixes in all of the classes of section 3.5.1. However, experience shows us that this class is extremely useful, and, since multiple inheritance of virtual bases involves some overhead, we define a separate class, `zNull` having the appropriate functionality.

3.5.3 Container classes

The most common property of D&C classes is that of *containership*. Containership is denoted by the HAS-A relationship of artificial intelligence [21]. Most D&C operations are characterized by the manipulation of some sort of data structure. This structure is either a standard class, or some combination of other D&C classes. We identify three different D&C class groups:

- Classes, which we will call *containers*, that merely HAS-A standard object-oriented data-types. These classes might or might not define how their contained data should be manipulated.
- Classes, which we will call *envelopes*, which perform operations on a single, D&C operand. These classes have a HAS-A relationship with the operand.
- Classes, which we will call *compounds*, which perform operations on two, or possibly more, (HAS-A) D&C operands.

```
class zCombine : virtual public DacPure
{
  public:
    zCombine() {}

  protected:
    Dac combine(const Dac&) const { return wrap(); }
};

class zEvaluate : virtual public DacPure
{
  public:
    zEvaluate() {}

  protected:
    Dac evaluate() const { return wrap(); }
};

class zSimple : virtual public DacPure
{
  public:
    zSimple() {}

  protected:
    const boolean simple() const { return true; }
};

class zUnit : virtual public DacPure
{
  public:
    zUnit() {}

  protected:
    const boolean simple() const {
        return boolean(size()≤1);
    }
};

class zDivide : virtual public DacPure
{
  public:
    zDivide() {}

  protected:
    Dac divide(Dac&, Dac&) const { return wrap(); }
};
```

Figure 3.18: Null mix-in classes

The figure shows mix-in classes that provide place-holders for the primary D&C functions. These classes are used where a function needs to be defined for a class, but its operation is not relevant to the problem being evaluated.

```
template <class T> class zContainer : public zNull {
  public:
    typedef zContainer<T> Container_t;

    zContainer(const T& t) : zNull(), member(t) { }
    virtual T data() const { return member; }
    void init(const T& x){ member=x; }

  protected:
    T member;
};

template<class T> inline Dac Container(const T& t)
{
    return Dac( new zContainer<T>(t) );
}
```

Figure 3.19: Beeblebrox container class

The figure shows a D&C class for holding a standard C++ class. The containership relationship means that standard objects can be manipulated as D&C objects.

For each of these groups we will define an appropriate D&C class. For containers we define a class `zCompound<T>`, where the template formal is the data-type that the class HAS-A object of. Experience shows us that we often want to use `zContainer<T>`s simply for their transmission properties; the D&C functionality is largely irrelevant. To this end we derive `zContainer<T>` from `zNull` so that all the D&C functionality is accounted for.

For envelopes we define a class `zEnvelope`, where the class HAS-A a single data-member which is itself another D&C class. All primary functions of the `zEnvelope` will be forwarded to the data-member (its letter).

For compounds we will define a class `zCompound` which is very similar to the `zEnvelope` except that it HAS-A two D&C letters. As with the `zEnvelope` the D&C primary functions are forwarded to the two letters. See figures 3.19, 3.20 and 3.21. See also appendix C for more complete descriptions. The object relationships, using Booch's notation ([14], figure 3.23), are given in figure 3.22.

The primary reason for using these classes is that any, more specialized, classes we define will only vary in minor ways from these superclasses. This means that many of the D&C primary functions will not need to be redefined – especially those to do with object transmission. In addition, functions that do require modification will need added functionality rather than different functionality. We can therefore ensure that these functions call their parent's version before continuing with the specifics. This genericity reduces the amount of redundant code.

3.5.4 Array manipulation

Having described some general D&C classes we now look at a specific container class. This helps to illustrate the use of containers, as well as being a useful class in its own right.

The most frequent D&C structure that we have encountered is the array. This structure

```
class zEnvelope : public DacRep
{
  public:
    zEnvelope(const Dac& d) : e_letter(d) { }

  protected:
    Dac e_letter;

    Dac divide(Dac& l, Dac& r) const
    {
        l = make();
        r = make();

        return e_letter.divide(
            ptr_cast<zEnvelope>(l)→e_letter,
            ptr_cast<zEnvelope>(r)→e_letter
            );
    }
/*
 * Other definitions removed for clarity
 */
    Dac combine(const Dac& r) const;
    const boolean simple() const;
    Dac evaluate() const;
/*
 * And so on ...
 */
};
```

Figure 3.20: Beeblebrox envelope class

The figure shows a D&C class for holding other D&C objects. The primary D&C functions for the class ensure that the contained object is operated upon also. The level of indirection introduced is useful for providing generic functionality without reference to specific D&C types.

```
class zCompound : public DacRep
{
  public:
    zCompound(const Dac& l, const Dac& r) : left(l), right(r) { }
    void init(const Dac& l, const Dac& r) { left=l; right=r; }

  protected:
    Dac left;
    Dac right;

    Dac divide(Dac& l, Dac& r) const {
        Dac ll,lr,rl,rr,ret;

        left.divide(ll,lr);
        right.divide(rl,rr);

        l = make();
        r = make();

        ptr_cast<zCompound>(l)->init(ll,rl);
        ptr_cast<zCompound>(r)->init(lr,rr);

        return Dac(0);
    }
/*
 * Other definitions removed for clarity
 */
    Dac combine(const Dac& r) const;
    const boolean simple() const;
    Dac evaluate() const;
/*
 * And so on ...
 */
};
```

Figure 3.21: Beeblebrox compound class

The figure shows a D&C class for holding two D&C objects. The class exhibits similar features to those of `zEnvelope`*. Notice that* `divide()` *arranges for its left child to contain two left children from its contained objects, and likewise for the right child.*

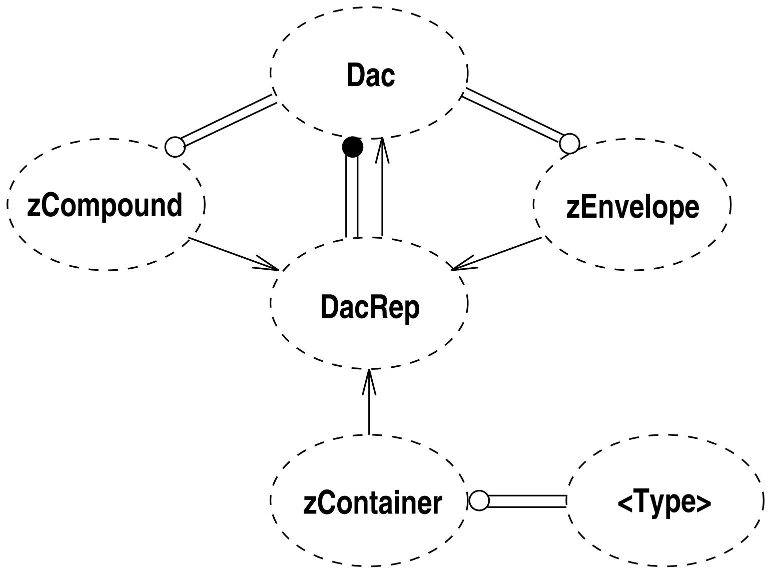


Figure 3.22: Container class relationships

The diagram shows the inheritance relationships between the D&C container classes, described earlier, and the standard D&C base classes. The diagram uses Booch's notation.

is easily partitioned due to its regularity. It is also efficiently communicable due to its contiguous nature. We define `zArray<T>` as its D&C counterpart. This class, virtually derived from `zContainer<BuiltinArray<T>>` provides a `BuiltinArray<T>` as a member, as well as the required D&C functions for I/O and dynamic casting. It also provides a `divide()` function for partitioning the array into two equal pieces. Virtual inheritance is used so that different types of primary function can be mixed in. A `BuiltinArray<T>` is defined assuming its array data can be transmitted block-wise, rather than iterating over each element. This vastly simplifies transmission. `BuiltinArray<T>` is derived from `Array<T>` which simply iterates over its array elements for transmission.

The structure is almost identical to the `zVector<T>` of section 5.3.3. However, providing a D&C structure that makes no assumptions about the algebraic properties of the array, is essential for arrays of complex data-types. Otherwise compilation is likely to fail from nonsensical operations being instantiated. A good example is that of an array of training frames for back-propagation (section 4.4.1). See figure 3.24.

3.5.4.1 Non-contiguous arrays

Originally, the structure was defined using an `Array<T>` class. However, it was found that this led to enormous problems in transmission. In C++, objects in arrays are initialized using default constructors. Thus to transmit an array of objects requires first the basic array information to be transmitted and a default array created. Then, individual object definitions are transmitted and assigned to the default place-holders. If the objects are small and the array large, then this involves a tremendous amount of overhead. A scheme was tried where heap allocation was constrained to allocate blocks of data contiguously, so

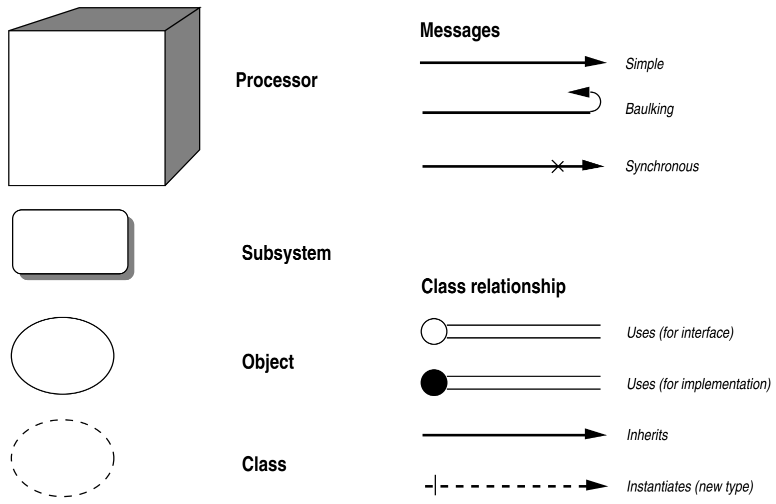


Figure 3.23: Booch's object notation

The diagram shows a subset of Booch's notation for object and class relationships used in this and subsequent chapters.

that the overall array could be transmitted in one go. However, this still led to a number of shortcomings, most notably a messy definition of the global operator `new()`. A new ANSI resolution now allows the definition of `operator new[]()`, which might possibly solve some of the problems. However it is far simpler to constrain the objects to be only data, i.e. no virtual functions and no pointers, and to treat them as simple types.

3.5.4.2 Reference counted arrays

`BuiltinArray<T>`s are reference counted to eliminate unnecessary data copying. However, we can provide some additional sophistication for the D&C system. In dividing an array, we initially only want to create new representations, rather than whole new arrays. To do this we can create new letters that hold new dimension information, but that point to the relevant parts of the original array data. We can then update the reference count of the original letter to reflect these additional references. Unfortunately, we may wish to delete the original letter while the other letters are still in existence. Thus we make each letter have a reference to its parent, so that only when all child letters are deleted will the original letter be deleted. This can be hidden transparently behind the array envelope. See figure 3.25.

This scheme is similar in operation to the structure representation of section 2.1.2, however, the optimization onus is put on the data holding structure rather than the D&C structure. This is a more strictly object-oriented approach, and results in greater code modularity and robustness. The D&C methods do not need to know whether array division is optimized or not, they will work regardless. Thus in defining standard data structures, we are on the look out for this sort of optimization. Although these optimizations are not D&C specific, it may be that only the D&C context will make the implementation worthwhile. Thus it is, that structures defined in standard libraries may not be suitable, efficiency-wise, for D&C. However, the converse is *not* true, structures defined in a suitable

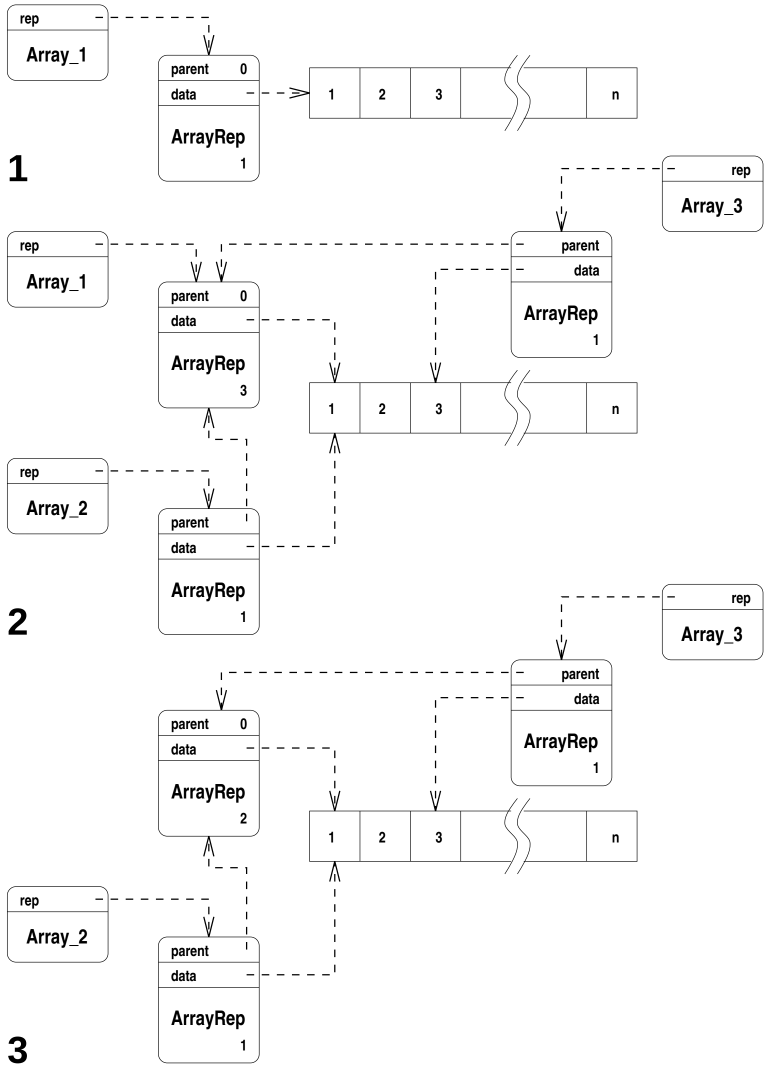
```
template <class T>
class zArray : virtual public zContainer<BuiltinArray<T> > {
  protected:
    zArray() {}
    zArray(const BuiltinArray<T>& a) { init(a); }

    Dac divide(Dac& l, Dac& r) const {
        l = make();
        r = make();

        ptr_cast<zArray<T> >(l)→init(
            BuiltinArray<T>(
                member.elems() / 2,
                member,
                0
                )
            );
        ptr_cast<zArray<T> >(r)→init(
            BuiltinArray<T>(
                member.elems() - member.elems() / 2,
                member,
                member.elems() / 2
                )
            );
        return 0;
    }
    const int size() const { return member.elems(); }
    ...
};
```

Figure 3.24: D&C array class

The figure shows a D&C class for manipulating arrays. `divide()` *is the key element of the class, yielding two D&C objects containing half each of the original array. Dynamic casting is used to obtain* `zArray` *objects from the generic* `Dac` *objects passed to* `divide()`*.*



The diagram shows the stages in dividing a reference counted array object. The original array has a reference count of 1 since it is the only referee. Then two more array objects are created with the original as a parent. The original array then has a reference count of 3 since it is referred to 3 times. Finally the original array object is deleted but the actual array data is not since a reference count still exists. Only when all references have been deleted will the data be removed as well.

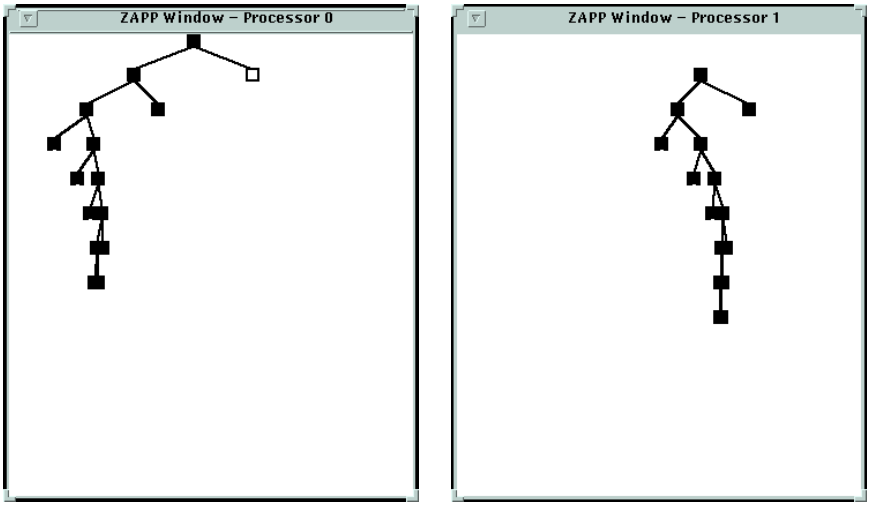
Figure 3.25: Reference counted arrays

way for D&C *will* be generally useful in other contexts.

3.5.5 Visualization

One of the last Beeblebrox classes we will consider here is one that allows us to visualize D&C in action.

Visualization has been a key tool in many parallel programming systems [53]. The



The diagram shows the screen display for the two processor evaluation of a D&C problem. Each solid black square represents a D&C object. Each hollow square represents an offloaded object, in this case an object has been offloaded to processor 1 and execution for that object continues on that processor. The original D&C object is represented by the topmost square on processor 0.

Figure 3.26: Screen display produced by a `zGraphic`

ability to see the run-time progress of a program can be the key to fault detection and performance enhancements. For the D&C system, the obvious visual metaphor is that of a tree. Unfortunately, since we now use stack-based evaluation, the way of displaying such a tree is not so obvious. We have two concerns, one is that visualization should not affect normal evaluation, i.e. the method used should be modular and separated from other program elements. The second concern is that visualization should be easily enabled for *any* standard D&C object. Activation should be through compilation, since we do not wish every D&C program to be carrying visualization baggage.

At first sight it would appear that we must define a class derived from the *envelope* `Dac`, and insert display code into versions of `run()` etc[5]. However, this denies our object-oriented outlook involving code duplication. Another, more serious, problem is that graphical information will be lost during parallel evaluation, since it is D&C wrappers that hold this information and these are frequently discarded.

It becomes obvious that the graphical control needs to be in the hand of the *letters*. Thus it is, that we define a D&C object `zGraphic` that is derived from a `zEnvelope`. This immediately solves most of our problems. The actual object for evaluation is the `zGraphic`'s letter, but the envelope handles display before delegating primary calls to the letter. This method is independent of the type of evaluation and the object being evaluated. Thus `run()`, `recurse()`, and `breadth()` should all be displayed equally correctly. To switch on visualization we merely wrap a D&C object in a `zGraphic`. A schematic

[5] And this is what was originally tried. It is a tribute to the new design that visualization can actually be accomplished far more simply.

```
class zGraphic : public zEnvelope
{
  private:
    static graphicswindow* theWindow;                    // window for display

    int depth;                                           // depth in tree
    int leaf;                                            // leaf number
    int xpos;                                            // x coordinate
    int ypos;                                            // y coordinate

  public:
    zGraphic(const Dac& d) : zEnvelope(d)
    {
        ...                                              // display an initial node
    }

  protected:
    Dac divide(Dac& l, Dac& r) const {
        Dac x = zEnvelope::divide(l,r);
        ...                                              // display two child nodes
        return x;
    }
    Dac combine(const Dac& d) const {
        Dac x = zEnvelope::combine(d);
        ...                                              // undisplay two child nodes
        return x;
    }
    void sti() {
        ...                                              // map a display window
    }
    void std() {
        ...                                              // unmap a display window
    }
    ...
};
```

Figure 3.27: D&C visualization class

The figure shows the basic layout of the D&C visualization class. Each primary function is forwarded to the contained object and the screen is updated.

structure is shown in figure 3.27. A screen dump of the resultant display is shown in figure 3.26.

3.5.6 Manipulating partition size

In figure 3.18 we showed the class `zUnit` which arranges for `simple()` to compare `size()` with unity. Experience shows us that this comparison is so common as to make it desirable to be the default behaviour. Thus we define `simple()` in this way in `DacRep`. This makes `zUnit` largely redundant, however, we have not dealt with the case where we want to compare with some fixed value other than unity. To this end we define a class `zPartition`. This class has a data member which represents the desired partition, which we can initialize by default from a command-line argument. Alternatively, the class can be explicitly initialized by the programmer. See figure 3.28.

```
class zPartition : virtual public DacPure {
  public:
    zPartition(int p=args::partition) : p_size(p) {};

  protected:
    int p_size;

    const boolean simple() const {
        return boolean ( size() ≤ p_size );
    }
};
```

Figure 3.28: Partition manipulation class

The figure shows a D&C mix-in class for altering the problem size at which D&C expansion stops. By default this size is 1.

3.6 Application to simple algorithms

We have described the ethos behind our object-oriented D&C system and described tools to enable D&C classes to be created quickly and easily. We have also described some core D&C classes. To illustrate their use, we now apply these ideas and techniques to a few simple algorithms.

3.6.1 Mergesort and quicksort

Two of the most well known D&C algorithms are quicksort and mergesort [96]. The former relies on the `divide()` phase of the algorithm to perform the sorting, the latter uses the `combine()` phase. Thus these two algorithms complement each other from a D&C point-of-view. In their purest form, no calculation is performed during `evaluate()`, however, to yield efficiency, some `evaluate()` processing is necessary.

We look first at quicksort.

3.6.2 Quicksort using D&C

Quicksort was invented in 1960 by C. A. Hoare. It is a good general purpose sorting algorithm, that is efficient in a wide variety of situations. Quicksort works by partitioning data into two parts, and then sorting the parts independently. The exact position of the partition depends on the data involved.

The data is partitioned as follows. First the rightmost element of the data is chosen to be in the correct position. This is then used as the partition. The data is then scanned from both ends until an element on the left is found which is greater than the partition, and an element on the right is found which is less than the partition. These elements are then swapped, and the procedure continues until the two scans cross. At this point the partitioning element is swapped with the leftmost element of the right sublist.

Once the data has been partitioned, the procedure is recursively applied to the two sublists. However, because our implementation already has the recursion built in, we only need to specify the partitioning operation.

3.6.2.1 D&C implementation

In order to sort something we need a structure to sort. The most common and useful structure to sort is an array. Thus we select the `zArray<T>` of section 3.5.4 as a base class for our quicksorting class.

```
template <class T>
class zCombineArray : virtual public zContainer<BuiltinArray<T> > {
  protected:
    zCombineArray() {}

    Dac combine(const Dac& r) const
    {
        Dac d = make();
        ptr_cast<zArray<T> >(d)→init(
            member | ptr_cast<zArray<T> >(r)→data()
            );
        return d;
    }
};
```

Figure 3.29: D&C array concatenation

The figure shows a D&C mix-in class that can be used to concatenate array-type results together. `combine()` *simply uses the array* `operator|` *to do the concatenation, and it is assumed that this operator has been defined.*

Starting with this class as a base, we now need to consider the recombination of results. For quicksort this is a simple matter of concatenating sorted sublists together. To implement this using the Beeblebrox library is a simple matter of mixing in a `zCombineArray<T>`. This class provides a `combine()` function which performs the required concatenation for `zArray<T>`s (see figure 3.29). For the moment, we will assume that the sort procedure continues until sublists are of unit length. We therefore require an `evaluate()` function that simply returns its parent object. This can be easily defined or mixed-in from

```
template <class T>
Dac zQuicksort<T>::divide(Dac& l, Dac& r) const
{
    T q,t;
    int i,j;
    BuiltinArray<T> a = member;

    if (a.elems() > 1) {
        i=-1; j=a.elems()-1; q=a[a.elems()-1];
        for (;;) {                                          // find the partition
            while (a[++i] < q);
            while (a[--j] > q);
            if (i≥j) break;
            t = a[i]; a[i] = a[j]; a[j] = t;                // swap elements
        }
        if (a[i] ≠ a[a.elems()-1]) {                        // swap partition
            t = a[i]; a[i] = a[ a.elems()-1 ]; a[ a.elems()-1 ] = t;
        }
        if (i==0) i++;

        l = make();
        r = make();

        ptr_cast<zContainer<BuiltinArray<T> > >(l)→init(
            BuiltinArray<T>( i, a, 0)
            );
        ptr_cast<zContainer<BuiltinArray<T> > >(r)→init(
            BuiltinArray<T>( a.elems() - i, a, i )
            );

        return 0;
    }
    return wrap();
}
```

Figure 3.30: Quicksort partitioning

The figure shows the essence of the quicksort algorithm – array partitioning – implemented in `divide()`*. The procedure partitions the original array member into a new array object,* `a`*, and then this object is used to create two new D&C objects, each containing half of the partitioned array.*

```
template <class T>
class zQuicksort : public zArray<T>,
                   public zCombineArray<T>,
                   public zEvaluate
{
  public:
    zQuicksort(const BuiltinArray<T>& a) : zArray<T>(a), zCombineArray<T>() {}

  protected:
    Dac divide(Dac& l, Dac& r) const;
};
```

Figure 3.31: A quicksort D&C class

The figure shows a D&C class that can be used for quicksorting arrays. The class defines array partitioning in `divide()`*. Other functionality is provided through mix-ins.* `zCombineArray<T>` *provides* `combine` *since combination is simply a matter of concatenation.* `zEvaluate` *provides* `evaluate()` *since evaluation is a null operation.*

a `zEvaluate`. All that remains to be done is to define the nuts and bolts of quicksort – `divide()` – and add the standard D&C auxiliary functions. The auxiliary functions can be added automatically using `dac-mode` in emacs. The division function is shown in figure 3.30 and the class in figure 3.31.

If we wish to gain a little more efficiency, then `evaluate()` can be defined to perform a general sorting algorithm. This could even be provided in a general mix-in class.

3.6.3 Mergesort using D&C

The basis of mergesort is combining two sorted data sets to yield a single sorted data set. If the initial data sets are of unit length, then the combine procedure can be used to sort the entire set. This merge procedure is simple in operation. Successive elements from the two data sets are compared and the smaller written to the resultant set. If an element was not written then that element is used for the next comparison. The procedure continues until all elements have been written.

To actually sort a data set, the set is first recursively split into smaller sets. These smaller sets are sorted, and then recursively combined using the merge operation above.

3.6.3.1 D&C implementation

The D&C implementation for mergesort is very similar to that of quicksort, only this time the core functionality is in `combine()`. We again start with a `zArray<T>` and this provides the recursive splitting that we require. Again, evaluate can be defined using `zEvaluate` or by mixing in a standard sort procedure. All that remains is to define an appropriate `combine()`, which is shown in figure 3.32, auxiliary functionality is defined using `dac-mode`. The actual class is shown in figure 3.33.

```
template <class T>
Dac zMergesort<T>::combine(const Dac& d) const
{
    BuiltinArray<T>
        I = ptr_cast<zContainer<BuiltinArray<T> > >(*this)→data(),
        J = ptr_cast<zContainer<BuiltinArray<T> > >(d)→data(),
        r(I.elems() + J.elems());

    int i=0, j=0;

    for (int k=0; k<r.elems(); k++) {
        if (i==I.elems()) r[k] = J[j++];
        else if (j==J.elems()) r[k] = I[i++];
        else r[k] = (I[i]<J[j]) ? I[i++] : J[j++];
    }
    Dac x=make();
    ptr_cast<zContainer<BuiltinArray<T> > >(x)→init(r);
    return x;
}
```

Figure 3.32: Mergesort combining

The figure shows sorted array merging for a D&C mergesort class. Once a sorted array has been generated, the returned D&C object is initialized with it.

```
template <class T>
class zMergesort : public zArray<T>,
                   public zEvaluate<T>
{
  public:
    zMergesort(const BuiltinArray<T>& a) : zArray<T>(a) { }

  protected:
    Dac combine(const Dac& r) const;
};
```

Figure 3.33: A mergesort D&C class

The figure shows a complete D&C mergesort class. Binary division is provided by the `zArray<T>` *class by default, and* `evaluate()` *is again a null operation.*

3.6.4 Other algorithms

These two examples give an indication as to the ease of programming using the Beeblebrox library. The only functions that need be hand coded are the same as those for the serial case. All other functionality is generated automatically, or has already been defined in a library class.

There are many other simple algorithms that can be implemented easily using D&C. Graph traversal, for instance, or finding the convex hull of a set of points [30, 96]. Dynamic programming problems are essentially D&C in operation, although they might also need some additional functionality to operate properly [72].

3.7 Summary

We have taken a detailed look at the design of a object-oriented D&C system. We have related the design to the actor model and described the ethos behind defining D&C classes. Finally we have looked at some core D&C classes and the implementation of some simple D&C algorithms. In the following chapters we look at some more complex algorithms and their implementation using our object-oriented D&C system. We also look at some additions to the Beeblebrox library, which allow greater flexibility in defining and evaluating D&C objects.

Much of chapter 3 is taken from [83], submitted to Concurrency Practice and Experience.

Chapter 4

Extending the design and applications

Vell, Zaphod's just zis guy you know. *Gag Halfrunt*

We have looked at the basic design of an object-oriented D&C system. We now consider some different applications to investigate what patterns emerge, and what improvements could be made. We consider heuristically both the programming and run-time aspects of the system, as well as performing a mathematical analysis of the run-time performance.

First of all, however, we consider some different D&C manipulation structures.

4.0.1 Chapter organization

In section 4.1 we look at some D&C structures other than the simple array. In section 4.2 we look at some extensions to the mix-in strategy introduced in section 3.1.4.3. In section 4.3 we look at some Beeblebrox classes for enhanced evaluation schemes. In sections 4.4 and 4.5 we look at some further applications. In section 4.6 we look at the performance results for these problems and then finally, in section 4.7, we look theoretically at the performance we would expect.

4.1 List processing

So far we have only considered D&C objects based on arrays. However, there are many instances in which it would be preferable to use a linked list. Linked lists have the desirable property that they are very easy to concatenate. Their drawback is that there is significant overhead involved in finding the middle of a list. This is because, unlike arrays, the elements need to be iterated over from the beginning of the list.

In order to implement a D&C list class we require a fairly comprehensive list class, preferably reference counted. The only such class, that is publicly available, is in the JCOOL class library, a derivative of Texas Instruments' COOL library. Unfortunately, this class does not compile under the available software, and also needs major modification to use in a message passing environment. Instead, we define our own list class that is parameterized and reference counted. The structure is outlined in figure 4.1. Essentially, each list node has a reference count associated with it, as well as a next pointer and a data item. To split a list into two, two new list objects are created. The first's head points to the beginning of the original, its tail to the middle. The second list object is similarly

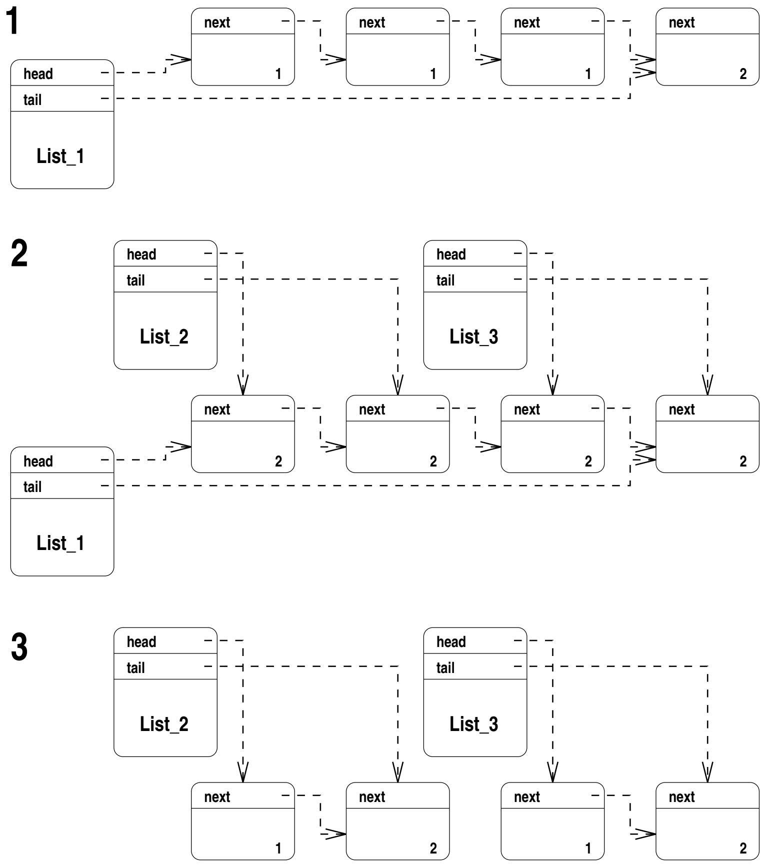


Figure 4.1: A reference counted list

The diagram shows the stages in creating two sub-lists from a single reference counted list. The two sub-list are simply pointers to positions within the original list. Only when the original list is deleted do the sub-lists become the owners of the actual data. List elements are deleted when their reference count goes to zero.

associated with the end of the original. No new list elements are created. The reference counts of those elements referenced by the new list objects are simply incremented. When the original list is deleted, its reference count is decremented. Only if this is zero, are the actual elements deleted. This is ideal for D&C, where we wish division to be very cheap. The scheme is quite similar to the enhanced array of section 3.5.4.

In order to create the D&C equivalent of a list we simply use a `zContainer<List<T>>` as a base class and then add whatever functionality is necessary. In our case the main requirement is division of lists; concatenation is trivial. Thus the essence of the class is shown in figure 4.2. Auxiliary functions are automatically inserted using `dac-mode`. Additional flexibility can be achieved by giving the structure a mix-in hierarchy, figure 4.3, as we did for `zArray<T>`.

This all works cleanly. If we wished to sort a list, for example, then we could create a version of the mergesort class given in section 3.6.3.

```
template <class T>
class zList : virtual public zContainer<List<T> > {
  public:
    zList(const List<T>& x) { init(x); }

  protected:
    Dac divide(Dac& p, Dac& q) const
    {
        p = make();
        q = make();

        ptr_cast<zContainer<List<T> > >(p)→init(
            List<T>(member.first(),member.nth(member.size()/2),member.size()/2)
            );
        ptr_cast<zContainer<List<T> > >(q)→init(
            List<T>(
                member.nth(member.size()/2+1),member.last(),
                member.size()-member.size()/2
                )
            );

        return 0;
    }
};
```

Figure 4.2: A D&C list class

The figure shows a D&C list class that splits its contained list in `divide()`*. The class relies on the* `List<T>` *class having the appropriate functions. Division is optimized by using reference counted lists.*

```
template <class T>
class zList : virtual public zContainer<List<T> > {
  public:
    zList(const List<T>& x) { init(x); }

  protected:
    Dac divide(Dac& l, Dac& r) const;
};

template <class T>
class zCombineList : virtual public zContainer<List<T> > {
  protected:
    zCombineList(){}
    Dac combine(const Dac& r) const
    {
        Dac d = make();
        ptr_cast<zContainer<List<T> > >(d)→init(
            member + ptr_cast<zContainer<List<T> > >(r)→data()
            );

        return d;
    }
};

template <class T>
class zEvaluateList : virtual public zContainer<List<T> > {
  protected:
    zEvaluateList() {}
    Dac evaluate() const
    {
        Dac d=make();
        ptr_cast<zContainer<List<T> > >(d)→init(member.clone());
        return d;
    }
};
```

Figure 4.3: D&C list class with mix-in support

The figure shows an extended class hierarchy for D&C lists. The class `zList<T>` *is the same as before. However, other mix-in based classes have been added which can be used to generate classes which concatenate lists in* `combine()`*, or clone lists in* `evaluate()`*. Classes wishing to use this functionality would simply inherit from the appropriate classes. Note that all the container relationships are* `virtual` *so that there is only ever one data item per object.*

Ideally, we might think that unifying the list and array structures would simplify definitions like this. However, in a parallel processing context, we are concerned with *performance*, and not taking advantage of the implicit structure of an array would considerably reduce performance . We are always making the assumption that, although D&C introduces considerable overhead, the 'inner loop' of our code (generally `evaluate()`) is nearly as fast as is possible. This assumption is only true if we have direct access to the underlying structure of the problem.

4.1.1 D&C lists of D&C objects

We now consider a special form of D&C list. The list nodes in a D&C list, by virtue of parameterization, can contain any sort of object. What would happen if we were to make the type of these objects *Dacs themselves*? Dividing and combining the list would be identical to that of a standard `zList<T>`, however, we could make `evaluate()` actually *run* each D&C object. How might this be useful? It is possible to envisage a scenario where problem execution involves a number of independent sub-problems. If each of these problems were to be executed sequentially then the serialness introduced would impair performance. What we really want to do is run each of these tasks concurrently. If we were to assemble each sub-task into a D&C list. Then running the list would achieve the desired effect [18].

```
class DacList : public zList<Dac>,
                public zCombineList<Dac>
{
  public:
    DacList(const Dac& d) : zList<Dac>(List<Dac>(d)) {}
    DacList(const Dac& a, const Dac& b) : zList<Dac>(List<Dac>(a,b)) {}
    DacList(const DacList& d) : zList<Dac>(d) {}
    const DacList& operator= (const DacList& d) { member=d.member; return d; }

    DacList append(const Dac& d) { member.append(d); return *this; }
    DacList operator, (const Dac& d) const { return DacList(*this).append(d); }
    DacList operator+ (const Dac& d) const { return DacList(*this).append(d); }

  protected:
    Dac evaluate() const
    {
        return Dac(new DacList(member.item().run()));
    }
};
```

Figure 4.4: A D&C list of D&C objects

The figure shows a D&C list of D&C objects. Each D&C object can be of any type and is run in `evaluate()`. *The class inherits from* `zCombineList<T>` *since the list elements need to be concatenated after evaluation.*

In order to facilitate the assembly if D&C lists we can include overloaded versions of the operators '+' and ','. The resultant class is shown in figure 4.4. Another approach to this problem is given in section 4.3.1.

4.1.2 D&C Lisp

In the preceding sections we have introduced different sorts of D&C lists. We have also described some different ways of manipulating these lists. With the `DacList`, we would especially like the manipulation and creation of this structure to be simple, and notationally convenient. Since list manipulation is at the core of a number of programming languages, it would be desirable to use syntax that is already well-known. The obvious choice of language is the LISt Processing language LISP [99].

Lisp in C++ is already the subject of the class library Lilly. We will not attempt to implement even a fraction of LISP functionality, we will simply use LISP syntax where appropriate. Thus functions that might be useful are `car()`, `cdr()` and `nth()` for dissecting lists. Implementing these functions is trivial if they are supported by the underlying list class.

4.2 Generalizing mix-in support

Mix-ins provide us with an invaluable way of combining D&C functionality. We now discuss some enhancements that we can make to aid in the definition, and use, of D&C mix-ins.

4.2.1 Creating mix-ins using templates

```
template <class Type, class Divide, class Combine, class Evaluate>
class Mixin : public Divide, public Combine,
              public Evaluate {
  public:
    Mixin(const Type& t) : Divide(t) {}
    ~Mixin();

  protected:
    Mixin(const Mixin& z) : Divide(z) {}
    Mixin(ibstream&);
    Mixin(DacRep::Exemplar e) : Divide(e) { }
    DacRep* scan(ibstream&) const;
    obstream& spawn(obstream&, const boolean=true) const;
    Dac make() const;
  private:
    static DacRep* exemplar;
};
```

Figure 4.5: Template mix-ins

The figure shows a parameterized class for combining mix-ins. The class inherits from its formal parameters which are assumed to be compatible mix-in classes. Note that the class would be considerably more useful if default arguments were supported for parameterized classes.

In section 3.1.4.3 we discussed mix-in support for D&C classes. In subsequent sections, we have made use of mix-in classes to enable rapid and flexible assembly of D&C classes. However, in creating a D&C class from mix-in classes, we often don't define any new functionality. The problem is that we still have to use `dac-mode` to insert all the required auxiliary functions. A better solution would be to provide a template class for which the template formals are the mix-in classes to use. Thus we might have a structure such as shown in figure 4.5. To use this we would simply declare an object using the types we require. For example with the `zList<T>` class we defined in section 4.1, the declaration might be:

```
zMixin<int, zList<int>, zCombineList<int>, zEvaluateList<int> > alist;
```

```
#define ZMIXIN3(Name,Type,Divide,Combine,Evaluate) \
class Name : public Divide, public Combine, \
             public Evaluate { \
  public: \
    Name(const Type& t) : Divide(t) {} \
    ~Name(); \
                                                          \
  protected: \
    Name(const Name& z) : Divide(z) {} \
    Name(ibstream&); \
    Name(DacRep::Exemplar e) : Divide(e) { } \
    DacRep* scan(ibstream&) const; \
    obstream& spawn(obstream&, const boolean=true) const; \
    Dac make() const; \
  private: \
    static DacRep* exemplar; \
}; \
```

Figure 4.6: Macro mix-ins

The figure shows a class macro for combining mix-ins. This macro can be used when a compiler does not support inheritance from template formals. The macro can be used in a similar way to the parameterized mix-in combiner described earlier.

We would have to provide other classes if we wished to mix in less, or more, of the primary functionality. Unfortunately, not all compilers support this syntax as yet, although it is legal ANSI C++. Thus we must revert to the macro processor to do the work for us. See figure 4.6. Of course this gives us slightly less flexibility, since the class must be declared at global scope; this is done automatically with template classes. However, the macro version is perfectly usable, though perhaps not aesthetically pleasing. Some object-oriented languages have direct support for adding functionality to a class [13]. C++ is not among them, so we must look for techniques like this to help us.

4.2.2 Parameterizing mix-in hierarchies

In previous sections (4.1.1) we have described the creation of D&C structure classes from standard classes. The creation of these classes has often involved defining mix-in versions for each primary function. Thus we might have a list D&C class for which we define a derived, mix-in class for division, combination and evaluation. Often, as is the case in the examples we have examined, the base class of the hierarchy is simply a container, and the only new functions that are defined are `divide()` etc. Since we can characterize the hierarchy in this way, we ought to be able to define a parameterization for this characterization. We can do this using template classes for which the formal parameter is the class we are trying to contain. The functions that need to be defined individually can be done so as specializations. See figure 4.7. Note that part of the beauty of this approach is that D&C auxiliary functions are already defined. As always, for clarity, these are not shown in the figure.

```
template <class T>
class zTemplate : virtual public zContainer<T> {
  public:
    zTemplate(const T& x) { init(x); }
    T* operator-> () { return &member; }

  protected:
    Dac divide(Dac& l, Dac& r) const;
};

template <class T>
class zCombineTemplate : virtual public zContainer<T> {
  protected:
    zCombineTemplate(){}
    Dac combine(const Dac& r) const;
};

template <class T>
class zEvaluateTemplate : virtual public zContainer<T> {
  protected:
    zEvaluateTemplate(){}
    Dac evaluate() const;
};
```

Figure 4.7: Parameterized mix-in hierarchy

The figure shows how parameterized classes can be used to create mix-in hierarchies. For the classes shown the programmer would have to only explicitly define the primary functions `divide()`, `combine()` *and* `evaluate()`.

One problem with this approach is that many compilers do not allow generic specializations. These can occur when defining a template of a template, e.g. `Cat<X<T>>`, and we want to define a specialization, e.g. `Cat<Mat<T>>::foo()`, but not specify the innermost template formal, for instance `Cat<Mat<int>>::foo()`.

4.2.3 Dynamic inheritance in place of mix-ins

The mix-in techniques we have described enable a programmer to rapidly prototype D&C classes. However, these techniques are based on static inheritance and depend heavily on templates. These two factors mean that their use is hindered by the creation of numerous different classes, resulting in global namespace pollution, large executable size, and very long compilation times. These problems would be largely alleviated if we could use dynamic inheritance, as supported by some message-passing based object-oriented languages [106]. However, because the range of functions we might want to define is quite limited (four at most), and because the interface for the classes we want to use is well defined (they are all `Dacs`), we can implement our own form of dynamic inheritance.

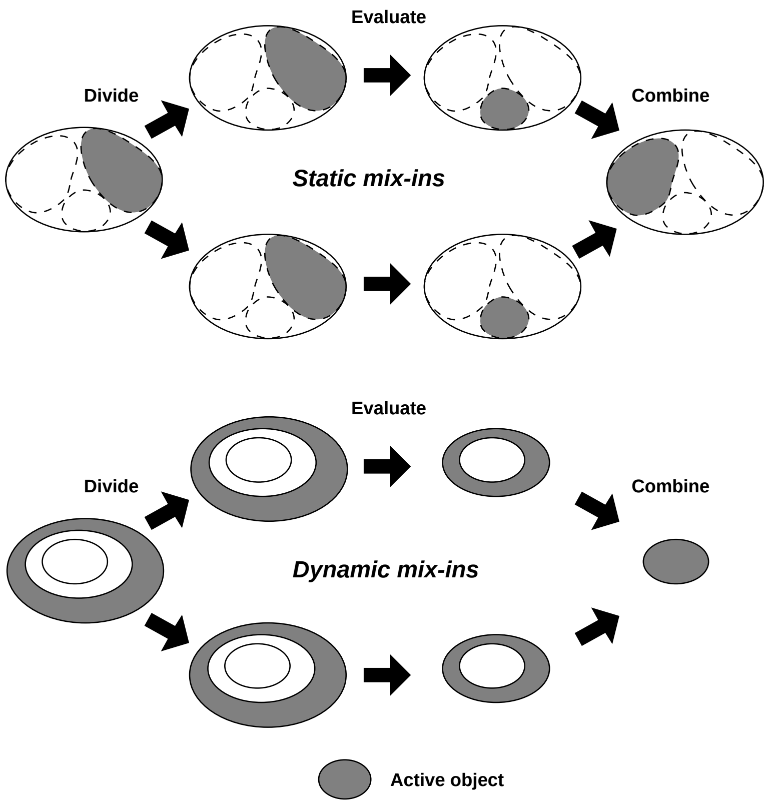


Figure 4.8: Object metamorphoses

The diagram highlights the differences between standard, static, mix-in based evaluation, and dynamic mix-in based evaluation. With standard mix-ins primary functionality is selected at compile time, and objects are subsequently invariant, each primary function being used at different stages through the evaluation. With dynamic mix-ins primary functionality is selected at run-time, and each object has one of the primary functions. As functions are used the objects that contain them are discarded.

Recall that object interactions in D&C are similar to those of actors [2]. We view our D&C world as consisting of objects which are constantly mutating, dividing or combining, much like animal cells. These state changes are normally effected by one part of the object which is active for the particular operation in question. The active part can be defined by a mix-in; in which case the object is really a combination of different objects, or by a function definition. The scenario is still the same – a unitary object metamorphosed by actions defined by its constituents. However, it is possible to view these object actions differently. Instead of viewing actions as constituents of the whole, we provide a one-to-one mapping between actions and objects. Thus each object is responsible for a particular function. Then, instead of combining these objects together to achieve total functionality,

```
class zManipulator : public zEnvelope {
  public:
    zManipulator(const Dac& d) : zEnvelope(d) { }
};

class zDivideManip : public zManipulator
{
    typedef Dac (*PFD)(const Dac&, Dac&, Dac&);

  public:
    zDivideManip(const PFD dp) : zManipulator(), divide_p(dp) { }
    zDivideManip(const PFD dp, const Dac& d)
        : zManipulator(d), divide_p(dp) { }
    Dac create(const Dac& d) const {
        return Dac(new zDivideManip(divide_p,d));
    }
  protected:
    PFD divide_p;

    Dac divide(Dac& l, Dac& r) const
    {
        l = make();
        r = make();
        return (*divide_p)(*this, l, r);
    }
};

#define DEFDIV(d_name) \
        Dac name2(z,d_name)(const Dac&, Dac&, Dac&); \
        zDivideManip d_name(name2(z,d_name)); \
        FUNCTOR(Dac,z##d_name,(const Dac& self, Dac& left, Dac& right))
```

Figure 4.9: Dynamic `divide()` mix-in

The digram shows a dynamic mix-in class that performs `divide()` *type operations. Objects of this class are initialized with a function through the use of the macro* `DEFDIV()`*. This function is then called by the class'* `divide()` *member. Note that the class is a* `zEnvelope` *since it will contain another D&C object to which functions other than* `divide()` *will be forwarded.*

```
class DacManip : public Dac {
  public:
    DacManip(const zManipulator& a, const zManipulator& b,
             const zManipulator& c, const zManipulator& d,
             const Dac& dd) : Dac()
    {
        Dac::operator=(create(dd,&a,&b,&c,&d));
    }
/*
 * and so on for the other constructors ...
 */
    DacManip(const zManipulator&, const zManipulator&,
             const zManipulator&, const Dac&);
    DacManip(const zManipulator&, const zManipulator&,
             const Dac&);
    DacManip(const zManipulator&, const Dac&);

  private:
    Dac create(const Dac& dd, const zManipulator* a, const zManipulator* b=0,
               const zManipulator* c=0, const zManipulator* d=0) const
    {
        zDivideManip *dm=0;
        zCombineManip *cm=0;
        zEvaluateManip *em=0;
        zSimpleManip *sm=0;

        const zManipulator* dlist[4];
        dlist[0] = a;
        dlist[1] = b;
        dlist[2] = c;
        dlist[3] = d;

        for (int x=0 ; x<4; x++) {
            if (dlist[x]) {
                if ( ptr_cast<zDivideManip>(dlist[x]) )
                    dm = ptr_cast<zDivideManip>(dlist[x]);
                else if ( ptr_cast<zCombineManip>(dlist[x]) )
                    cm = ptr_cast<zCombineManip>(dlist[x]);
                else if ( ptr_cast<zEvaluateManip>(dlist[x]) )
                    em = ptr_cast<zEvaluateManip>(dlist[x]);
                else if ( ptr_cast<zSimpleManip>(dlist[x]) )
                    sm = ptr_cast<zSimpleManip>(dlist[x]);
            }
        }

        Dac p;
        if (cm) p=cm→create(dd);
        if (em) p=em→create(p);
        if (sm) p=sm→create(p);
        if (dm) p=sm→create(p);

        return p;
    }
};
```

Figure 4.10: Combination class for dynamic mix-ins

The figure shows a D&C wrapper class that can be used to combine dynamic mix-in classes such as `zDivideManip`*. Objects of this class can be evaluated identically to* `Dac`*s. The class constructor ensures that the mix-in objects are ordered correctly.*

```
#include <Vector.h>
#include <zVector.h>
#include <zManip.h>

DEFDIV(zappo)
{
    ptr_probe<zContainer<int> >(left)->init(
        ptr_probe<zContainer<int> >(self)->data() /2
        );

    ptr_probe<zContainer<int> >(right)->init(
        ptr_probe<zContainer<int> >(self)->data() *2
        );
    return 0;
}

DEFCOMB(zippo)
{
    ptr_probe<zContainer<Vector<int> > >(result)->init(
        ptr_probe<zContainer<Vector<int> > >(left)->data()
        +
        ptr_probe<zContainer<Vector<int> > >(right)->data()
        );
}

DEFEVAL(zyppo)
{
    ptr_probe<zContainer<Vector<int> > >(result)->init(
        ptr_probe<zContainer<Vector<int> > >(self)->data() * 2
        );
}

extern "C" void mainCore()
{
    Vector<int> v(10);
    v = Random(-5,5);
    Dac x = DacManip( zippo, zyppo, new zVector<int>(v) );
    cout << x.run() << endl;
}
```

Figure 4.11: Example usage of dynamic mix-ins

The figure shows how dynamic mix-ins can be used. Three mix-in functions are defined using macros similar to `DEFDIV()`*. Note that these functions use* `ptr_probe<T>` *to access D&C data. The local variables* `result`, `left`, `right` *and* `self` *are defined by the macros. The defined mix-in classes are then combined with a standard* `zVector<T>` *object using a* `DacManip`*, and evaluated.*

as in static inheritance, we encase each object inside one of the others. Evaluation then involves objects forwarding operations that they do not know how to process. At each stage of evaluation, one particular object is active, just as in the static inheritance case, but compile time dependencies have been removed. See figure 4.8.

These classes need an actual function for processing, which we provide as a pointer. If we were to parameterize upon a function, then we would be back to square one in terms of code explosion. To simplify the overall use of these classes we define a macro which creates an object from a function definition. This object can then be used, using `create()`, to clone objects of the same type, i.e. using the same function. See figure 4.9. Similar classes are provided for the other D&C functions. To actually combine these objects into a cohesive whole, we define a class `DacManip` which arranges objects into the correct order, and creates an aggregate D&C object from them. See figure 4.10. An example usage of these classes is given in figure 4.11.

Of course, the space and compile-time overheads we have eliminated, have been traded for a degree of performance. However, the scheme does allow extremely rapid and simple prototyping of D&C objects. If it is found that better performance is required, then standard mix-in classes can be defined. These techniques are similar to instance level mix-ins [54].

4.2.3.1 Extensions to dynamic casting

In section 3.1.4.5 we described a dynamic casting scheme for D&C objects. In the dynamic mix-in scheme that we have just described, the mix-in functions will not be able to tell how many envelopes deep the object they wish to manipulate is. This is because the actual D&C object that is operated upon may provide some of the D&C functionality. Thus dynamic casting of a D&C wrapper is not sufficient. What we actually need to be able to do is to follow, transparently, container relationships. We therefore provide an additional casting class `ptr_probe<T>`.

In order to follow container relationships we provide an additional parameter to `get_this_ptr()` which says whether to do so or not. For an envelope, `get_this_ptr()` will initially try a dynamic cast before following container relationships. A similar scheme applies for compounds. `get_this_ptr()` for envelopes is shown in figure 4.12. `ptr_probe<T>` is used in an identical manner to `ptr_cast<T>`.

4.3 Evaluation enhancements

In the next two sections we look at two ways we can improve performance by a greater and more even distribution of parallelism. In any parallel system we are always trying to exploit parallelism, thus any improvements in this area are desirable.

4.3.1 Concurrent D&C object execution

In section 4.1.1 we considered a D&C list of D&C objects. This structure enabled different types of D&C object to be evaluated in parallel. However, the drawback of this approach is that all D&C objects need to have been created in order for evaluation to proceed. An alternative approach might be to allow D&C objects to be evaluated as they are created, as normal, but not to wait for the results before proceeding to the next object. This would overcome the need to wait for all objects to be created. The choice, however, between

```
template <class T> class ptr_probe {
    const DacRep* p_d;
  public:
    ptr_probe(const Dac& d) : p_d(d.rep()) {}
    ptr_probe(const DacRep* d) : p_d(d) {}
    operator T*()
    { return (T*)(p_d->get_this_ptr(int(T::exemplar->type), true)); }
    T* operator-> ()
    { return (T*)(p_d->get_this_ptr(int(T::exemplar->type), true)); }
};

class zEnvelope : public DacRep
{
  protected:
    Dac e_letter;
    void* get_this_ptr(int i, int p) const {
        void *v=0;
        if (i == int(exemplar->type)) {
            v = (void*)this;
        }
        else if ( (v = DacRep::get_this_ptr(i,p)) == 0 && p) {
            v = e_letter.get_this_ptr(i,p);
        }
        return v;
    }

  private:
    static DacRep* exemplar;
};
```

Figure 4.12: Probing container relationships

The figure shows how D&C container relationships can be followed using a dynamic casting class. `get_this_ptr()` *tries inheritance relationships first before trying container relationships.*

one method or the other, would depend on the computational requirements for creating objects and the computation between object instantiations.

How might the latter method be implemented? The key is to remember that the execution stack is continuously being scanned by the scheduling process, regardless of whether a D&C pass is executing or not. The scheduler is stateless with respect to evaluation sequences, it simply evaluates available objects. Thus, instead of applying an evaluation function, such as `run()`, to a D&C object, we could simply push the object onto the execution stack. The scheduler will then whisk the object away for distributed evaluation. Once all objects have been created, we can pop any un-evaluated ones off the stack and complete the evaluation. If this evaluation is carried out using the stack, as for any normal D&C evaluation, then load-balance will continue to be maintained.

The only question that remains to be answered is how this can be implemented transparently, using the tools available to us. Pushing objects onto the stack is easily accomplished, however, we need some method of returning evaluated objects behind the back of the programmer. Since the desired scenario involves *evaluation* features, the natural place to put functionality is in a *wrapper*, `DacPar`, derived from `Dac`. We can arrange for the

```
class DacPar : public Dac {
  public:
    DacPar(const Dac& d) { init(d); }
    DacPar(DacRep* d) { init(Dac(d)); }
    const Dac& operator=(const Dac& d) { return init(d); }

    const DacRep* rep() const {                    // intercept attempts to get the letter
        if (head) runp();
        return Dac::rep();
    }

  protected:
    const Dac& init(const Dac& d) {
        if (d==0) ::error() << "init() failed:  zero dac object" << fatal;

        Record* r=::new Record(*this);                            // add object to list
        r->next = head;
        head = r;

        p->push(d, DacStack::pending, DacStack::parallel);           // add to stack
        return d;
    }

    static struct Record {                                            // object list
        Record(Dac& d) : dac(d) {}
        ~Record(){}
        Dac& dac;
        Record* next;
    } *head;

    void DacPar::runp() const
    {
        Record* d;                                 // iterate over all recorded objects
        while (p->key() == DacStack::parallel && head) {

            if (p->state() & (DacStack::evaluated|DacStack::fixed)) {
                head->dac = p->pop();                             // pop if evaluated ...
            } else {
                head->dac = p->pop().run();                              // run of not
            }

            d = head;
            head = d->next;
            ::delete d;
        }
    }
};
```

Figure 4.13: Parallel D&C wrappers

The figure shows a D&C wrapper class for evaluating D&C objects in parallel. If a `DacPar` *object is initialized with a* `Dac` *then the* `Dac` *is added to the list* `head` *and pushed onto the evaluation stack. When the* `Dac`*'s letter is accessed,* `runp()` *is invoked to ensure that the object has been evaluated.*

`DacPar` constructor to push its letter onto the evaluation stack. We also need some record of where the evaluated object needs to be returned to. Thus, we enter the *envelope's* address into a linked list. We arrange for `operator=` to evaluate any remaining un-evaluated objects, and then assign each result to the correct envelope as recorded in the linked list. We can also arrange for any attempt to access an envelope's letter, most notably use of `ptr_cast<T>`, to trigger evaluation. The class is shown in figure 4.13.

Catching accesses that might fail is roughly equivalent to the `rtf()` – return to future – mechanism of the Mentat system [44]. It is only by virtue of our object-oriented D&C system that dependency management can be accomplished easily and transparently. Specifically it is the separation of envelope and letter, evaluation and evaluated, that is key to transparency. The envelope / letter idiom is used here as a data-hiding mechanism [14]. All data-dependencies will be on the *letter*, which is hidden by encapsulation. Breaking encapsulation – done dynamically here – is caught and synchronizes evaluation state. If encapsulation is not violated then data dependencies will be secure.

4.3.2 Nested evaluation

In section 3.1.2 we described a stack-based implementation for D&C. The advantages of this approach were simplicity and space efficiency, however, there is another advantage. Stack-based evaluation has many similarities with normal computational execution, and thus we can recurse evaluations, nesting one on top of another.

We have argued [81] (section 2.2.1) that the design of efficient parallel algorithms should only be a means to an end, rather than an end in itself. However, many parallel implementations concentrate solely on the efficiency of a single algorithm without much thought as to the operation of this part in a wider whole. This is especially true of D&C implementations. We have argued that D&C gives us the means to integrate together efficient parallel algorithms; nested evaluation makes this more of a reality. By adopting a stack-based evaluation structure, it becomes possible to evaluate multiple D&C tasks at the same time, i.e. a D&C task can contain other, differing D&C tasks which will be evaluated in parallel as well as the main task.

The beauty of this system is that, although nested D&C tasks imply a finer grain of concurrency, they do not imply a performance loss. This is because the higher level, and therefore larger, tasks will be stacked first. The finer grain tasks will only be evaluated in parallel if there are no larger tasks available on the stack. In theory this should only happen at the very beginning and end of processing.

In case parallel evaluation is *not* required – in debugging D&C applications for example – it is easy to use the recursive implementation which is interchangeable with the stack-based algorithm. If necessary, it would be trivial to pass a flag to a D&C object when it is initialized, determining which algorithm to use. The only change that need to be made to the evaluation algorithm is to record stack start positions. This prevents separate evaluations from trespassing on each others data.

4.4 Further applications

In ensuing sections we look at some more complex examples of D&C problems. We examine the simplicity with which these problems can be expressed and implemented, and then go on to make use of some of the techniques we have discussed so far. Finally we examine run-time performance and changes we can make to improve this.

First of all we return to the horizontal back-propagation algorithm of section 2.3.

4.4.1 Back-propagation revisited

In section 2.3, we described informally a D&C implementation of the back-propagation algorithm. We now return to this and take a more detailed look at the implementation, using the techniques of chapter 3.

4.4.2 D&C implementation

The basis of the back-propagation algorithm is iterating over training frames calculating weight matrix updates. We thus choose a `zArray<T>` as our basic D&C structure. Ideally, we might want to use two `zArray<T>`s one for input frames and one for outputs. Unfortunately, if we were to do this then we would have to explicitly program sequential iteration. Instead we create a `zArray<T>` of training frames where a training frame is a vector of inputs and a vector of outputs. In order for these objects to be safely stored contiguously (see section 2.1.2) we use `SimpleArray<T>`'s. These structures are parameterized on the array size, and data is allocated in the structure itself rather than separately from the heap. The drawback with using these structures is that changing any array dimensions is a compile-time action.

```
template <class T>
class zMapArray : virtual public zContainer<BuiltinArray<T> >
{
  protected:
    zMapArray() {}
    virtual Dac mapfunc(const T&) const=0;

    Dac evaluate() const
    {
        BuiltinArray<T> b = data();
        Dac r = mapfunc(b[0]);                              // get the first result

        int elems = b.elems();                              // do the rest
        for (int i=1; i<elems; i++) {
            r = r.combine( mapfunc(b[i]) );
        }

        return r;
    }
};
```

Figure 4.14: A D&C array iterator

The figure shows a D&C class for iterating over an array. `mapfunc()` *is assumed to have been defined in a derived class and is called for every element of the array. The class is* `virtual`*ly derived from an array container so that only one data item per object exists.*

Thus our back-propagation object is derived from `zArray<TrainFrame<T>>`, `zPartition`, `zCombine` and `zMapArray<TrainFrame<T>>`. A `zMapArray<T>` is another member of the Beeblebrox library. It is a mix-in for `zArray<T>` derivatives, and provides `evaluate()`

```
template <class T, int I, int O> struct TrainFrame
{                                                          // Backprop training frames
    SimpleArray<T, I> inputs;
    SimpleArray<T, O> outputs;

    operator Vector<T>() const { return Vector<T>(I+O,(T*)(&inputs)); }
};

template <class T, int I, int O>
class zBackprop : public zArray< TrainFrame<T, I, O> >,
                  public zMapArray< TrainFrame<T, I, O> >,
                  public zPartition
{
    Matrix<T> W_ij;
    Matrix<T> W_jk;

  public:
    Dac mapfunc(const TrainFrame<T,I,O>& t) const
    {
        Vector<T> S_j, S_k, O_j, O_k, d_k;

        Matrix<T> d_Wij;
        Matrix<T> d_Wjk;

        Vector<T> t_k = t.outputs;
        Vector<T> O_i = t.inputs;
                                    // feed forward equations
        S_j = ~W_ij * O_i;
        O_j = sigmoid(S_j);

        S_k = ~W_jk * O_j;
        O_k = sigmoid(S_k);
                                    // feed backward of error equations
        d_k = (t_k - O_k) * dsigmoid(S_k);
        d_Wjk = O_j * ~d_k;
        d_Wij = O_i * ~((W_jk * d_k) * dsigmoid(S_j));

        return Dac( new zSum<Matrix<T> >(d_Wjk) ),
            Dac( new zSum<Matrix<T> >(d_Wij) );
    }
};
```

Figure 4.15: A D&C class for back-propagation

The figure shows a D&C class that computes the back-propagation algorithm in the iterator function `mapfunc()`*. The class is parameterized on a* `TrainFrame<T,I,O>` *class which is used for holding the training data.* `mapfunc()` *returns a compound D&C object which arranges for the partial weight update matrices to be added together.*

which is arranged to apply the member function `mapfunc()` to each element of the array. See figure 4.14. `zMapArray<T>` is unnecessary if division proceeds to vectors of unit length. However, if we wish to have a reasonably efficient implementation, then vectors of significant lengths must be processed. Thus it is that we use `zMapArray<T>` and `zPartition`.

All that remains is to define the core operations in `mapfunc()`. This simply applies the back-propagation algorithm to the training frame provided by `mapfunc()`'s argument. The result we are interested in is the *sum* across frames of the two weight matrices updates. We achieve this by constructing a `zCompound` from two `zSum<Matrix<T>>`s, and returning this from `mapfunc()`. `zSum<T>`s sum their formal parameters in `combine()`, section 5.3.3 describes this class in more detail. Note the use of `operator,` to combine two `Dac` objects into a compound. `dac-mode` is then used to fill in all remaining auxiliary functions. The resultant class is shown in figure 4.15.

4.5 Tree language models with D&C

In this section we describe the use of tree language models for speech recognition, and the implementation of one particular tree growing algorithm using object-oriented D&C. We then consider a parallel improvement to the algorithm using nested D&C. The following discussion is based on [108].

In the field of automatic speech recognition (ASR), language models, which attempt to provide an accurate prediction of the next word in a sequence, are important for good overall performance of any ASR system.

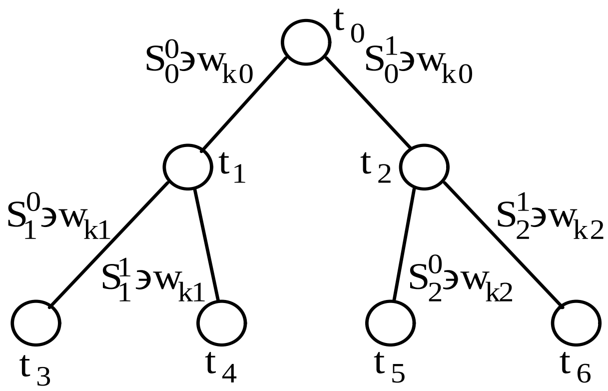


Figure 4.16: Tree

The diagram shows schematically a decision tree. The nodes are labelled t_i. At each node a decision is made as to which set of words the current word is in.

Decision trees (figure 4.16) are one possible type of language model [108]. A tree T may be viewed as a set of nodes $T = \{t_0, t_1, \ldots, t_n\}$, with t_0 reserved as the root node. The input to the tree $w_{j-n}, w_{j-n+1}, \ldots, w_{j-1}$ is a string of the previous n words, and the output from a leaf of the tree is a probability distribution over the possible predicted words $\hat{W}$. Starting from the root node of the tree, at each non-terminal node t_c, a binary function Q_c is performed on w_k, one of the n words of the input. This function takes the form: 'is $w_k \in S_c^j$?', where S_c^j is the jth set of words (for binary branching $0 \leq j < 2$). If the result is true, the left branch from the node is followed, otherwise the right branch. This is repeated until a terminal node is encountered.

In growing a tree, the objective lies in minimizing risk for a given cost constraint. Each step consists of splitting some terminal node t into two children in order to maximize a merit function, merit(.), for some test Q, which will lead to a tree which satisfies the global constraints.

Due to the computational complexity of growing optimal trees, practical design procedures (deciding on the best splits and hence the sets S_c) are invariably steepest-descent based. However, even practical approaches are extremely computationally intensive and a good candidate for parallel evaluation.

For the purposes of this work, a clustering-based algorithm due to Chou [28] and implemented serially by Waegner and Young [108] was employed. Vectors consisting of the conditional probabilities of the predicted words, conditioned on a particular instantiation of w_k, are iteratively clustered into bins. The clustering algorithm is classically D&C with *all* computation being done in the divide phase of the algorithm.

4.5.1 Parallel implementation of Chou's algorithm

The strategy we employed in implementing a parallel version of Chou's algorithm is important, as it reflects a general strategy for writing programs using object-oriented D&C. Our initial strategy entailed the following:

1. Identify how the algorithm is "divide-and-conquer-able".
2. Identify data that child nodes require from their parents and vice versa and encapsulate these in an object definition (class) derived from `DacRep`.
3. Identify data required by all nodes and add these to the class as static members.
4. Add the required primary D&C functions to the class definition through: inheritance if they are standard, or by writing them if they are not.
5. Format the class in the emacs editor with `dac-mode` so that objects of that class are usable in a message passing environment.
6. Define in `sti()` any initial conditions required.

Our original aim was to use much of the serial code, written in C, as it stood. Much of this code was concerned with actual D&C evaluation, so this could be thrown out in favour of our parallel framework; leaving only the guts of the partitioning algorithm itself. This, we hoped, could be left as a C module that only required linking in to the parallel framework. For an initial attempt this proved possible – if slightly clumsy – by defining appropriate conversion operators from C++-world array objects and the like to C-world arrays. However, when it came to more complicated implementations, it was found far easier to recompile the C-code with a C++ compiler, thus giving the C-module easy access to all the C++-world data structures. Although this approach does not work with programs for which the source is not available, it is vastly preferable from a flexibility point-of-view.

The clustering algorithm made up the flesh of the `divide()` phase of the algorithm and in terms of writing functionality there was little else to do. Since we are *growing* these trees and therefore have no interest in combining anything, `evaluate()` was made to return a `zNull` object. This object did nothing except satisfy the requirements for successful completion of the evaluation algorithm.

4.5.1.1 Tree dumping

One other important issue needs considering. Data is generated from each non-terminal and terminal node in the tree, but how can we guarantee that the tree constructed on disk is ordered correctly? The data cannot be written to a single file as there is no way to tell the size or shape of a tree before evaluation – it is an unbalanced D&C computation. The only obvious solution is to write a separate file for each sub-task evaluated. Several methods were tried of naming these files so that their order could be determined easily, but this required preorder node-number information which would have been available under a serial implementation, but which was indeterminate under the parallel implementation.

In the end the nodes were numbered in level order using `node()` - a numbering scheme which is independent of the size of the tree. The output files were identified by the root node that was processed in a sub-task, and then the files were parsed using Sedgewick's stack-based pre-order tree-traversal algorithm in order to discover their concatenation order.

Since tree growing is more generally applicable to D&C than just for language modelling, this technique should be more widely applicable.

The speedup results for the initial implementation are given in section 4.6.

4.5.2 Improved parallel performance using nested D&C

In this particular tree-growing application, at each non-terminal node we are trying to partition a corpus into two. The partitioning algorithm is of time complexity $\mathcal{O}(N)$. So it can be seen that at the start of evaluation, when there is a single large context, the (serial) partitioning of this will dominate the execution time of the whole. At the end of execution the partitioning is of many small contexts but these are done in parallel.

Thus it would be desirable to parallelize the initial partitioning somewhat to achieve better processor utilization. The partitioning algorithm involves looping over the entire corpus for each member of a given context. Thus, if we were to parallelize across the context elements we should be able to improve the performance of the partitioning by a factor close to the number of elements in a context.

4.5.2.1 Nested Implementation

In designing this extra level of parallelism we adopt an approach similar to that given in section 4.5.1. However, this time we are merely iterating over an integer so a general D&C iterator can be used to do much of the programming leg-work, see figure 4.17. From a functionality point-of-view we merely have to separate the loop and the partitioning algorithm, and assign each to a class.

At this point we can make use of inheritance to simplify the class definition. The tight coupling of the outer loop with the partitioning algorithm also implies a tight coupling of data, and we can thus factor the data common to both control structures into a single base class. It is worth considering this property for a moment, as it is an important one for our object-oriented approach. In general, once a D&C strategy has been decided for a particular problem, then the next candidates for parallelism are any outer loops in the D&C primary functions. However, the only required additions to the locality of reference

```
class zIterator : public DacRep, public zUnit {
 public:
   zIterator(int sa=0, int stop=0)
       : DacRep(elems), start(sa), elems(stop-start+1) {}

   Dac zIterator::iterate(int sa, int stop)
   {
       start = sa;
       elems = stop-start+1;

       return run();
   }

 protected:
   Dac divide(Dac& l, Dac& r) const
   {
       l = make();
       r = make();

       zIterator* il = ptr_cast<zIterator>(l);
       zIterator* ir = ptr_cast<zIterator>(r);

       il->elems = elems / 2;
       ir->start = start + elems / 2;
       ir->elems = elems - elems / 2;

       return 0;
   }

   Dac evaluate() const
   {
       return mapfunc(start);
   }

   virtual Dac mapfunc(int) const=0;          // defined by derived classes

   int start;
   int elems;
};
```

Figure 4.17: A D&C iterator class

The figure shows a D&C class for general iteration. Like `zMapArray<T> mapfunc()` *is assumed to have been defined by a derived class, and is called in* `evaluate()` *with each value of the iteration. Iteration is started by the function* `iterate()` *which is initialized with the iteration bounds.*

already provided by the D&C partitioning, will be data local to the D&C primary functions. Thus, the factoring together of common data will be a common occurrence in making use of nested D&C, and it is only by virtue of the object-oriented approach that this is easily achieved.

The fact that the parallelized looping as a sub-task of the parallelized clustering is coped with is because of the generality of the stack-based approach described in section 3.1.2.

A comparison of the performance of the two approaches is given in section 4.6.

4.6 Performance of the system

In this section we give performance results for the simple and nested implementations of the tree-growing algorithm. We then consider some implications of these results and give some improved results through an understanding of D&C systems in general.

In the ensuing discussion it is important to realize that tree-growing is a true D&C application, in other words the appropriate method of serial evaluation is D&C. For this reason there is no need to compare the parallel implementation with a control as the uniprocessor case is the same, performance-wise, as the serial case; neglecting processor usage due to scheduling. It is also important to realize that partition size [88] is an integral part of the algorithm. Thus finding an optimal partition for parallel evaluation must be entirely separate from specifying a D&C partition.[1]n this and subsequent sections, a balanced tree is assumed. Whilst not true in general, this assumption is believed to be valid for the data sets used here.

4.6.1 Non-nested implementation

Figure 4.18 gives the speedup results for the non-nested implementation using a small corpus of 16384 words (16376 contexts) and a cluster size of 64 words. The experiments were conducted on a toroidal mesh of T800 transputers running the Trollius^{TM} operating system.

As can be seen performance is good though, as noted in [81], there are some discontinuities in the curve. As we explained in section 2.4.1.1 these discontinuities can be attributed to the connectivity of a transputer node. The depth-first evaluation scheme that we employ means that the size of task offloaded to neighbouring processors decreases in size as $1/2^n$, where n is the task number. So the first processor receives 1/2 the problem, the second 1/4 and so on. However, once the nearest neighbours have been exhausted due to the *single-steal* rule [72], then the speedup is limited to some extent by the size of problem remaining on the root processor. Thus we would expect the speedup to be limited to $2^{n_{links}}$ – however many processors there are.

This is borne out by figure 4.18. The processor numbering scheme we have adopted means that there is only one link available for 1-4 processors, two for 5-12 processors and three for 12-16 processors. The breakpoints for the number of links correspond to the discrepancies in the graph. The sudden drop in performance after each breakpoint can be attributed to a large proportion of the problem being scheduled to a solitary processor. This processor is connected to the newly available link with no other neighbours to offload to.

[1] I

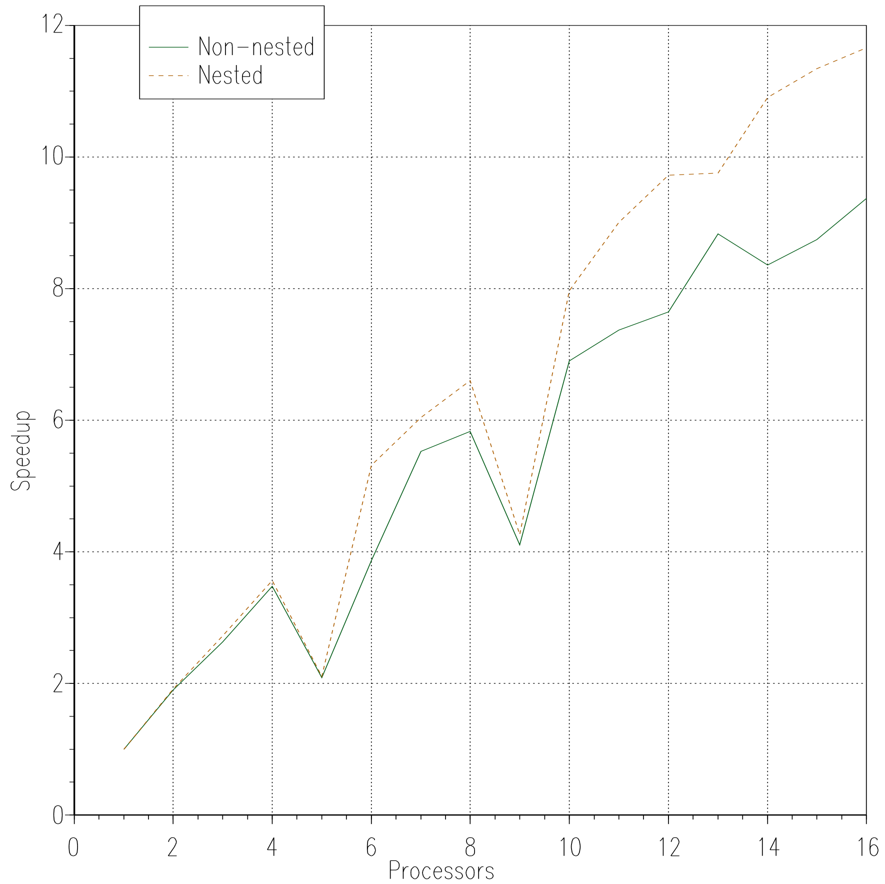


Figure 4.18: Nested and non-nested performance

The diagram compares the nested and non-nested performances of a D&C based language model. As can be seen, nesting gives significant performance improvement.

However, the results are inconclusive and we will present more compelling evidence below.

4.6.2 Nested implementation

In figure 4.18 the speedup results for the nested implementation are presented, together with the non-nested implementation. As can be seen, the nested implementation gives a significant improvement over the simple implementation. We would also expect this improvement to increase as the problem size is increased and parallel overheads become less significant. However, we would expect the maximum performance increase to have an upper bound equal to the maximum speedup given by the nested performance only – in this case 8.

4.6.3 Performance improvements

We have surmised that the performance of a depth-first D&C system will be limited by the connectivity of the processors, and we have provided some scanty evidence for this. Therefore, for a 4-link processor, we would expect the performance to be largely unaffected by an increase in the number of processors past 16, if the connectivity of the processors remained unchanged. If we increase the number of processors from 16 to 32, we obtain the results given in figure 4.19.

Obviously the system as it stands is not at all scalable, and scalability is highly desirable. However, if our connectivity argument is right then we can achieve better scalability by utilising a processor network with a higher degree of connectivity. Specifically, if we require each processor to have access to a larger number of neighbours, then a hypercube architecture is the obvious candidate. Furthermore, if speedup is limited to $2^{n_{links}}$ and speedup is also limited to $n_{processors}$ then we will attain maximum speedup for:

$$\begin{aligned} 2^{n_{links}} &= n_{processors} \\ n_{links} &= log_2(n_{processors}) \end{aligned}$$

which is true for all hypercubes.

4.6.3.1 Virtual hypercubes

It is clear that a hypercube interconnection network is desirable for D&C systems, but a transputer only has four links! Ideally we would use a network of TI C40's which would yield a physical hypercube of maximum degree 6; but is this really necessary? We can achieve higher dimensionality hypercubes by forming groups of physically connected hypercubes and connecting these to each other by means of "virtual" – or multi-hop – links. Arranging for this to happen is simple if the real hypercubes are numbered in units of 16. In this case a processor will have virtual neighbours at processor identifiers:

$$(procid + 2^{i+4})\ MOD\ 2^{i+5}$$

where $i = 0, 1, \ldots$.

Applying this to the 32 processor case yields the results given by figure 4.20. As can be seen the speedup is now much more linear with a greater maximum speedup.

Parallel overheads could possibly account for the decreasing efficiency. If this is the case then increasing the problem size should result in increased efficiency.

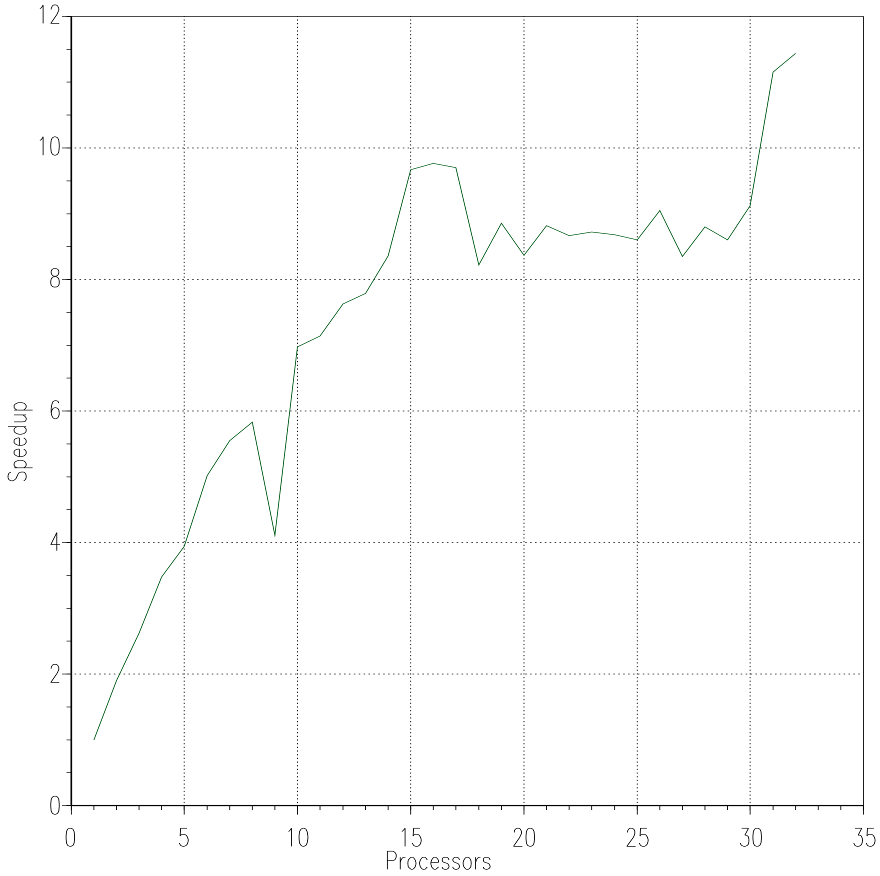


Figure 4.19: 32 processors non-nested performance

The diagram shows how performance is severely impaired for more than 16 mesh connected processors. We can see no improvement between 16 and 30 processors, and it is only at 31 processors, when all 4 links of the root processor are used, that performance reaches that of the 16 processor implementation.

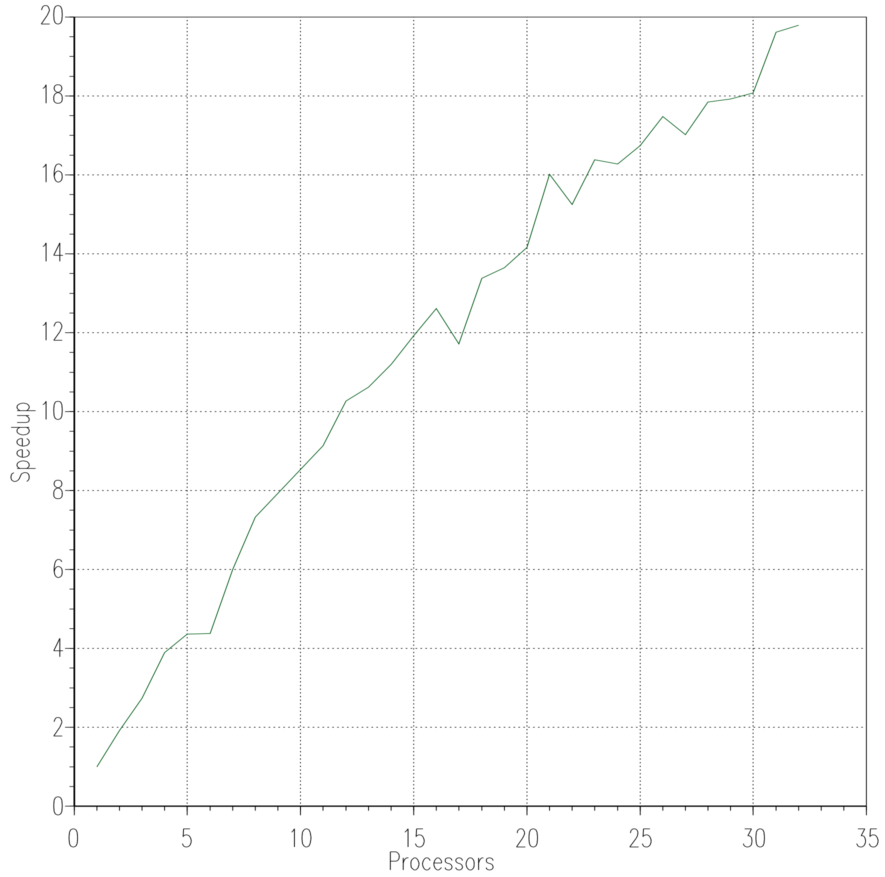


Figure 4.20: Virtual hypercube performance

The diagram shows the performance for a 32 processor implementation of the D&C language model, but this time the processors are hypercube connected. Performance is now relatively scalable with a continuous improvement in performance as more processors are added. It is interesting to note that performance appears to be largely unaffected by the fact that some of the processor connections are multi-hop.

4.6.3.2 Problem scaling to reduce overheads

By increasing the size of problem we obtain the results given by figure 4.21.

As can be seen, increasing the problem size *does* increase the efficiency of the system. However, a corpus of 100000 words was the largest problem that could be tried with the memory available, and it is not clear that the maximum efficiency has been obtained with a problem of this size.

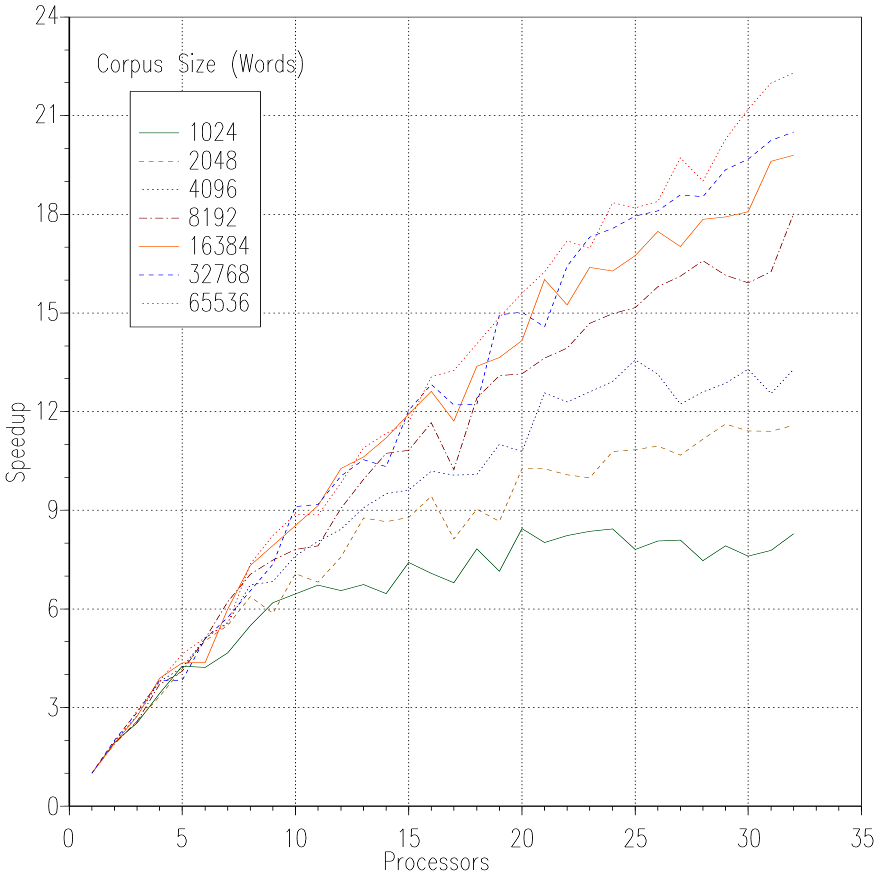


Figure 4.21: Virtual hypercube performance with increasing problem size

The diagram shows how relative performance can be improved by increasing the size of the evaluated problem. As expected the asymptote for this improvement is that of linear speedup; although it would take an infinitely large problem to achieve this.

Interestingly enough, adding a further dimension to the hypercube (figure 4.22) for the 100000 word case yields little further speedup, performance actually decreasing after 46 processors. It would therefore appear that tuning the D&C partition would be a sensible thing to do, to limit the number of processors used, as well as evaluating still larger problems.

4.7 Theoretical speedup

In this section we present some theoretical speedup results for D&C and relate them to the practical results presented in section 4.6. Lewis et al [69] gave the maximum possible speedup for D&C problems as $N/log_2 N$. However, our results exhibit rather better performance than this for an algorithm with classical D&C properties. Therefore, we develop a better model of D&C which takes into account the effects of problem scaling.

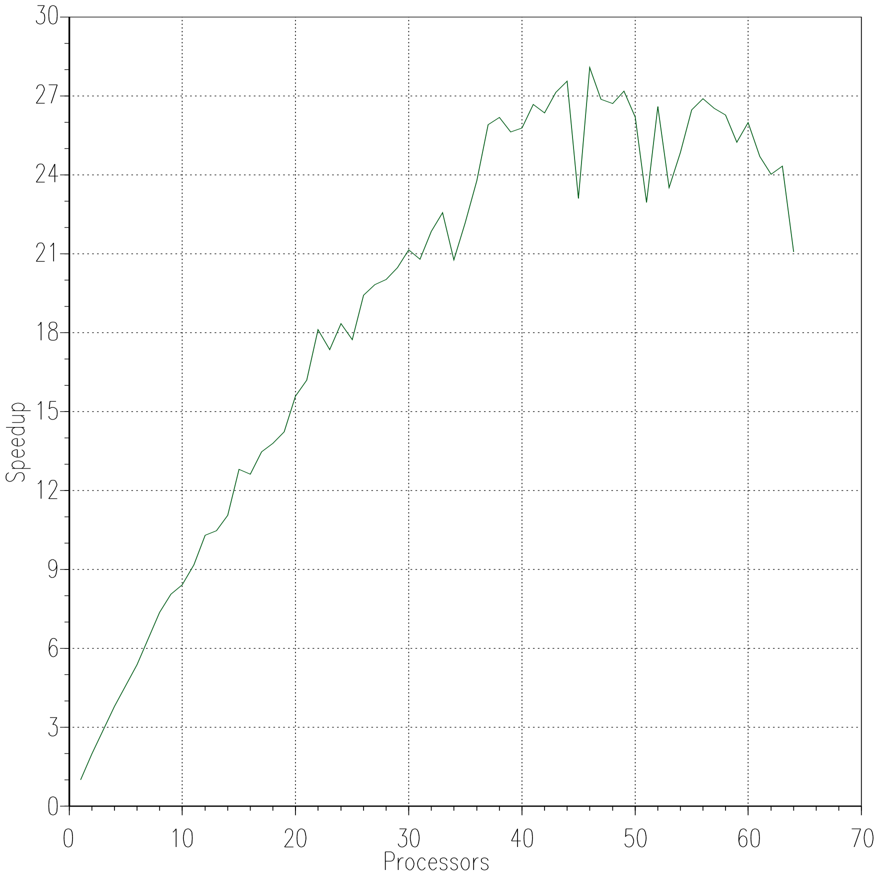


Figure 4.22: Degree 6 virtual hypercube performance

The diagram shows the performance of a 64 processor, degree 6 hypercube implementation of the D&C language model. Performance is good up to about 45 processors, and then tails off rapidly. We surmise that this is due to a combination of the size of the problem involved, and the number of multi-hop connections being used by the virtual hypercube.

These results tie up well with the practical results of section 4.6. We then develop a more complex model which takes into account the possibility of variable divide time; a property which the tree algorithm should exhibit. However, the theoretical results are much worse than our experimental results and so we conclude that this property is not significant for the tree algorithm.

4.7.0.3 Framework

In the execution of a D&C problem, we define $d = 1 \ldots D$ as the depth down the D&C tree so that there are:

$$n_D = 2^D \tag{4.1}$$

terminal nodes in the tree if the tree is balanced. This also means that there are:

$$N = 2^{D+1} - 1 \tag{4.2}$$

nodes in the tree in total, and

$$\begin{aligned} 2^D - 1 &= n_D - 1 &(4.3)\\ &= n_t &(4.4)\end{aligned}$$

non-terminal nodes. We note also that the maximum depth:

$$D = log_2(n_D) \tag{4.5}$$

We define:

$$\begin{aligned} t_d &= \text{time to perform a single divide}\\ t_c &= \text{time to perform a single combine}\\ t_f &= \text{time to perform a single evaluation}\end{aligned}$$

but we will often assume that $t_d = t_c$.

We assume that communication time is zero and that there is no limit to the availability of processors.

4.7.1 Basic D&C

We first investigate a simple theoretical model of D&C, where the divide time dominates. This yields the result given by Lewis et al. [69].

We note that it is not possible to make use of more than n_D processors. Thus if we wish to evaluate all nodes in parallel the execution time will be:

$$\begin{aligned} T(n_D) &= Dt_d + t_f + Dt_c\\ &= 2Dt_d + t_f\\ &= 2t_d log_2(n_D) + t_f &(4.6)\end{aligned}$$

if $t_d = t_c$. This is because the time to evaluate all nodes at depth $= k$ in parallel will be t_d, and all nodes at depth k cannot be evaluated before nodes at depth $k-1$.

The serial execution time for this problem is given by:

$$\begin{aligned} T(1) &= n_t t_d + n_t t_c + n_D t_f\\ &= (n_D - 1)(t_d + t_c) + n_D t_f\\ &= 2t_d(n_D - 1) + n_D t_f &(4.7)\end{aligned}$$

thus the speedup,

$$\begin{aligned} S &= T(1)/T(N)\\ &= \frac{2t_d(n_D - 1) + n_D t_f}{2t_d log_2(n_D) + t_f} &(4.8)\end{aligned}$$

We note that the speedup is n_D for the best case of $t_d = 0$. The worst case is for small t_f, which gives:

$$\begin{aligned} \lim_{t_f \to 0} S &= \frac{2t_d(n_D - 1)}{2t_d log_2(n_D)}\\ &= \frac{n_D - 1}{log_2(n_D)} &(4.9)\end{aligned}$$

which is that given by Lewis et al. [69].

4.7.2 D&C with problem scaling

The result obtained above obviously scales very badly with increasing numbers of processors. However, as demonstrated by Gustaffason et al. [52], the sensible thing to do is to scale the *problem* relative to the number of processors. Thus, we develop a theoretical model based on this premise.

If we scale the problem relative to the number of processors then we can view the execution on $n_p = 2^p$ processors as a purely parallel evaluation up to depth p, and a serial evaluation of n_D/n_p nodes.

In this case the serial execution time is the same as above, but the parallel execution time is given by:

$$\begin{aligned} T(N) &= 2t_d log_2(n_p) + \frac{n_D}{n_p} t_f + 2t_d \left(\frac{n_D}{n_p} - 1 \right) \\ &= 2t_d log_2(n_p) + \frac{n_D}{n_p}(t_f + 2t_d) - 2t_d \end{aligned} \tag{4.10}$$

so that the speedup is:

$$S = \frac{2t_d(n_D - 1) + n_D t_f}{2t_d log_2(n_p) + \frac{n_D}{n_p}(t_f + 2t_d) - 2t_d} \tag{4.11}$$

If we introduce a *scaling ratio*:

$$\alpha = \frac{n_D}{n_p} \tag{4.12}$$

$$n_D = \alpha n_p \tag{4.13}$$

then the speedup becomes:

$$S = \frac{2t_d(\alpha n_p - 1) + \alpha n_p t_f}{2t_d log_2(n_p) + \alpha(t_f + 2t_d) - 2t_d} \tag{4.14}$$

$$\begin{aligned} \lim_{t_f \to 0} S &= \frac{2t_d(\alpha n_p - 1)}{2t_d log_2(n_p) + \alpha 2t_d - 2t_d} \\ &= \frac{\alpha n_p - 1}{log_2(n_p) + \alpha - 1} \end{aligned} \tag{4.15}$$

Thus by increasing α we can produce a more linear speedup than the simple D&C case of section 4.7.1. Figure 4.23 shows theoretical speedups for varying α. These results tie in nicely with the practical results presented in figure 4.21. The case $\alpha = 1$ is equivalent to the simple case, given above. It can be seen, therefore, that problem scaling is a good thing to do with D&C systems, as this yields more efficient speedups.

Note that these are worst case results, in practice $t_f \neq 0$ and therefore the $n_p t_f$ term will be significant, increasing the linearity of the speedup.

Intuitively this is the result we would expect; for the case $\alpha = 1$ processor utilization is only at a maximum when nodes at maximum depth are being evaluated. However, if $\alpha > 1$ then processor utilization is at a maximum from depth p onwards.

4.7.3 D&C with variable divide time

The tree algorithm presented in section 4.5 processes a context of ever decreasing size. The algorithm involves looping over each member of the context; so we would expect the iteration time - the divide time – to decrease as the algorithm progresses. We now analyse the theoretical speedup expected from systems with this property.

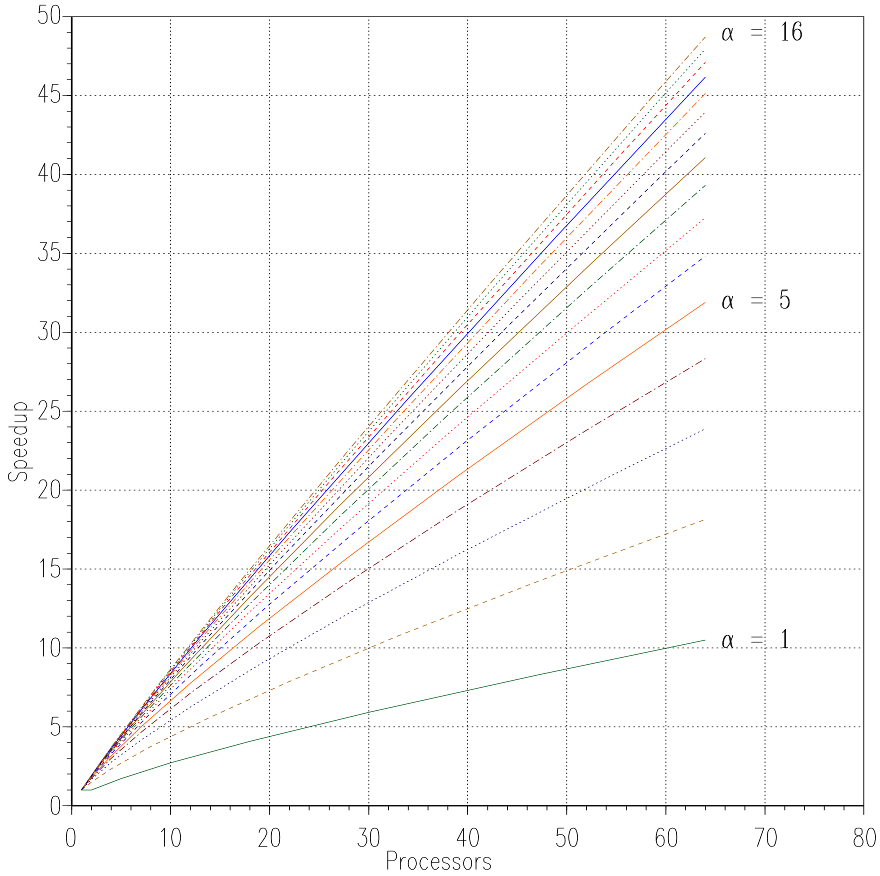


Figure 4.23: Performance with varying scaling ratio

The diagram shows the theoretical performance improvement from increasing the relative size of the problem. It demonstrates that if the number of D&C nodes is only an order of magnitude larger than the number of processors then performance is significantly increased. The ratio of D&C nodes to processors is represented by α.

4.7.3.1 Basic algorithm

The divide time should vary as $1/n_d$ where n_d is the number of nodes at depth d. If we therefore take t_d as the maximum divide time, then the actual divide time will be t_d/n_d. Thus the serial execution time for this case will be:

$$\begin{aligned} T(1) &= \left[\sum_{i=0}^{D-1} \frac{t_d 2^i}{2^i} + \sum_{i=0}^{D-1} \frac{t_c 2^i}{2^i}\right] + 2^D t_f \\ &= D(t_d + t_c) + 2^D t_f \\ &= 2t_d log_2(n_d) + n_d t_f \qquad (4.16) \end{aligned}$$

and the parallel execution time will be:

$$\begin{aligned}
T(N) &= \left[\sum_{i=0}^{D-1} \frac{t_d}{2^i} + \sum_{i=0}^{D-1} \frac{t_c}{2^i}\right] + t_f \\
&= 2t_d \sum_{i=0}^{D-1} \frac{1}{2^i} + t_f \\
&= 2t_d \left(\frac{1 - 1/2^D}{1 - 1/2}\right) + t_f \\
&= 4t_d(1 - 1/n_d) + t_f \qquad (4.17)
\end{aligned}$$

So the speedup will be:

$$\begin{aligned}
S &= T(1)/T(N) \\
&= \frac{2t_d log_2(n_d) + n_d t_f}{4t_d(1 - 1/n_d) + t_f} \\
\lim_{t_f \to 0} S &= \frac{log_2(n_d)}{2(1 - 1/n_d)} \qquad (4.18)
\end{aligned}$$

This is a very poor result, since for large n_d the speedup is $log_2(n_d)$. Thus, as previously, we must look at variable divide time with problem scaling.

4.7.3.2 Variable divide time with problem scaling

In this instance we can again view the parallel execution as purely parallel up to n_p nodes and then serial for n_d/n_p nodes, ignoring any t_f terms at depth n_p since they are not applicable. We note also that the maximum divide time at depth p will be t_d/n_p. Thus:

$$\begin{aligned}
T(N) &= 4t_d(1 - 1/n_p) + 2\frac{t_d}{n_p} log_2(n_d/n_p) + \frac{n_d}{n_p} t_f \\
&= 4t_d(1 - 1/n_p) + 2\frac{t_d}{n_p} log_2(\alpha) + \alpha t_f \\
&= 2\frac{t_d}{n_p}\left(log_2(\alpha) - 2\right) + 4t_d + \alpha t_f \qquad (4.19)
\end{aligned}$$

The serial execution time is the same as above so the speedup is:

$$\begin{aligned}
S &= T(1)/T(N) \\
&= \frac{2t_d log_2(n_d) + n_d t_f}{2\frac{t_d}{n_p}\left(log_2(\alpha) - 2\right) + 4t_d + \alpha t_f} \\
&= \frac{2t_d log_2(\alpha n_p) + \alpha n_p t_f}{2\frac{t_d}{n_p}\left(log_2(\alpha) - 2\right) + 4t_d + \alpha t_f} \qquad (4.20) \\
\lim_{t_f \to 0} S &= \frac{2t_d log_2(\alpha n_p)}{2\frac{t_d}{n_p}(log_2(\alpha) - 2) + 4t_d} \\
&= \frac{n_p log_2(\alpha n_p)}{log_2(\alpha) - 2 + 2n_p} \qquad (4.21)
\end{aligned}$$

Again increasing α will yield better performance but not nearly as good as that for the fixed divide time case. In practice the t_f terms will be significant and performance will be better.

4.7.4 Application to the tree implementation

The tree growing implementation presented in section 4.5 is one in which the divide time *is* variable. However, the results presented in section 4.6 are considerably better than those indicated by equation 4.21. This is partially due to the nested implementation but also must be due to the fact that $t_f \neq 0$ and that there must be some constant element in the divide time. Presumably this constant element will be less dominant for larger problems so the effect of increasing the scaling ratio will become less effective for larger problems. For the largest problem tried:

$$\begin{aligned} \alpha &= n_d/n_p \\ &= \frac{100000/64}{64} \\ &= 24.41 \end{aligned}$$

Thus for a fixed divide-time and $t_f = 0$ the maximum expected speedup on 64 processors is:

$$\begin{aligned} \lim_{t_f \to 0} S &= \frac{24.41 \times 64 - 1}{log_2 64 + 24.41 - 1} \\ &= 53.09 \end{aligned}$$

and for the variable divide time case:

$$\begin{aligned} \lim_{t_f \to 0} S &= \frac{64 log_2(24.41 \times 64)}{log_2(24.41) - 2 + 2 \times 64} \\ &= 5.20 \end{aligned}$$

Which is far less than the results actually obtained (figure 4.22). So obviously the variable divide time effect cannot be particularly significant for this particular problem.

We may conclude then that D&C can provide much better speedups than that presented by Lewis et al [69], if the problem is scaled relative to the number of processors. However, we note that variable divide time leads to very poor performance even if the problem is scaled. Fortunately, few problems will *only* display properties of variable divide time.

4.8 Summary

We have described some Beeblebrox library classes which yield a degree of programming convenience. We have also described classes which yield improved parallel performance. In particular we have demonstrated the performance improvement given by nested D&C evaluations. We have examined the D&C algorithm theoretically, and demonstrated that problem scaling and hypercube connectivity increase parallel performance.

Chapter 5

The Kanerva model

Every kingdom divided against itself will be ruined. *Matthew 12*

So far we have mainly discussed problems that have a direct mapping to D&C, and with a fairly implementation-oriented perspective. We have seen the limitations of D&C as far as speedup is concerned and also discussed some improvements performance-wise. However, our object-oriented environment provides features on which we wish to capitalize. It is to this that we now turn.

We need to address two fundamental concerns:

- Can we allow an application programmer to statically and, more importantly, dynamically manipulate D&C objects to give, transparently, added functionality and speed?
- Can we ensure such manipulations make efficient use of system resources where possible?

We start with a problem, for without a real world situation it is difficult to envisage where potential gains might be made.

5.0.1 Chapter organization

In section 5.1 we discuss the Kanerva sparse, distributed memory model, and in section 5.2 its parallel implementation. In section 5.3 we look at the D&C features required to implement this model using delayed evaluation. In section 5.4 we look at refinements that can be made to the logical structure of these features, and in section 5.5 we look at refinements that can be made to the evaluation structure of these features. In section 5.6 we describe how these features can be presented to the programmer in a seamless fashion. In section 5.7 we look at some automatic partitioning optimizations. Equipped with these features we then describe the Kanerva implementation in section 5.8. In section 5.9 we look at the performance figures for this implementation. Finally in section 5.10 we look at the optimization of expressions constructed using delayed evaluation.

5.1 The Kanerva memory model

Kanerva derived a memory model by considering what form a neural network should take in order to exhibit the functionality of human memory [64]. He demonstrated that his design would have interesting pattern matching properties in a sufficiently large implementation. Little practical work has been done on this model because Kanerva's theory predicted that its most powerful properties would only be exhibited in implementations far too big to be simulated on serial computers. Kanerva's model has been subsequently modified to try and achieve useful results for smaller implementations [85], however his original results are still superior for the right size of implementation.

5.1.1 Kanerva's memory theory

The Kanerva model consists of three layers: the input unit layer, the location unit layer and the output layer. Each location unit has a fixed, random, binary address with the same characteristics as the input pattern. The locations have outputs described below and these outputs are connected to the output units by a conventional mesh of links holding bias values.

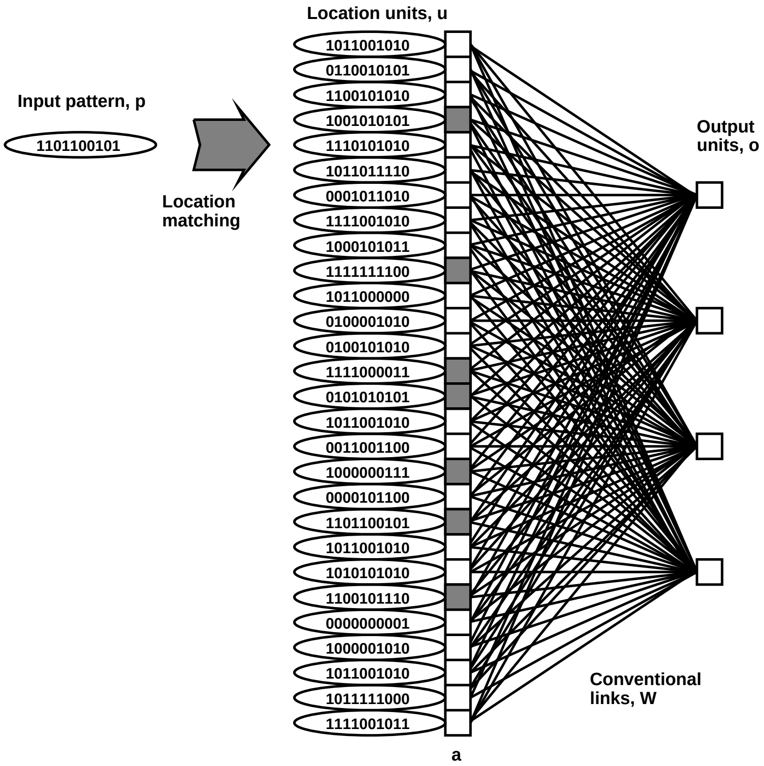
In order to store a pattern, each of the location units' addresses are compared with the input pattern. If the Hamming distance between the two is less than the activation radius, then the location is considered to be active and its output is set to 1. If not then the output is set to 0. The activation radius is a parameter of the model. The desired output pattern is presented at the outputs and each link from an active location unit to an output location is examined. If the output is 1 then the corresponding link bias is incremented by 1. If the output is 0 then the corresponding link bias is decremented by 1. The pattern is then considered stored. More sophisticated approaches to storage have been found [85], but the simple storage algorithm given above is sufficient for large enough models.

In order to retrieve a pattern, the active locations are determined as for storage, but this time the outputs are generated by multiplying the vector of active locations by the matrix of link biases. The resulting output vector is adjusted so that values greater than 0 are made 1 and values less than or equal to 0 are made 0. See Figure 5.1.

Kanerva showed that large-scale models would be useful for pattern recognition in corrupt data, because of the properties of binary spaces with many dimensions. For any point in the space, if the dimensionality n is high, the distribution of the distances to other points in the space is such that the vast majority are a Hamming distance $n/2$ away, and few are closer or more distant than this. For larger n this becomes even more pronounced. As a consequence of this, the overlap between those locations activated by a pattern and those activated by a corrupted version of the same pattern is considerable. However, for the model to be able to store a significant number of patterns and for these patterns to be retrieved reliably, Kanerva showed that the pattern space needed to be at least 10^3 bits and the number of location units to be at least 10^6. If we desire 1000 outputs this leads to a storage requirement of roughly 2Gb - generally too large by modern computer standards. Additionally, finding the Hamming distance between the input pattern and all the location units is a computationally intensive task requiring 10^9 operations. These two features make the Kanerva memory model an attractive proposition for a parallel implementation.

5.2 Towards a parallel implementation

The first, and possibly most obvious, feature of the Kanerva model is that it is an algorithm-parallel rather than data-parallel problem. Unlike back-propagation there are



$$a_i = \begin{cases} 1 & \text{if } d_h(u_i, p) < k_r \\ 0 & \text{otherwise} \end{cases}$$

Retrieval:

$$\mathbf{o} = f(\mathbf{Wa}), \quad f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Storage:

$$\mathbf{W} = \mathbf{W} + \mathbf{o}_d \mathbf{a}^T$$

Figure 5.1: The Kanerva model

The diagram shows schematically the Kanerva memory model. The input pattern p *is compared with the location units* u*, and the active locations* a *determined. The outputs are then given by the product of the active locations and the weight matrix* W*, normalized to 0 or 1. For storage the weight matrix is updated by the product of the desired outputs* o_d*, normalized to -1 or +1, and the active locations.*

not large numbers of training frames needing to be presented to the model, and the model is not small enough, in its full incarnation, to be accommodated by a single processor. Thus we are naturally drawn back to reconsider delayed evaluation as an implementing technology for this algorithm.

The second consideration is also related to size. If a full implementation of the Kanerva model is of the order or 2Gb then it is unlikely to fit on a 128Mb transputer array, which is the hardware we have available. Instead, we are drawn to a readily available technology which does have the resources we require, namely that of workstation clusters. However, in using this technology we must bear in mind the significantly different communication / computation characteristics that modern workstations have to transputers.

5.2.1 Cluster computing

Cluster computing is becoming a common feature of parallel processing systems. Many parallel processing systems claim to run on clusters of workstations [48], and some parallel systems are specifically designed for workstation clusters [41]. Cluster computing is attractive because of the ready availability of the hardware. Any network of computers is potentially a parallel machine and cluster computing aims to make it so. Trollius^{TM}' design is such that a cluster computing implementation is easily achieved and has been done so in Trollius LAM^{TM}. In fact $Trollius^{TM}$ has been designed as a heterogeneous operating system [16]. Since our D&C system is designed to sit portably on top of Trollius^{TM}, it is a simple matter to use the system on a network of workstations. The only additional concerns are memory management, and where the network might be heterogeneous in nature and issues such as byte ordering become a problem.

In the light of this let us look at the equations relevant to the Kanerva model.

5.2.2 Kanerva maths

If we define a function $d_h(a_1, a_2)$ which finds the hamming distance between a_1 and a_2, and a mapping function:

$$\begin{aligned} map:\ \{\mathbf{v}, f\} &\rightarrow \mathbf{u}\ WHERE \\ &\mathbf{u} = \{f(v_1), \ldots, f(v_n)\} \end{aligned} \tag{5.1}$$

Then the vector of active locations $\mathbf{a}$ is:

$$\mathbf{a} = map(map(\mathbf{u}, d_h), < r_k) \tag{5.2}$$

where $\mathbf{u}$ is the vector of location units, and r_k is the activation radius. The weight updates are then:

$$\Delta\mathbf{W} = (2\mathbf{o_d} - 1)\mathbf{a}^T \tag{5.3}$$

$$\mathbf{W} = \mathbf{W} + \Delta\mathbf{W} \tag{5.4}$$

the adjustment of $\mathbf{o_d}$, the desired outputs, being to -1, +1 rather than 0, 1.

In order to retrieve a value from the model, the outputs $\mathbf{o}$ are given by:

$$\mathbf{o} = \mathbf{W}\ map(map(\mathbf{u}, d_h), < r_k) \tag{5.5}$$

with $\mathbf{o}$ bounded to the range 0,1.

These equations are relatively simple, but the fact we must bear in mind is the size of the vectors we are dealing with. Because these are so large, we have a very different problem to that of back propagation. Evaluating these equations will involve a major computational effort, and anything we can do to reduce the level of computation by increasing the parallelism would be beneficial. In the next section we will examine some features we can introduce to the Beeblebrox class library to improve the efficiency of problems of this type.

5.3 Features

In examining the Kanerva model we are looking at a problem that can be represented simply in mathematical notation. In adding features to the Beeblebrox system we are trying to do so in a way that matches the generality of the mathematics, rather than the specifics of the Kanerva model. However, it is likely that any initial implementation of these features will contain a certain bias towards the Kanerva model.

5.3.1 Delayed evaluation

Delayed evaluation, as described in section 2.2.3, must form the basis of our parallel implementation. It is obvious that an implementation that evaluates *map* functions individually over a large array is going to be enormously inefficient. Thus we must ensure that operation sequences are stored before evaluation. Examining the Kanerva equations, it would appear that we need objects to represent *map*, vector multiplication, matrix-vector multiplication and matrix addition. However, blindly defining these objects takes no account of the parallel environment in which we are operating. Delayed evaluation seeks to take advantage of the decomposed state of extant data, but it will only work if the delayed evaluation objects match this state to the method of evaluation.

For instance, if we multiply two vectors to yield a matrix, it is possible to do this in two ways. These two ways depend on whether the left or right vectors are distributed across the processor network, If the left is distributed then the distributed result will be that of row vectors that need to be concatenated on top of one another. If the right is distributed, then the distributed result will be that of column vectors which need to be concatenated next to each other. Either way is possible, except that if the resulting matrix needs to be added to another distributed matrix, then the partitioning of both must match if delayed evaluation is to be employed. See figure 5.2. If we then remember that vectors can be partitioned two ways – partitioned or not at all – and matrices can be partitioned four ways – column-wise, row-wise, block-wise or not at all – then the number of possible classes appears to explode to match the possible combinations of these two data types!

5.3.2 Polymorphic operands

As outlined above, the differing partitioning possibilities for D&C objects of the same HAS-A type could lead to a multitude of operation type classes. These classes would represent the union of all possible combinations. However, since we are using delayed evaluation, it is possible to avoid this problem.

D&C delayed evaluation works by recursively calling the D&C primary functions when a compound object representation has been constructed. Since the operands in delayed evaluation are made to be polymorphic D&C types, the operation class knows nothing about the operands it is working on. It only knows their eventual type, which is extracted using `ptr_cast<T>`. The whole scheme ensures that operands, which are themselves compound representations, are evaluated before being used. This works to our advantage, since we can make the classes with differing partitions evaluate to the correct type for extraction. This means that the original operand can even be a class which does not actually contain data to start with but in which `evaluate()` triggers the data's presence.

This usage relies on the the *evaluated types* of the operands being correct for the operation class being used. If this is not the case then `ptr_cast<T>` will return 0, and, in

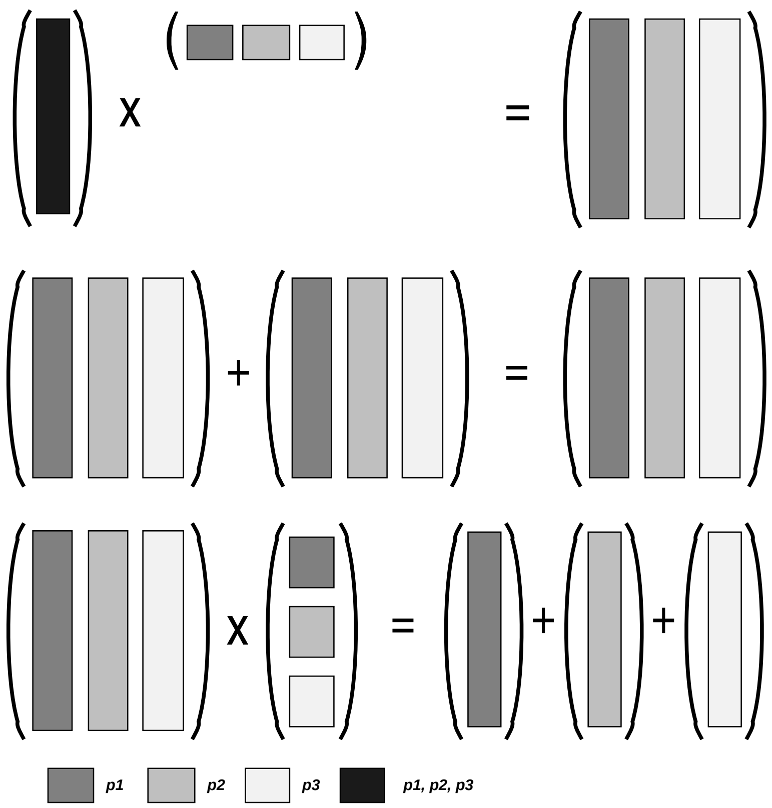


Figure 5.2: Operand partitioning

The diagram shows how vector multiplication, matrix addition and matrix-vector multiplication can be performed consecutively over a distributed processor network. The shading indicates which processor each particular data segment resides on. The final results are distributed vectors which must be added together to achieve a single vector result.

an ideal world an exception would be thrown. In the real world we check for this condition at run-time.

5.3.3 Initial object definitions

For now we will provide only those classes that we need for the Kanerva model.

Since calculations involving the location units are the critical factor in this evaluation, it is this vector that we desire to partition. Thus the vector-vector multiplication in equations 5.4 will have the right-hand operand partitioned and the resultant matrix will be partitioned column-wise. The matrix addition in 5.4 must be partitioned column-wise as well. The matrix-vector multiplication of equation 5.5 must then be of a column-wise partitioned matrix and a partitioned vector, since the vector results from the location units. A little thought shows that this is possible, since multiplying a matrix and vector in this way will result in whole vectors, that simply need to be added together to achieve

```
template <class T>
class zSum : public zContainer<T>
{
  public:
    zSum(const T& t) : zContainer<T>(t) {}
    zSum(const Dac& d) : zContainer<T>(d) { }
    operator T() const { return T(member); }

  protected:
    Dac combine(const Dac& r) const
    {
        return Dac(
            new zSum<T>(
                member + ptr_cast<zSum<T> >(r)→member
                )
            );
    }
};
```

Figure 5.3: Beeblebrox combining adder

The figure shows a D&C container class which sums its data member in `combine()`. *The result returned from* `combine()` *is another* `zSum<T>` *object so that summation will continue recursively.*

the correct result. This addition can be simply achieved in the combine phase of D&C. This scheme is show schematically in figure 5.2.

From this informal analysis we can identify all of the different D&C container class groups of section 3.5.3.

Equipped with these classes we can then define the classes we really want. We derive `zMatrix<T>` and `zVector<T>` classes from `zContainer<T>`, `zMapf<T>` from `zEnvelope` and `zProductVectorVector<T>`, `zProductMatrixVector<T>` and `zSumMatrixMatrix<T>` from `zCompound`. Any partitioning differences can be met by further deriving the classes derived from `zContainer`. In order to fulfil the requirement for the matrix-vector multiplication to have its distributed results added together, we can arrange for `zProductMatrixVector<T>` to change its return argument from evaluate() into a `zSum<T>`. We derive this class from a `zContainer<T>`, but give it a combine function that performs the required addition (figure 5.3). See figure 5.10 for the class relationships.

The matrix class that `zMatrix<T>` is based on is reference counted like the `zArray<T>` of section 3.5.4. The overall operation is similar to that presented by Birchenall [12], allowing matrices and sub-matrices to be used interchangeably.

In all these classes the actual functionality involved – the addition, multiplication etc – is contained in one of the D&C primary functions, usually `evaluate()`. See figures 5.4-5.9. We note here that this operation is usually the only place where the type of the resulting object needs to be known. The resultant type is either hard-wired into the class definition or is passed as a template formal for an object instantiation. In order to make our object-oriented strategy as general as possible, we need to think of ways of removing these dependencies. To this, and other generalizations, we now turn.

```
template <class T> class zMatrix : public zContainer<Matrix<T> >
{
  public:
    zMatrix(const Matrix<T>& m) : zContainer<Matrix<T> >(m) {}

  protected:
/*
 * Divide a matrix into two vertical strips.
 */
    Dac divide(Dac& l, Dac& r) const
    {
        l = make();
        r = make();

        ptr_cast<zContainer<Matrix<T> > >(l)→init(
            Matrix<T>(
                member,
                0,0,member.rows(),member.cols()/2
                )
            );
        ptr_cast<zContainer<Matrix<T> > >(r)→init(
            Matrix<T> (
                member,
                0,member.cols()/2,member.rows(),member.cols()-member.cols()/2
                )
            );
        return 0;
    }
/*
 * Other definitions removed for clarity.
 */
    Dac combine(const Dac&) const;
};
```

Figure 5.4: Beeblebrox matrix class

The figure shows a D&C class for containing matrices. `divide()` *is defined to generate two sub-matrices split column-wise. It is assumed that the matrix class has the appropriate functionality for generating sub-matrices.*

```
template <class T> class zVector : public zContainer<Vector<T> >
{
  public:
    zVector(const Vector<T>& v) : zContainer<Vector<T> >(v) { }

  protected:
/*
 * Divide a vector up into two halves.
 */
    Dac divide(Dac& l, Dac& r) const {
        l = make();
        r = make();

        ptr_cast<zVector<T> >(l)→init(
            Vector<T>(
                member.elems() / 2,
                member,
                0
                )
            );
        ptr_cast<zVector<T> >(r)→init(
            Vector<T>(
                member.elems() - member.elems() / 2,
                member,
                member.elems() / 2
                )
            );
        return 0;
    }
/*
 * Other definitions removed for clarity.
 */
    Dac combine(const Dac&) const;
};
```

Figure 5.5: Beeblebrox vector class

The figure shows a D&C class for containing vectors. `divide()` *is defined to generate two sub-vectors split equally. It is assumed that the vector class has the appropriate functionality for generating sub-vectors.*

```
template <class T, class F> class zMapf : public zEnvelope
{
  public:
    zMapf(const DacVector<T>& v, const F(*p)(T));
    zMapf(const DacVector<T>& v, const F(*p)(T,T), T a);

  protected:
/*
 * map over the vector applying the function pfv.
 */
    Dac zMapf<T,F>::evaluate() const
    {
        if (!args) {
            return Dac(
                new zVector<F>(
                    mapf(
                        ptr_cast<zVector<T> >( e_letter.evaluate() )→data(),
                        (F(*)(T)))pfv,
                        Resolve<F>()
                        )
                    )
                );
        } else {
            return Dac(
                new zVector<F>(
                    mapf(
                        ptr_cast<zVector<T> >( e_letter.evaluate() )→data(),
                        (F(*)(T,T)))pfv,
                        arg1,
                        Resolve<F>()
                        )
                    )
                );
        }
    }

    void (*pfv)();                              // Function to call
    T arg1;                                     // First argument to function
    unsigned short args;                        // Number of function arguments
};
```

Figure 5.6: Beeblebrox vector mapping class class

The figure shows a D&C class for applying a function to consecutive elements of a vector. The class is initialized with a pointer to the appropriate function, and the mapping occurs in `evaluate()`*. Note that the class is parameterized over the argument vector type and the return vector type. If these types differ then the overloaded function* `map()` *is resolved by the dummy class* `Resolve<F>`*.*

```
template <class T> class zProductVectorVector : public zCompound
{
  public:
    zProductVectorVector(const Dac& l, const Dac& r) : zCompound(l,r) { }

  protected:
    Dac evaluate() const
    {
        return Dac (                                    // Multiply two vectors together.
            new zMatrix<T>(
                ptr_cast<zContainer<Vector<T> > >(left.evaluate())→data()
                *
                ~(ptr_cast<zContainer<Vector<T> > >(right.evaluate())→data())
                )
            );
    }
};
```

Figure 5.7: Beeblebrox vector multiplication class

The figure shows a D&C class for multiplying two vectors together. The actual multiplication occurs in `evaluate()`*. Note that the contained data members are evaluated before the multiplication.*

5.4 Generalizing the class relationships

In considering how we can generalize the D&C class relationships, we must address two competing issues.

The strongly typed language we are using tends to force us to make types explicit. This is good in one sense because we benefit from the type checking facilities of the language compiler. The problem is that explicit types lead to an explosion of classes when we use them in a delayed evaluation structure (see section 5.3). We can overcome this explosion from a programing point-of-view by extensive use of parameterized types. However, this does not solve the problem, it merely shifts the onus from programmer to compiler. The multitude of classes still exist, except that they are now generated at compile time. To a certain extent we are prepared to accept this: any lessening of programmer burden is a step in the right direction. However, the over use of templates tends to result in very large executable sizes and long compilation times, which may, or may not, be acceptable.

Another related problem is of concern. The D&C interface and hierarchy has been designed to allow class functionality to be 'mixed-in' to class definitions. Where the resultant class is going to be used many times, this method is entirely acceptable. However, if the resultant class is only going to be used once or twice, we are again creating classes we would like to avoid. We have discussed mix-in templates in section 4.2.1, but this again only alleviates programmer burden.

To address these problems we introduce some new concepts.

5.4.1 Evaluation manipulation classes

Often in defining D&C classes we wish to make one of the primary functions, usually `evaluate()`, return a D&C object that is of a different type to the evaluating object. The example we have encountered thus far is that of matrix-vector multiplication involving the

```
template <class T> class zSum : public zContainer<T>
{
  public:
    zSum(const T& t) : zContainer<T>(t) {}

  protected:
  Dac combine(const Dac& r) const
    {
        return Dac(                                  // Add two operands together.
            new zSum<T>(
                member + ptr_cast<zSum<T> >(r)→member
                )
            );
    }
};

template <class T> class zProductMatrixVector : public zCompound
{
  public:
    zProductMatrixVector(const Dac& l, const Dac& r) : zCompound(l,r) { }

  protected:
    Dac evaluate() const
    {
        return Dac (                                 // Multiply a matrix and a vector.
            new zSum<Vector<T> >(
                ptr_cast<zContainer<Matrix<T> > >(left.evaluate())→data()
                *
                ptr_cast<zContainer<Vector<T> > >(right.evaluate())→data()
                )
            );
    }
};
```

Figure 5.8: Beeblebrox matrix-vector multiplication class

The figure shows a D&C class for multiplying a matrix and vector together. The same considerations apply as for vector-vector multiplication.

```
template <class T> class zSumMatrixMatrix : public zCompound
{
  public:
    zSumMatrixMatrix(const Dac& l, const Dac& r) : zCompound(l,r) { }

  protected:
    Dac evaluate() const
    {
        return Dac (                                         // Add two matrices together.
            new zMatrix<T>(
                ptr_cast<zContainer<Matrix<T> > >(left.evaluate())->data()
                +
                ptr_cast<zContainer<Matrix<T> > >(right.evaluate())->data()
                )
            );
    }
};
```

Figure 5.9: Beeblebrox matrix addition class

The figure shows a D&C class for multiplying two matrices together. The same considerations apply as for vector-vector multiplication.

summation of distributed vectors. In this example `evaluate()` generates a `Vector<T>` which it then converts to a `zSum<T>` so that the proper combination takes place.

The problem is that there are many overall operations which involve different combinations of the operation performed in `evaluate()` and the type `evaluate()` returns. For instance, take the example of multiplying a vector by a transposed vector. If both vectors are distributed then summing the partial results will achieve the desired result. However, if the partial results are concatenated then the result will be element-wise multiplication. In these two cases the operation performed on the vector operands is multiplication, it is only the return type that is changed. In the former case the return type is a `zSum<T>`, in the latter a straight `zVector<T>`. Ideally, we don't want to have to define two separate classes to perform these operations. Instead we use *evaluation manipulation* objects.

Evaluation manipulation classes are simply `zEnvelope`s. The only difference is that `evaluate()` is defined to convert the returned object of its letter's `evaluate()` to a given type. Furthermore, we can make these classes more general by defining a single parameterized class. Thus the type that the result of `evaluate()` is converted to is that of the formal parameter of the manipulator. See figure 5.11.

Evaluation manipulators can also be used to change the resulting partition of a D&C object. We have to remember that D&C operator classes have no knowledge of the types of their operands. They only know the evaluated type, and the evaluated type contains no partitioning information as it has been lost during the execution of `evaluate()`. Thus, in the `zSumMatrixMatrix<T>` class, figure 5.9, we have assumed that `zMatrix<T>` has the same partition as its operands. So far this assumption is correct, since we have defined all `zMatrix<T>`s to be partitioned column-wise. However, we can partition matrices other ways, and if we do then our assumption is no longer valid. We could define a `zSumVerticalMatrixMatrix<T>`, but this again results in more classes than we would like. Instead, we let the `zSumMatrixMatrix<T>` be wrapped up in an evaluation manipulator, and require the user of the class to provide a manipulator of the right type.

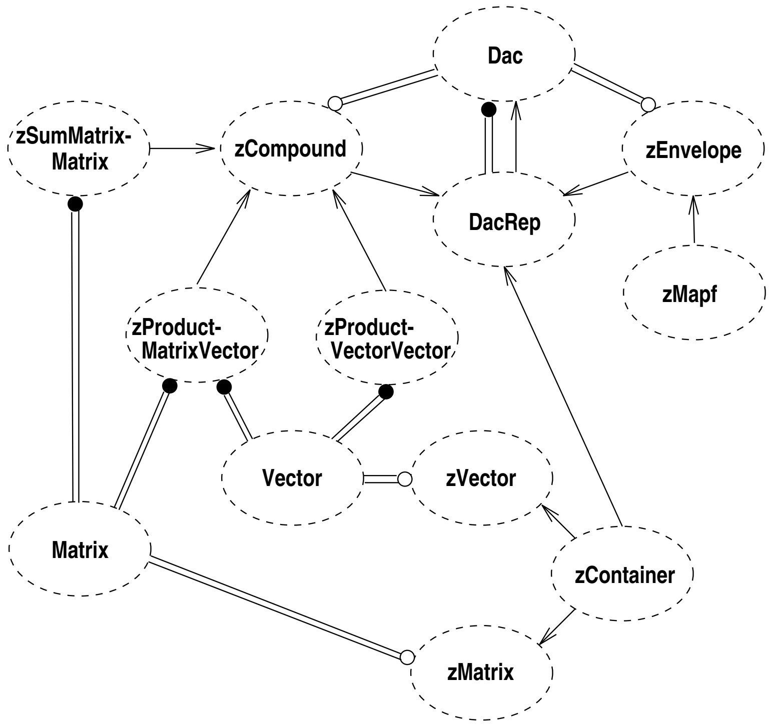


Figure 5.10: Initial D&C classes

The diagram shows the hierarchy of classes introduced so far. The diagram uses Booch's notation.

In the particular case above, it is difficult to see how the desired affect could have been achieved with mix-ins, thus highlighting the usefulness of this class. This is because we are not choosing between different `evaluate()`s, but are rather trying to combine two different `evaluate()`s to achieve a whole.

5.4.2 Generic templates

In examining evaluation manipulators we have introduced the idea of making operand types be the formal parameters of a parameterized type. We can extend this idea further to the classes we have already described, by making their operand types template formals, rather than hardwired matrices or vectors. Thus `zSumMatrixMatrix<T>`, for example, becomes `zSumOpTemplate<T1,T2>`, the two template formals `T1` and `T2` being `Matrix<T>`s in this case. See figure 5.12.

We can use this idea for multiplication as well, replacing `zProdMatrixVector<T>` with `zProdOpTemplate<T1,T2>` and using `zEvaluateTemplate<T>` to modify `evaluate()`'s return type to a `zSum<Vector<T>>`. This particular example throws up another issue that has already surreptitiously been answered. The problem is this: what is the evaluated return type of the `zProdOpTemplate<T1,T2>`? Unlike addition, the return type can be that of either, or neither, of the operands. Matrix-vector multiplication returns a matrix, vector-transpose vector multiplication returns a scalar, transpose vector-vector multiplication returns a matrix. We have to know the return type in order to encapsulate

```
template <class T> class zEvaluateTemplate : public zEnvelope {
  public:
    zEvaluateTemplate(const Dac& d) : zEnvelope(d) {}

    Dac evaluate() const
    {                                          // Formal parameter is a D&C class
        return Dac( new T(e_letter.evaluate()) );
    }
};
```

Figure 5.11: Evaluation manipulator

The figure shows a D&C class for evaluation manipulation. `evaluate()` *is defined to return an object of the type of* `zEvaluateTemplate<T>`*'s formal parameter. Note that the contained object is evaluated before being converted into the formal parameter type.*

```
template <class L, class R>
class zSumOpTemplate : public zCompound {
  public:
    zSumOpTemplate(const Dac& l, const Dac& r) : zCompound(l,r) { }

  protected:
    Dac evaluate() const
    {
/*
 * add and containerize letters
 */
        return Container (
            ptr_cast<zContainer<L> >( left.evaluate() )->data()
            +
            ptr_cast<zContainer<R> >( right.evaluate() )->data()
            );
    }
};
```

Figure 5.12: General operand summation template

The figure shows a D&C class that generalizes some of the operation classes introduced so far. The formal parameters of the class, `L` *and* `R`*, are the types of the operands to be added together. The actual addition is performed in* `evaluate()`*.*

the evaluated object in an appropriately typed D&C object. Or do we? In figure 3.19 there is a template *function*, `Container()`, defined. By using a template function we can make the compiler match the type of the function's parameter to the `zContainer<T>` we require. The type of container required is fixed by the return type of the operation in question. The function can then return this, correctly typed, object. By using this function to construct `zContainer<T>`s we can avoid having to put explicit typing in the `zProdOpTemplate<T1,T2>` [31].

5.4.3 Type field operation classes

So far we have managed to reduce our original plethora of classes to fewer parameterized classes. We can reduce the number still further by introducing type-fields for some common operators. This means that the number of instantiated classes is not dependent on the number of operators. Instead it is simply dependent on the number of different types being used for delayed evaluation.

The key to this is introducing an enumerated type representing different operations: `Mult` for `operator*`, `Plus` for `operator+` and so on. Then, for given argument types, we can have one class representing all infix operations. At run-time we choose between the different possibilities using a switch statement. This necessarily introduces additional overhead, but greatly simplifies our class structure. In transmission, the operator type of the object is transmitted as well as the object and its arguments.

Since most *operand* type checking will be performed by an outer wrapper class (see section 5.6), the type-checking we lose through the use of type fields is not significant. Although it is possible for there to be run-time errors if the wrapper class has not done its job properly.

The last thing we must note is that this simplification cannot be made without the use of evaluation manipulators. Although some operations could be implemented without evaluation manipulators – simple additions etc – the number would be so few as to achieve little real reduction in class numbers. With evaluation manipulators however we can encapsulate all operations in a single class. See figure 5.13.

5.4.4 Polymorphic evaluation manipulators

In the previous sections we used evaluation manipulators to reduce the number of class definitions; whether programmed or automatically instantiated from parameterized classes. However, `zEvaluateTemplate<T>` is parameterized itself, thus every different use of it introduces a new instantiation. This is acceptable from a time perspective but not necessarily so from a space perspective. Thus we introduce a new type of evaluation manipulator that is truly polymorphic and needs only a single class definition.

In order to implement this class we use the ideas of classless languages such as Self [106]. Instead of an evaluation manipulator containing a template formal which defines the type of object we are going to build, the class contains an actual object of the type we require. From this object we clone objects as needed. In order to be truly polymorphic, and therefore typeless, the type of the object member is simply a `Dac`. This means that the *underlying* type can be anything at all, without the need for additional classes. Initialization can be carried out by making the clone function, which has to be virtual anyway, accept an argument of type `Dac`. It is then up to the cloning object to use this argument in an appropriate way. Since the cloning object is never actually used itself, it can be reference counted in `divide()` et al. rather than duplicated. See figure 5.14.

```
template <class L, class R>
class zOpTemplate : public zCompound {
  public:
    zOpTemplate(const Dac& l, const Dac& r, const Operation o)
      : zCompound(l,r), op(o) { }

  protected:
    int op;                                                  // type of operation
    Dac evaluate() const
    {
/*
 * operate on and containerize letters
 */
        Dac l = left.evaluate();
        Dac r = right.evaluate();

        Dac x;

        switch (Operation(op)) {
          case Plus:
            x = Container ( ptr_cast<zContainer<L> >(l)->data()
                          + ptr_cast<zContainer<R> >(r)->data());
            break;
          case Minus:
            x = Container ( ptr_cast<zContainer<L> >(l)->data()
                          - ptr_cast<zContainer<R> >(r)->data());
            break;
          case Mult:
            x = Container ( ptr_cast<zContainer<L> >(l)->data()
                          * ptr_cast<zContainer<R> >(r)->data());
            break;
          case PlusEq:
            x = Container ( ptr_cast<zContainer<L> >(l)->member
                          += ptr_cast<zContainer<R> >(r)->data() );
            break;
          case MinusEq:
            x = Container ( ptr_cast<zContainer<L> >(l)->member
                          -= ptr_cast<zContainer<R> >(r)->data() );
            break;
          case MultEq:
            x = Container ( ptr_cast<zContainer<L> >(l)->member
                          *= ptr_cast<zContainer<R> >(r)->data() );
            break;
          default:;
        }
        return x;
    }
};
```

Figure 5.13: Type field operator template

The figure shows a D&C class for performing various binary operations on two D&C objects. The operands are held in the `zCompound` *which* `zOpTemplate<L,R>` *inherits from. The formal parameters of the class are the actual types of the operands. Which operation is performed is determined by the type field* `op`*. Note that the returned objects are encapsulated using the* `Container()` *template function, so that the correct return type is selected.*

```
class zEvaluateWrap : public zEnvelope
{
  public:
    zEvaluateWrap(const Dac& w, const Dac& d) : zEnvelope(d), wrapper(w) { }

  protected:
    Dac wrapper;
    Dac evaluate() const
    {
        return wrapper.make_copy( zEnvelope::evaluate() );
    }
};
```

Figure 5.14: Polymorphic evaluation manipulator

The figure shows the polymorphic equivalent of `zEvaluateTemplate<T>`*. Each object of the class contains a D&C object,* `wrapper`*, into which the return value of* `evaluate()` *is put. It is assumed that all D&C classes which will be used as wrappers have the virtual member function* `make_copy()` *defined.*

Two caveats apply to this class. The first is that the cloning function fattens the interface of the base D&C class `Dac`. The second is that carrying a real object around creates additional overhead in transmission. Since cloning with a generic D&C argument is likely to be a more generally useful function, the former is not a particular hardship. The latter means that careful consideration should be given by the programmer as to what type of evaluation manipulator best suits his needs.

This type of manipulation is similar to the dynamic mix-ins of section 4.2.3

5.5 Evaluation optimizations

Thus far we have considered what can be termed programming optimizations. Since we are also, and perhaps primarily, concerned with raw performance it would be good if we could provide some classes that implement run-time optimizations. Once again, we will use the Kanerva model as our trial example and try and extrapolate possible enhancements from the possibilities within this example.

5.5.1 Persistence

The first thing we note is that currently, Kanerva storage and retrieval constitute two different operations both of which require a separate D&C pass. We note also that for both operations the data (location units and weights matrix) is partitioned the same way. This is also true for different storage and retrieval operations. In fact both operations can be carried out with the data distribution being fixed. However, our current D&C implementation requires that the data be divided, broadcast and retrieved for each D&C pass. This is clearly ludicrous, in terms of performance, since we would be needlessly transmitting large amounts of data. What is really needed is for the data to be locally cached between operations so that the only transmission necessary is that of a new input pattern.

```
class zPersistent : public zEnvelope {
  protected:
    enum State { Real, Virtual, Flush };

  public:
    zPersistent(const Dac& d)
        : zEnvelope(d), p_obj(p_key++), p_state(Real), p_fixed(false) {}

    void real() { p_state = Real; }
    void flush() { p_state = Flush; }

  protected:
    unsigned int p_obj;
    short p_state;
    short p_fixed;
    static unsigned int p_key;                          // current object id
    static HashList<Dac>* p_list;                       // list of cached Dac's

    Dac divide(Dac& l, Dac& r) const
    {
        if (node()==1 && p_state ≠ Flush) {
            if (!p_list→empty()) {
                if (Dac::schd→fix()) {
                    ((zPersistent*)this)→p_fixed = true;
                }
            }
            if (p_list→search((p_obj≪24)+node()) ≠ 0) {
                ((zPersistent*)this)→p_state = Virtual;
            }
            else {
                p_list→insert((p_obj≪24)+node(), *this);
            }
        }
        zEnvelope::divide(l,r);

        return 0;
    }

    Dac evaluate() const
    {
        Dac r;
/*
 * If  it's real, evaluate and  cache it. If  it's not read the cache.
 * Then flush the object if required.
 */
        if (p_state == Real) {
            r = e_letter.evaluate();
            p_list→insert((p_obj≪24)+node(), r);
        }
        else {
            if ((r = p_list→search((p_obj≪24)+node())) == 0) {
                error(__FILE__,__LINE__) ≪ "cache miss" ≪ fatal;
            }
            if (p_state == Flush) {
                p_list→delete_item((p_obj≪24)+node());
            }
        }
        return r;
    }
};
```

Figure 5.15: Persistent beeblebrox class

```
const Dac& persist(Dac& d)
{
    if (ptr_cast<zPersistent>(d)) {
        ptr_cast<zPersistent>(d)→real();
    } else {
        d = Dac( new zPersistent(d) ) ;
    }
    return d;
}

const Dac& flush(Dac& d)
{
    if (ptr_cast<zPersistent>(d))
        ptr_cast<zPersistent>(d)→flush();
    return d;
}
```

Figure 5.16: Persistent object manipulators

The figure shows two functions for making D&C objects persistent and transient.
The previous figure shows a D&C class for making D&C objects persistent. Objects that are designated persistent are entered into the hash table `p_list` *in* `evaluate()`*, unless they are already there in which case they are retrieved. The first pass through* `divide()` *registers objects that will be persistent and manipulates the state of the scheduler.*

How can we implement this in a transparent fashion?

We make the assumption that data will not be required until `evaluate()` is executed. If this assumption is incorrect then caching data will not gain anything anyway, as data will not necessarily have been distributed. We define a `zEnvelope` class for which `evaluate()` consults its cache for the letter and if found returns that. If the `zEvaluate` object cannot get a cache hit then it returns `evaluate()` of its letter and caches the return value.

This implementation strategy is essentially giving D&C objects the ability to be persistent [13, p190]. The envelope approach we have adopted means that the letter object need not have any knowledge about persistent objects at all. It is only the envelope that has that knowledge. Of course these persistent objects need to be created in the first place and it is here that the programmer can give hints as to which objects need to be persistent. We can do this by providing a function `persist(Dac&)` which takes its D&C argument, and replaces the letter (remember `Dac` objects themselves are envelope classes) with a persistent `zEnvelope` letter, which itself has the original letter as its letter. See figures 5.15 and 5.16.

Ideally we would like the system to automatically make objects persistent as required. However, this would probably entail structure inferences which only a compiler could provide, and thus would draw us away from our "no-new-compiler" philosophy. One possibility is to cache *all* objects and then discard objects that haven't been referenced for a while[1]. The only problem with this is that the cache needs to be big enough to

[1] Like a LRU cache

hold *all* objects from a single D&C pass. The resultant space overhead might well prove unacceptable.

5.5.1.1 Temporary persistence

If we were to do a Kanerva retrieval immediately after a Kanerva storage then we note that the vector of active locations is the same for both operations. In this instance it would be natural to make this particular object persist between the two operations. This is perfectly feasible with the implementation given above, but what of the cached objects once we have used them? Really we need to be able to flush objects on their final, fixed usage. To do this we provide another function `flush(Dac&)` which causes its argument to be retrieved from cache when `evaluate()`ed, however, the object is subsequently removed from the cache.

Again, it would be good to devise some policy which made this happen automatically.

5.5.1.2 Recording the schedule

In order to implement persistent objects, we need to be able to guarantee that the processor that data has been cached on is the processor from which the same data will be retrieved. Obviously, having a dynamic schedule will not provide this guarantee and it is therefore necessary to fix the schedule after it has been played through once. Rather like a cassette tape, we need to record what originally happened and then playback the recording for subsequent schedules. At first sight it would appear that all that is necessary is to record the order and processor to which data was sent. However, a little thought shows that this is not sufficient. Although we can guarantee that data is going to the right destination, we cannot guarantee that it is the right data that is being sent. The data that is sent is that at the bottom of the evaluation stack, and the node at this location is dependent on how far the evaluation has progressed. Thus we need to ensure that the correct data is sent. This can be achieved by labelling each node in level-order and recording each object that is sent and received. Then, when the schedule has been recorded, worker nodes will only request the correct data item, and the scheduler will only release the data it has to the correct node.

Two more issues need to be addressed:

- We need to prevent the evaluator from evaluating nodes that the scheduler is going to need at a later time. We can achieve this by setting a 'high-water' mark in the stack, indicating the 'lowest' node that the scheduler evaluated first time around. In any independent evaluation[2], this position will be the first node that the evaluator will try that is needed by the scheduler – by virtue of the level-order numbering and depth-first evaluation employed. Thus, if we prevent the evaluator from using this node, all the others required by the scheduler in that evaluation will be safe. However, this does not go far enough for processors that are not the origin – i.e. most. These processors will process many independent evaluations during the course of recording, and thus a high-water mark must be set for *each* object that is evaluated. In order to do this, we need to record the last scheduled output for any evaluation – to use in setting the high-water mark – as well as recording the scheduled outputs themselves.

[2] The cycle of receiving an object, evaluating it and returning it to the sender.

- We need to determine when the scheduler switches from recording to playback. For the root processor this is trivial, the switch can be made when the persistent objects switch from storage to retrieval. For worker processors this is not so straightforward. These processors have no knowledge of the 'state' of the machine, they simply take nodes and evaluate them. Thus, we need to build into the scheduler some checking of whether the node being evaluated is persistent and then whether it is cached or not. The switch from recording to playback can be made to logically mirror this information. This is slightly complicated by the fact that a worker processor cannot know to switch until it actually has the first node in the playback. Thus, the first request in the playback must be for any node, putting the onus on the scheduler to ensure that the correct node is sent.

  In fact, in practical implementations it was found that broadcasting the switch to worker processes, from recording to playback, achieved far more reliable results.

5.5.2 Delayed assignment

We now have a system that enables us to avoid unnecessary broadcasts of information. However, a major problem remains: cached D&C objects still need to be initialized on the root processor. If our Kanerva model is very big then the space overhead that this incurs will prove debilitating.

The assumptions we made in the previous section, about where cached data would be used, are still valid for data initialization. We only need to access the data during `evaluate()`, *after* the data has been distributed. So, the question we must now ask is: how is the data initialized. There are three possibilities:

- initialized to zero,
- initialized with random data, or
- initialized from a data file.

The Kanerva model actually fits the second case since the location units are initialized with random data. For each of these cases we can define a class that *virtually* represents the data, and structure, of the object we require in `evaluate()`. These classes need not contain any data at all – and so will incur negligible space overhead – but merely contain a definition of how to realize the data when required. For zero or random initializations the implementation is simple. For initializations from a file the situation is slightly complicated by the need to track the position in the file from which the data will come. The whole process can be concealed behind the initialization of `zMatrix<T>`s etc. We will term this technique *delayed assignment*; see figure 5.17.

5.5.2.1 Back-propagation with delayed assignment

This technique is potentially very useful on hardware with limited space resources. Thus, we can try the back-propagation experiments described in section 2.4 with even larger numbers of training frames. Figure 5.18 compares the various options for 1024 training frames.

Examining the results we can see one factor dominates all the results, and that is whether data is obtained from file or generated on-line. If data is generated on-line then we

```
TEMPL class zDelayMatrix : public zNull {
  public:
    zDelayMatrix(const char* fn) : d_type(fn) {
        ifstream f(fn);
        if (!f.good()) error() << "open:  " << fn << fatal;
        f >> d_rows >> d_cols;
    }
    zDelayMatrix(int r, int c, const abstractType& a)
        : zNull(), d_type(a), d_rows(r), d_cols(c) {}
    zDelayMatrix(int r=0, int c=0) : zNull(), d_rows(r), d_cols(c), d_type() {}

  protected:
    Dac evaluate() const {
/*
 * make the virtual data into real data and return it.
 */
        Matrix<T> m;

        if (d_type.id() == abstractType::random) {
            m = Matrix<T>(d_rows, d_cols);
            m = d_type;
        }
        else if (d_type.id() == abstractType::file) {
            ifstream f(d_type.rep->a_file.name); int i; T t;
            if (!f.good())
                error() << "open:  " << d_type.rep->a_file.name << fatal;
            f >> i >> i;                                        // get the size
            for (i=0; i<d_type.a_pos; i++) f >> t;          // skip uneeded data
            m = Matrix<T>(f, d_rows, d_cols);
        }
        else m = Matrix<T>(d_rows, d_cols);

        return Dac( new zContainer<Matrix<T> >(m) );
    }

    Dac divide(Dac& l, Dac& r) const {
/*
 * divide the virtual data into two parts
 */
        l = make(); r = make();

        zDelayMatrix<T> *ld = ptr_cast<zDelayMatrix<T> >(l);
        zDelayMatrix<T> *rd = ptr_cast<zDelayMatrix<T> >(r);

        ld->init(d_rows, d_cols/2); rd->init(d_rows, d_cols - d_cols/2);

        if (d_type.id()==abstractType::random) {
            ld->d_type.seed=d_type.seed<<1;
            rd->d_type.seed=(d_type.seed<<1) +1;
        }
        else if (d_type.id()==abstractType::file) {
            ld->d_type.a_pos=d_type.a_pos;
            rd->d_type.a_pos=d_type.a_pos+(d_rows * d_cols/2);
        }

        return 0;
    }

    int d_rows, d_cols;
    abstractType d_type;
    void init(const int r, const int c, const int s=0) {
        d_rows = r; d_cols = c; if (s) d_type.seed = s;
    }
};
```

Figure 5.17: Delayed assignment matrix class

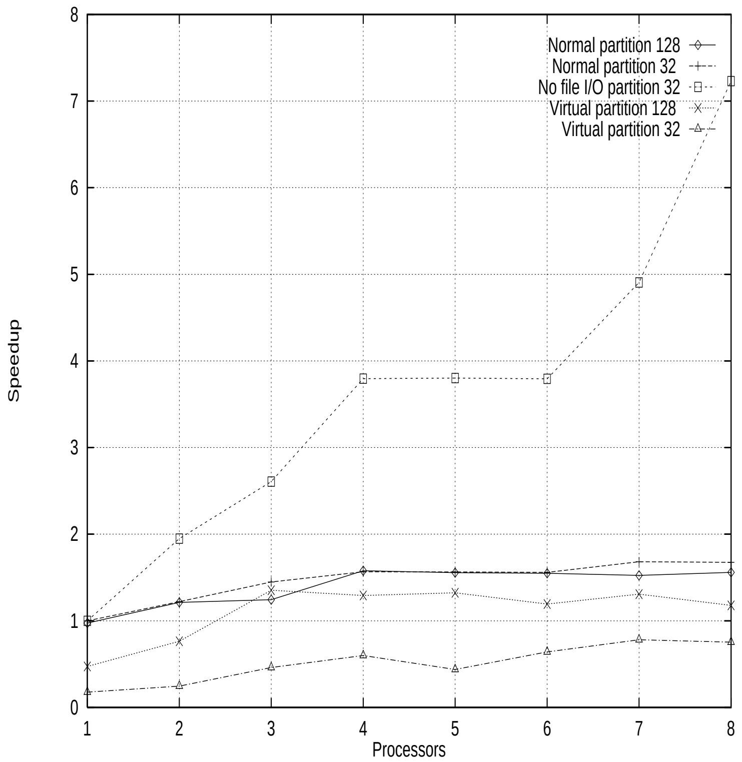


Figure 5.18: Results for different types of D&C back-propagation

The diagram shows the performance figures for different implementations, with different partitions, of D&C back-propagation. The speedup is given by $T(N)/T(1)_{best}$ where $T(1)_{best}$ is the time on 1 processor for a standard, fast, implementation.

obtain a nearly optimal speedup curve. If the data is obtained from file then performance is very poor indeed. These experiments were carried out on a transputer farm and it is obvious that the I/O hardware employed must be fairly poor. However, this factor aside we can still gain some insight into the relative performance of delayed assignment.

Partition	256	512	1024	2048
Speedup	0.72	1.43	1.22	0.90

Table 5.1: Delayed assignment for 8192 training frames on 8 processors

We observe that for small partitions performance is extremely poor. We surmise that this is because for every task, on average half the data file has to be scanned. If the number of tasks is large then this overhead is going to be large also. If we increase the partition then performance begins to approach that of normal assignment. Of course the real advantage is in evaluating very large problems, and a problem 8 times the size of the maximum achieved with normal assignment, was successfully evaluated using delayed assignment. The results are shown in table 5.1 assuming that the serial execution is going to be 8 times that for 1024 frames. The results show that it is no worse than the results for lower numbers of frames, although overall performance is very poor. We envisage using a binary file format making some difference, but better overall I / O is what is really required.

5.5.3 Distributed assignment

We have now considered the cases of: delaying assignment till distribution, and caching data that is only going to be accessed when distributed. We have one further case to consider. If the data is initialized when distributed and updated from cache when distributed, then there will be circumstances when the result need never be returned. For instance if we invoke `operator+=` on distributed matrices then the assignment can be performed in place. If we have no need for the result, apart from for the next distributed operation, then we can wrap the result in an *evaluation manipulator* that simply returns `zNull`. In this way we can save on yet more needless communication. `operator+=` and others can easily be defined in `zOpTemplate<L,R>`,

Of course, if we do then need an undistributed result we must retrieve the distributed results. This can simply be achieved by re-evaluating the delayed matrix wrapped in an evaluation manipulator that defines concatenation of distributed results.

5.5.4 Summary

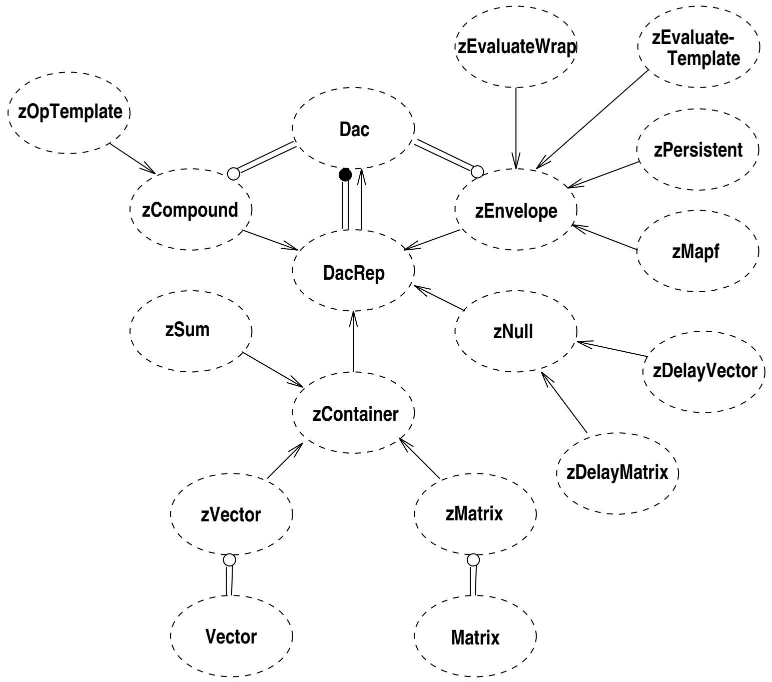


Figure 5.19: Delayed evaluation D&C classes

The diagram shows the hierarchy of delayed evaluation classes introduced in this chapter. The diagram uses Booch's notation .

We have described may different types of D&C letter classes. These classes provide programming and evaluation convenience and efficiency. Each class is highly modular having no dependency upon classes to which it has no logical relation. Even classes which have logical relationships are logically distinct. General manipulation type classes have no impact, performance or compilation-wise, on a program unless they are actually used. Objects which are modified by these classes have no knowledge that this is the case; encapsulation is maintained. That this is the case is attributable to the overall design of the D&C system, and the object-oriented environment.

The class hierarchy of classes described so far in this chapter is shown in figure 5.19.

5.6 Managing the evaluation classes

So far we have described various delayed evaluation classes; classes to make a programmer's life easier and classes to achieve better run-time performance. All of these classes are concerned with D&C evaluation, however, none of them provide any of the characteristic functionality of the structures they represent. There is no public `operator+` defined for `zMatrix<T>`s. This is because each class is a *letter* type class derived from `DacRep`. Each is derived directly or indirectly from `DacRep` and none are for direct manipulation by an end-user. As we have intimated in section 5.4.3, it is the envelope class `Dac` that provides the manipulation functionality. We now turn to the definition of this type of class when applied to algebraic operations.

5.6.1 General management

As the management classes we require must be wrappers, it is logical that they be derived from `Dac`. If they are defined in this way then their default operation will be as for normal wrappers. Any further functionality can be added as required.

One characteristic of these wrappers, whatever the types they are managing, is that `operator=` invokes `Dac::run()`. This is in keeping with how delayed evaluation is supposed to work. Since we may wish to assign objects without evaluating them, we will also define `operator<<` as an insertion operator working to this effect.

Most other operators are specific to the representation type – matrices, vectors etc – and it is to these that we will now turn.

5.6.2 Managing matrices

In this section we describe in detail the implementation of operators that we require for matrices in the Kanerva model. These operations are defined for `DacMatrix<T>`, an envelope derived from `Dac`.

5.6.2.1 Initialization and instantiation

The first thing we need to be able to do with `DacMatrix<T>`s is to instantiate them. Thus, we define a constructor that takes a real `Matrix<T>` as an argument and builds a `zMatrix<T>` from it. We also wish to be able to define `zDelayMatrix<T>`s of arbitrary dimensions, and so we provide a constructor that creates a `zDelayMatrix<T>` from row and column arguments.

Having the ability to create `zDelayMatrix<T>`s is good, but we also need to be able to initialize the structure with zero, random or file arguments. To do this we define an `operator=` that takes an `abstractType` as an argument. However, we must remember that this operator has no knowledge about the *actual* type of the object it represents.

```
template <class T> class DacMatrix : public Dac {
  public:
    DacMatrix() : Dac() { }
    DacMatrix(const Matrix<T>& m) : Dac( new zMatrix<T>(m) ) { }
    DacMatrix(const int r, const int c)
        : Dac( new zDelayMatrix<T>(r,c) ), transposed(false) { }
    const abstractType& operator= (const abstractType& a)
    {
/*
 * if its a delay vector then we can change its initialisation spec.
 */
        if (ptr_cast<zDelayMatrix<T> >(*this)) {
            ptr_cast<zDelayMatrix<T> >(*this)->init(a);
/*
 * if its not then we'll try to assign to it anyway
 */
        }
        else if (ptr_cast<Container_t>(*this)) {
            ptr_cast<Container_t>(*this)->member = a;
        }
        else {
            error() << "parse error:  DacMatrix = abstractType" << fatal;
        }
        return a;
    }
};
```

Figure 5.20: Initialization and instantiation of D&C matrices

The figure shows how a D&C matrix wrapper class can be initialized with matrix type parameters. A `zDelayMatrix<T>` *is used if possible, otherwise a* `zMatrix<T>` *is used. Note that the class is derived from a* `Dac` *so that all* `Dac`*-type functions will work for objects of this type.*

Therefore we must check to see if its type really is a `zDelayMatrix<T>` before assigning to it. If it is not, then there is a possibility it is simply a container in which case we can initialize that instead. See figure 5.20.

5.6.2.2 Simple operations

We now need the ability to perform simple operations on `DacMatrix<T>`s using `zOpTemplate<L,R>`. By simple, we refer to operations where the operand types and return type are the same as the object type. For these cases we do not need to do any special processing with evaluation manipulators, we simply need to make sure that the `evaluate()`ed partition is consistent with the `divide()`ed partition.

We can do this quite simply by making the infix operators for `DacMatrix<T>`s create `zOpTemplate<Matrix<T>,Matrix<T>>`s. We wrap this object in an evaluation manipulator which turns the `zContainer<Matrix<T>>` return type back into a `zMatrix<T>`. We also must check for the possibility of `zPersistent` objects. If the object in question is persistent then we can make the evaluation manipulator return `zNull`. Figure 5.21 shows this functionality and also demonstrates the use of polymorphic evaluation manipulators for the persistent case.

```
template <class T> class DacMatrix : public Dac {
    const DacMatrix<T>& operator+=(const DacMatrix<T>& d)
    { return op_eq(d,PlusEq); }
    const DacMatrix<T>& operator-=(const DacMatrix<T>& d)
    { return op_eq(d,MinusEq); }
    DacMatrix<T> operator- (const DacMatrix<T>& d) const
    { return DacMatrix<T>( op(d,Minus) ); }
    DacMatrix<T> operator+ (const DacMatrix<T>& d) const
    { return DacMatrix<T>( op(d,Plus) ); }

    DacRep* op(const DacMatrix<T>& d, const Operation o) const
    {
        DacMatrix<T> l=*this,r=d;

        DacRep *p;
        if (ptr_cast<zPersistent>(*this)) {
/*
 * stop anything coming back for a delayed matrix
 */
            p = new zEvaluateWrap(
                Dac( new zNull ),
                Dac(
                    new zOpTemplate<Matrix<T>,Matrix<T> >(l,r,o)
                    )
                );
        } else {
            p = new zEvaluateTemplate<zMatrix<T> > (
                Dac(
                    new zOpTemplate<Matrix<T>,Matrix<T> >(l,r,o)
                    )
                );
        }
        return p;
    }

    const DacMatrix<T>& op_eq(const DacMatrix<T>& d, const Operation o)
    {
        Dac x( op(d,o) );
        x = x.run();

        if ( ! ptr_cast<zPersistent>(*this) ) {
            if ( ptr_cast<Container_t>(x) ) {
                xlate(
                    new zMatrix<T>(
                        ptr_cast<zContainer<Matrix<T> > >(x)->data()
                        )
                    );
            } else {
                error() << "result was not a container" << fatal;
            }
        }
        return *this;
    }

};
```

Figure 5.21: Simple operations on D&C matrix objects

The figure shows how the matrix wrapper can be used to construct binary operation objects. Note the use of evaluation templates to modify distributed return types.

Operators that perform an assignment as well should simply fit within this framework. The only difference in operation is that the result is swapped for the envelope's current letter.

5.6.2.3 Other operations

For the Kanerva model, the only other operation we really need for matrices is vector multiplication. This can quite easily be achieved by using a `zSum<Vector<T>>` evaluation manipulator, described previously. The actual operation can be implemented using a `zOpTemplate<Matrix<T>,Vector<T>>` for multiplication. See figure 5.22.

```
template <class T>
DacVector<T> DacMatrix<T>::operator* (const DacVector<T>& v) const
{
    return DacVector<T> (
        new zEvaluateTemplate< zSum< Vector<T> > >(
            Dac(
                new zOpTemplate<Matrix<T>,Vector<T> >(*this,v,Mult)
                )
            )
        );
}
```

Figure 5.22: Matrix-vector multiplication

The figure shows the implementation of a matrix wrapper multiplication operator. A `zOpTemplate<L,R>` *is constructed, and its distributed return type is modified by an evaluation template so that resulting vectors are added together.*

5.6.3 Managing vectors

Most vector operations are implemented in an almost identical fashion to matrix operations. We will therefore not discuss their implementation here.

One operation we will mention though, is that of mapping over vector elements. Since `mapf(<DacVector>, <fn>)` is perhaps a more intuitive usage than `<DacVector>.mapf(<fn>)`, we will provide global template functions to achieve this functionality. These functions simply create `zMapf<T,F>`s. `evaluate()` for `zMapf<T,F>`s returns a `zVector<T>` anyway so nothing more is required. See figure 5.23.

Note that we could argue that `evaluate()` should simply return a `zContainer<Vector<T>>` and then require an evaluation manipulator to arrange the right type for reconstruction. Two considerations make this approach less appealing. The first is that vectors, unlike matrices, will only ever be in one distributed, partitioned state. Thus assuming, in `zMapf<T,F>`, that the partitioned state is going to be constant is reasonable. The second consideration is that a `zVector<T>` has a `zContainer<Vector<T>>` as a base class. Thus, if we wish to manipulate the evaluation return type, casting to a container will still work.

In describing `zMapf<T,F>`s we have neglected an important topic; that of function portability. The next section will address this issue.

```
template <class T> DacVector<T> mapf(const DacVector<T>& d, T (*p)(T))
{
    return DacVector<T>( new zMapf<T,T>(d,p) );
}

template <class T> DacVector<T> mapf(const DacVector<T>& d, T (*p)(T,T), T a)
{
    return DacVector<T>( new zMapf<T,T>(d,p,a) );
}

template<class T, class R>
DacVector<R> mapf(const DacVector<T>& d, R (*p)(T), Resolve<R>)
{
    return DacVector<R>( new zMapf<T,R>(d,p) );
}

template<class T, class R>
DacVector<R> mapf(const DacVector<T>& d, R (*p)(T,T), T a, Resolve<R>)
{
    return DacVector<R>( new zMapf<T,R>(d,p,a) );
}
```

Figure 5.23: Mapping over D&C vectors

The figure shows how `zMapf<T,F>` *objects can be constructed using template functions. The constructed objects are wrapped in* `DacVector<T>`*s so than further processing can occur.*

5.6.3.1 Functors

In order to apply a function iteratively to the elements of a vector, we need to have the address of that function. This is simple to provide on a single processor, but when computation is offloaded to another processor how can the function be transported? On processors with the same hardware architecture, the function addresses *might* be the same. With heterogeneous processor clusters this will definitely not be the case. Since C++ is not interpreted, it is not practical to transmit the function definition. Instead we define objects called *functors*. Each of these objects represents a function but it also contains a textual key for the function. This key can then be transmitted between processors and used to look up the actual function address. Of course, functions to be used in this way must be registered in some way. In order to do this we provide a macro `FUNCTOR()` for defining functions so that they are automatically registered.

5.7 Intelligent partitioning

We now have letter and envelope classes which should enable us to implement the Kanerva model using D&C and delayed evaluation. However, one very important issue still remains.

In discussing the wrapper implementations, `DacMatrix<T>` and `DacVector<T>`, we have made little reference to partitioned state. We have already described how different operations can be implemented in a number of different ways, depending on the distributed state of the data. We have not, however, described how we manage, or decide on, this state.

Since the whole concept of delayed evaluation revolves around run-time optimization, managing partitioned state must ultimately have the same focus. Keeping this in mind we will first concentrate on structure manipulation.

5.7.1 Type changing

As mentioned in section 5.3.1, vectors and matrices can be distributed in a number of different ways. This distribution, however, should be a characteristic that is hidden, to a large degree, by the implementation. In managing these structures we wish to deal with matrix and vector abstractions, rather than the specifics of their implementation. However, managing abstractions in this way falls neatly into the scope of object-oriented technology. We define the various partitioning options as classes with a common base class, representing the overall abstraction we are dealing with.

This approach is fine for presenting a polymorphic *interface*, however, part of the problem is that we cannot say initially with much certainty, of what *actual* type an object is. What we really need to be able to do is instantiate an object of the *abstraction*, and then manipulate that according to subsequent requirements. To do this we define a dynamically bound function `change_type()`, which takes as an argument the type of partition required, and returns a standard D&C object. We can then define this function in the base class abstraction to return an appropriately partitioned object. Additionally, for objects that already have a partitioned state, we can define this function to return a null object if the required and actual states do not match. We also define a partition enquiry function, `partition()`, for all delayed evaluation type D&C objects. See figure 5.24.

In doing this we have effectively eliminated much of the need for evaluation wrappers to deal with partitioned objects. Instead, partitioning information is confined to the classes to which it actually applies.

We note also that since `change_type()` is part of the general D&C interface, it can be defined recursively for `zEnvelope` and `zCompound` objects. This means that entire D&C compound objects can be changed using this generic interface, *without* any lower level involvement by the management class in question. It also means that D&C envelopes like `zPersistent` can transparently implement `change_type()` without impinging upon the functionality of their letter. Again the type information is restricted to those classes to which it applies.

5.7.1.1 Interaction with generic classes

The only problem that this approach falls foul of, is with its interaction with generic D&C classes. We have defined generic operator classes that can cope with most variations of infix operation. However, the partitioning requirements associated with each operation can be quite varied. A simple solution is to combine the partitioned states of the *operands* in some general fashion, though this does not completely solve the problem. Consider matrix-vector multiplication. If the matrix is partitioned vertically and the vector is partitioned horizontally, what is the partitioned state of the result? In actual fact the result has neither of these states, as to achieve a result partitioned vectors must be summed together (see figure 5.2). It is difficult to see how this particular operation can reveal its true state without specific programmer intervention. One way to provide this specific information is to define template specializations for the functions in question. Unfortunately the nested generic specializations (section 4.2.2) that this would require is not supported by many compilers.

```
enum Partition {
    P_none = 0x00,
    P_horizontal = 0x01,
    P_vertical = 0x02,
    P_block = 0x04,
    P_cube = 0x08,
    P_persistent = 0x10,
    P_fixed = 0x20
};

class Dac {
    ...
    const Partition partition() const
    { return d_rep->partition(); }
    Dac change_type(const Partition p) const
    { return d_rep->change_type(p); }
    const Partition mask(const Partition p) const
    { return Partition(p&0xf); }
...
};

template <class T>
class zMatrix : public zContainer<Matrix<T> > {
    Dac change_type(const Partition p) const {
        Dac x;
        switch (mask(p)) {
          case P_horizontal:
            x = Dac( new zHorizontalMatrix<T>(*this) );
            break;
          case P_vertical:
            x = Dac( new zVerticalMatrix<T>(*this) );
            break;
          default:
            error() << "couldn't change type to:  " << p << fatal;
        }
        return x;
    }
}
```

Figure 5.24: Example partition manipulation

The figure shows how a generic D&C matrix object, of type `zMatrix<T>`*, can be converted into a partitioned matrix object through* `change_type()`*. The argument to* `change_type()` *is an enumerated type describing the desired partition.*

5.7.2 Integrating type changing

As demonstrated in figure 5.21 most common operations can be defined in a general operation function. To expand this to cope with different partitioned states, we make the assumption that partitioned states must be the same for correct operation. This assumption is not entirely correct (see section 5.7.1.1), but will do for many common operations. Type manipulation is then simply a matter of making the partitions match, and can be represented in pseudo-code as shown in figure 5.25.

```
FUNCTION type_coerce()
BEGIN
        IF <right partition is fixed, left partition not fixed>
        THEN
                <change left type to match right>
        ELSE IF <left partition is fixed, right partition not fixed>
        THEN
                <change right type to match left>
        ELSE IF <neither operand is fixed>
        THEN
                <arbitrarily change both types to be the same partition>
        ELSE
                <error, operation is impossible>
END
```

Figure 5.25: Type coercion

The resultant compound D&C object must be wrapped in an evaluation manipulator of the appropriate partition. With template evaluation manipulators the resultant partitioned type needs to be explicitly set. However, by using polymorphic evaluation manipulators, the type of the evaluation manipulator *itself* can be changed dynamically. In fact the manipulator can even have its type modified to a type that has no relation to the original. This is useful for asserting the return type of persistent objects, for example. This, perhaps, is another plus on the side of polymorphic evaluation manipulators. See figure 5.26.

Using polymorphic evaluation manipulators in this fashion gives us insight into a possible way of solving the problem of section 5.7.1.1. We could put the partitioning information of an operation in the evaluation manipulator for that operation. Difficult cases will be dealt with separately by the managing class anyway (cf. `operator*` for `DacMatrix<T>`s), so it is quite feasible to make these functions add the partitioning information as well. Since we are really only interested in the *evaluated* partition of an object, and since that is precisely what an evaluation manipulator represents, it would seem natural to use evaluation manipulators in this way.

5.7.3 Other type changing requirements

We have now described the basics of dynamically specifying partitioned state of D&C objects. This general procedure is the same for `DacVector<T>`s, although greatly simplified by the limited range of possible vector partitions. Ideally we would implement the full

```
temaplte <class T>
DacRep* DacMatrix<T>::op(const DacMatrix<T>& d, const Operation o) const
{
    DacMatrix<T> l=*this,r=d;
/*
 * Change the types of the matrices we are working on if we are able.
 * if either is P_none then make it the type of the other.
 */
    ...
/*
 * now check for delayed-ness and persistent-ness
 */
    DacRep *p;
    if (ptr_cast<zPersistent>(*this)) {
/*
 * stop anything coming back for a delayed matrix
 */
        p = new zEvaluateWrap(
            Dac( new zNull ),
            Dac(
                new zOpTemplate<Matrix<T>,Matrix<T> >(l,r,o)
                )
            );
    } else if (l.partition() & P_vertical) {
        p = new zEvaluateTemplate<zVerticalMatrix<T> > (
            Dac(
                new zOpTemplate<Matrix<T>,Matrix<T> >(l,r,o)
                )
            );
    } else if (l.partition() & P_horizontal) {
        p = new zEvaluateTemplate<zHorizontalMatrix<T> > (
            Dac(
                new zOpTemplate<Matrix<T>,Matrix<T> >(l,r,o)
                )
            );
    } else {
        error() << "parse error" << fatal;
    }

    return p;
}
```

Figure 5.26: Changing the type of simple operations

The figure shows how type coercion integrates into the `DacMatrix<T>` *operator function described earlier. Once the operands have been coerced to appropriate types an operation object is created, and wrapped in an evaluation manipulator which reflects the partitioning of the operands.*

range of algebraic operators, but for now we have sufficient functionality to implement the Kanerva model, and have shown the basic structure of our delayed evaluation, D&C system.

5.8 Kanerva implementation

We have described the Kanerva model and its mathematical representation. We have designed and described D&C classes which implement delayed evaluation, and D&C classes which manipulate these in an outwardly normal fashion. The D&C classes `DacMatrix<T>` and `DacVector<T>` have the same outward functionality, bar persistence, as their standard maths library counterparts. All that remains is to implement the Kanerva model using these classes. Figure 5.27 shows this implementation.

Looking at this implementation we can see that there is very little evidence of parallel evaluation or even D&C. The only clues we have are the fact that the location units and the weight matrix are defined to be persistent D&C objects. The presence of functors also also points towards something slightly non-standard, although the functions themselves would have to be declared anyway[3]. It is only the method of declaration that is slightly strange. Of course the simplicity belies the complexity of the implementation, however, the implementation is designed to be general and flexible so that the complexity need only be dealt with once.

The other thing to note is that because we have adopted a modular and, where possible, parameterized approach, it is simple to create arbitrarily typed `DacVector<T>`s etc. Since, in the Kanerva model, we desire to perform operations on locations with large numbers of bits, it is simple to parameterize the `DacVector<T>`s with an arbitrary number class `BigNum<T>`, rather than from `int` or `long`, say. If the implementation of `BigNum<T>` is sufficiently complete then the only *programmer* changes needed are to the main routine. The application itself will need complete recompilation, however.

We now turn to the performance of the implementation at run-time.

5.9 Results

In examining performance figures for the implementation it is good to remember why we expect reasonable performance. The essence of delayed evaluation is to group together as much distributed computation as possible and so to reduce serial evaluation. Additionally, with D&C, we expect some overhead associated with the D&C implementation, however, by optimizing the computational grain size we hope that this overhead will be small compared to the computation in each grain. On the other hand we desire *balanced* computation so that we make optimal use of the computing resources available to us. This requires that we have enough grains to allow some dynamic load balancing. The question is, which of these factors is most significant for the problem in question.

The results were obtained by running the code on a local area network of Sun Sparc^{TM} workstations. The workstations' capabilities were quite varied although the spread was kept reasonably even. Unfortunately, it proved impossible to run programs with the workstations at zero load. However, this is perhaps a more realistic scenario. The number of locations used was 500,000, the number of bits 32.

The initial results in figure 5.28 are not too promising. We have an observable speedup but it is only for small numbers of processors. Varying the partition makes a significant difference as to whether we get any speedup at all. In observing the system at runtime we notice that one problem is the fixed schedule used for persistent objects. Once the schedule has been run and is fixed there can be no more load balancing for the system,

[3] In C++ one cannot take the address of a built-in operator.

```
FUNCTOR(int,ham,(BigNum<N_BITS> a, BigNum<N_BITS> b))
{
    int s=0;
    for (BigNum<N_BITS> r = a∧b; r≠0; r ≫= 1) s += r&1;
    return s;
}

FUNCTOR(short,leq,(int a, int b))
{ return a ≤ b; }

FUNCTOR(short,gt,(short a, short b))
{ return a > b; }

extern "C" void mainCore(int argc, char ∗argv[])
{
    const n_bits = N_BITS;
    const n_locations = args::size;
    const k_radius = 9;
    const n_outputs = n_bits;
    const n_iters = 16;

    DacMatrix<short> Weights( n_outputs, n_locations );
    DacVector<BigNum<N_BITS> > Locations( n_locations );
    DacVector<short> Outputs( n_outputs );

    Vector<BigNum<N_BITS> > inputs(n_iters), outputs(n_iters);
    Vector<short> output(N_BITS);
        // Initialise structures
    inputs = outputs = Locations = Random;
    Weights = Zero;
        // Convert the weights and locations into persistent objects.
    persist(Locations);
    persist(Weights);

    DacVector<short> Active_locs;
        // store 16 inputs;
    for (int x=0; x<n_iters; x++) {
        Active_locs ≪ mapf(
            mapf(Locations,ham,inputs(x),Resolve<int>()),
            leq,k_radius,Resolve<short>()
            );
            // make up the output vector
        for (int y=0; y<N_BITS; y++) output(y) = inputs(x)[y];
        Outputs = output ∗ short(2) - short(1);
            // Perform the evaluation
        Weights += Outputs ∗ ∼Active_locs;
    }
        // retrieve 16 outputs
    for (x = 0; x< n_iters; x++) {
        BigNum<N_BITS> input=inputs(x);
        Outputs = Weights
                  ∗
                  mapf(
                      mapf(Locations,ham,input,Resolve<int>()),
                      leq,k_radius,Resolve<short>()
                      );

        input = Zero;
        for (int y=0; y < N_BITS; y++) {
            input += (BigNum<N_BITS>((Outputs(y)>0?1:0)) ≪ y);
        }
    }
}
```

Figure 5.27: The D&C Kanerva implementation

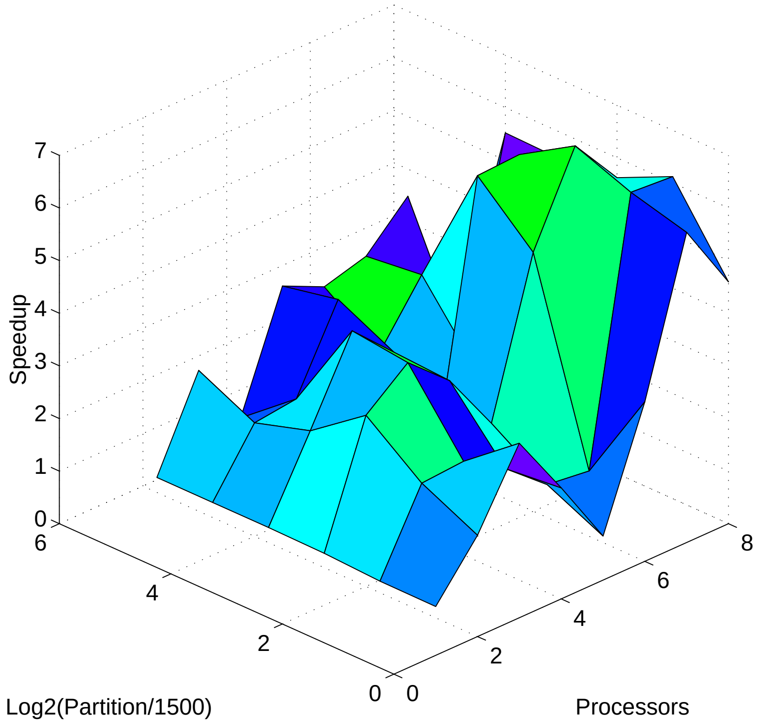


Figure 5.28: Performance results for depth-first Kanerva evaluation

since task placement is fixed. The load balance that was correct for the first D&C pass is often not good at all for the second and subsequent D&C passes. The obvious reason for this is that the computing power available on each node is not constant. Since each processor is a UNIX^{TM} workstation, and is being used by other system and user processes, the computational load on the processor is constantly varying. By increasing the process' priorities we can get better, smoother performance shown in figure 5.29.

The other thing of note is that communication between workstations is slow relative to the processor speed[4]. This means that the cost of offloading work is much greater for workstation clusters than it is for transputer arrays. If offloading work is expensive then the system's ability to load-balance will be impaired. This is because only large grain tasks can be profitably offloaded, so that a small computational window on a remote processor cannot be utilized.

These factors force us to look for better ways of matching the computation we have to do to the computational power available. The factor in our favour is that the task in question has very balanced computational characteristics.

5.9.1 Improving performance

One possible way to look for performance improvement, is to give persistent objects the ability to migrate. This means that instead of fixing task-to-processor allocation, so that cached objects are guaranteed to be available, we allow objects to be fetched from remote processors if the local processor doesn't have the right object. By doing this the

[4]The communication is also relatively unpredictable for the reasons given above.

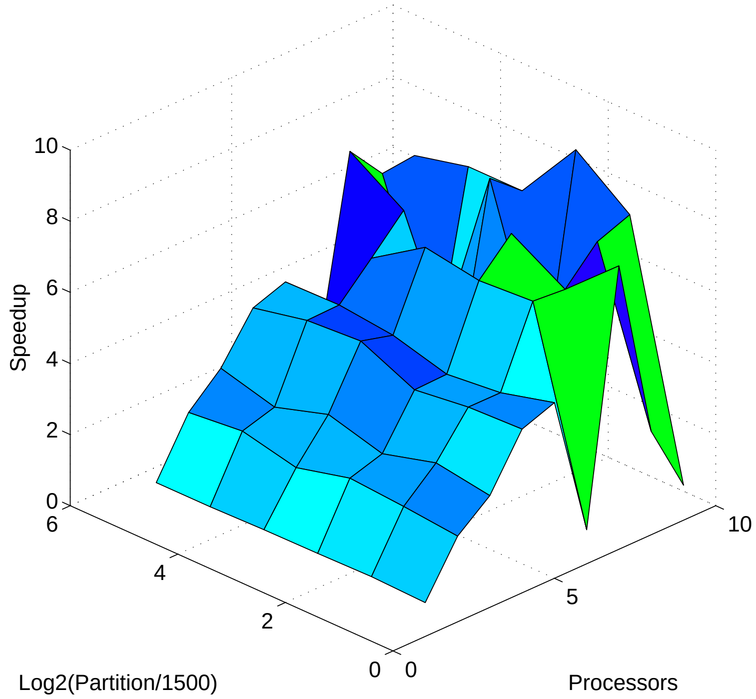


Figure 5.29: Performance results for depth-first Kanerva evaluation

load-balancing could be kept dynamic, hopefully maximising processor utilization. What discourages us from this approach is the low inter-processor communication bandwidth. The varying nature of processor availability would mean that objects will have to migrate quite regularly. Since this migration would be slow, we would expect poor overall performance from the system.

5.9.1.1 Dynamic partitioning

Another possible approach is to vary the partition dynamically. Since the partition appears to be crucial to good performance it would be advisable to try to use its optimal value. Thus the schedule could be fixed as before, but, as the partition is varied objects could be allowed to migrate. Since increasing the partition is equivalent to increasing the computational grain-size, this would effectively prune the fixed schedule of unprofitable offloads. Alternatively, we could try several different fixed schedules and select the best of them. Unfortunately, exploring options in this fashion rather defeats the aim of the exercise which is good *run-time* performance. The only situation in which we might envisage improved run-time performance is for multiple Kanerva storages. In this instance it might be possible to find an optimal partition [91] in a number of cycles much less than the total. If this is the case then the performance gain for the remaining cycles might outweigh the performance loss for the exploration cycles.

5.9.1.2 Matched grains

The easiest approach, however, is to allocate matched grains to processors. Thus for N processors we would generate N grains and schedule those N all at once. On *average* the

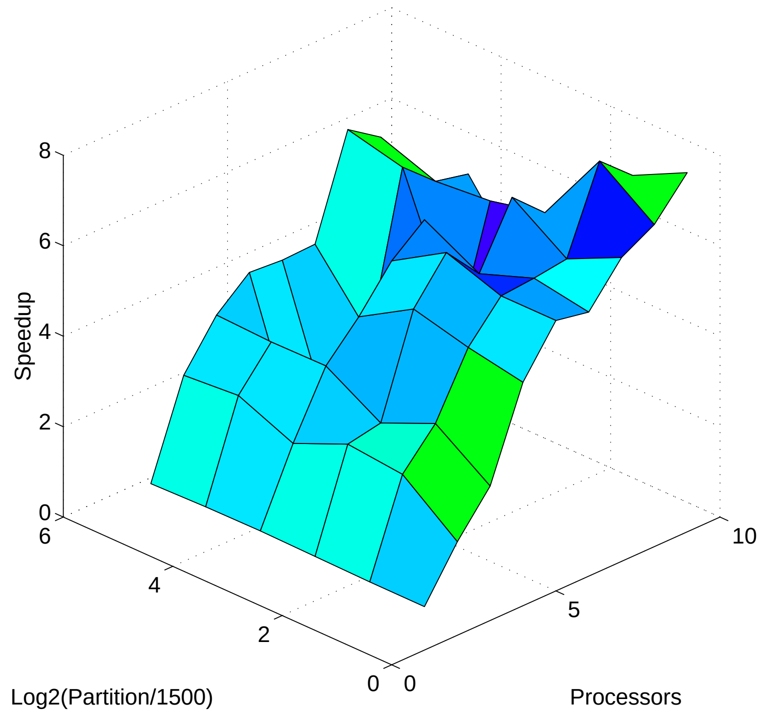


Figure 5.30: Performance results for breadth-first / depth-first Kanerva evaluation

computation will make reasonable use of computing resources since the actual problem is so balanced. For the case where processors have different capabilities, the scheduler could be switched to normal, dynamic operation after an initial allocation of tasks. It is this method that we will try. Actually implementing this sort of task allocation is trivial since that is actually what the breadth-first evaluation of section 3.3.2.2 does. However, instead of specifying a depth to descend to, we calculate the depth based on the number of processors available. This depth is given by $log_2 N_{procs}$. Evaluating the resultant nodes using *depth-first* evaluation yields the subsequent dynamic schedule that we require.

The results are shown in figure 5.30. We can see that the speedups are much more consistent than those for purely depth-first expansion. We note also that good performance is obtained for larger partitions where essentially only breadth-first expansion is used. It is puzzling that we get good performance for small partitions with large numbers of processors. We might explain that it is due to a more uniform load balance, but this does not explain why performance is good also for *large* partitions. The most likely explanation is that we are seeing the results of the combination of *two* types of scheduling. For large partitions breadth-first dominates, for small partitions depth-first is also significant.

It is interesting to see what performance is like for breadth-first only. The results are given in figure 5.31. As we can see, performance is worse than that of the hybrid scheduling, although this may be due to the loads at the time of the machines involved.

5.10 Dynamic expression optimization

In implementing the Kanerva model using a delayed-evaluation D&C system, we have tried to design and use generally applicable system components. However, we have also made use of our knowledge of the computational characteristics of the model, in order

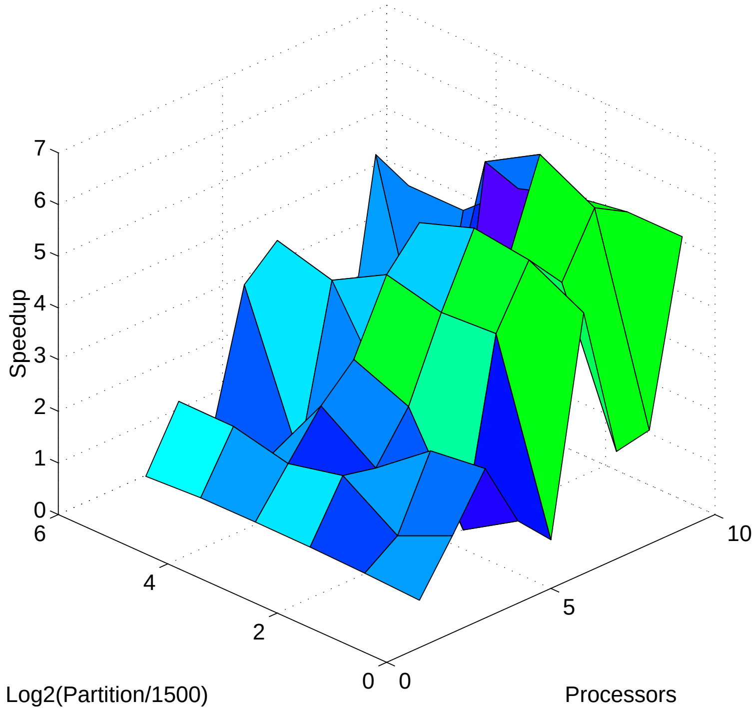


Figure 5.31: Performance results for breadth-first Kanerva evaluation

to make the implementation efficient. It would be ridiculous if the implementation were to not try partitioning the location unit vector. But what of other problems? How can we make sure that the system makes at least some reference to the amount of parallelism available in different partitioning strategies? Fortunately, the balanced nature of algebraic problems like this means that we can approximate quite easily the amount of computation available.

As a start, we can simply make sure that we partition along the longer dimension of matrices. This will not always yield an optimal solution, however, we are only really concerned to catch the case where the dimensions are very disparate, as in the Kanerva model. A more complex approach is to try and *minimize* the number of distributed operations. If we minimize this quantity then we are effectively maximising the amount of distribution, assuming that the total number of operations is constant. Of course, determining the correct partition in this way only applies for a single D&C pass. The problem is more complicated if the computation requires two or more D&C passes, and the 'break point'[5] varies depending on the partition. In this case it might be necessary to minimize the *sum* of D&C pass totals.

5.10.1 Back-propagation revisited

As an example let us revisit the back-propagation of section 2.3. With our original system we were unable to implement this algorithm due to the multitude of different D&C types required. We will try this algorithm again, using the scheme built up for the Kanerva

[5] The point at which one D&C pass terminates and another begins.

model, to see what, if any, changes need to be made. Obviously this is a largely academic exercise since implementing the algorithm in this way is not going to be highly efficient.

```
template <class T> class zTpTemplate : public zEnvelope {
  public:
    zTpTemplate(const Dac& d) : zEnvelope(d) { }

  protected:
    Dac evaluate() const {
        return Container(
            ~(ptr_cast<zContainer<T> >(e_letter.evaluate())->data())
            );
    }

    Dac change_type(const Partition p) const;
    {
        Dac x;
        if (p & P_horizontal) x = zEnvelope::change_type(P_vertical);
        else if (p & P_vertical)x = zEnvelope::change_type(P_horizontal);
        else x = zEnvelope::change_type(p);
        return x;
    }

    const Partition partition() const;
    {
        Partition p = zEnvelope::partition();
        if (p & P_horizontal) p = Partition((p|P_vertical) & ~P_horizontal);
        else if (p & P_vertical)p = Partition((p|P_horizontal) & ~P_vertical);
        return p;
    }
};
```

Figure 5.32: A transposing D&C class

The figure shows a D&C class for transposing its contained object. The template formal is the actual type of the object to be transposed. The transpose operation is done in `evaluate()` *after the operand has been evaluated. Note that* `change_type()` *and* `partition()` *invert the partitions they operate on.*

The first obvious problem is that transposes are not being dealt with intelligently enough. The Kanerva model only requires a single transpose operation, and thus was dealt with specifically inside the vector-vector `operator*`. For the back-propagation equations a variety of transposes are required. Treating each separately as a special case is by definition not generic in application. The answer is to create an envelope class which transposes its letter in `evaluate()`. This is essentially the same as the functionality of `zOpTemplate<L,R>`, except this is for a unary operation. It would be possible to create a general, type-field based class for unary operations. However, the actual usefulness of such a class is not as clear-cut as it was for binary operations.

The transpose class is shown in figure 5.32. The main thing to note is that the letter's partition is inverted, and the returned object from `change_type()` is the opposite of that which was requested.

5.10.1.1 Improving type manipulation

The other problems are all to do with type manipulation. In section 5.7.2 we looked at a simple algorithm for determining the required partition for a pair of D&C objects. There are two problems with this:

- If there is no obvious partition then an arbitrary choice is made. This may not be the best choice.
- If there are partition incompatibilities in the object hierarchy then evaluation will fail.

For the Kanerva model an "arbitrary" choice was made that happened to coincide with the optimal partitioning strategy. However, this is of no use in general. The second item means that it is impossible to evaluate complicated expressions.

To address the first of these issues we modify `type_coerce()`, so that if there is no obvious partition, it will try different partitions to see which yields an object with the largest `size()`. Recall that `size()` is often used to determine whether an object is `simple()` or not. Thus an object with larger `size()` will, on the whole, have greater inherent parallelism. Thus `size()` is a good metric for estimating the computational effort involved in evaluating an object. Note that this quantity is not sufficient for absolute computation comparisons, and would not be sufficient for intelligent scheduling [86]. See figure 5.33.

To address the second issue we introduce a function `sanitize()` which scans a D&C object looking for `zCompounds` in its make-up. We surmise that type incompatibilities will *only* occur in `zCompounds`. Simple container relationships will have only one logical partition, but compound relationships have two, one for each letter. If these two are different then we have a type incompatibility that means the object cannot be evaluated in a single D&C pass. Having found a compound relationship, we recurse down the letters, applying `sanitize()` to each. If the object is still not sane then we evaluate the left letter. Then, if still not sane we evaluate the right letter. The recursion then unwinds. It is interesting to note that we are essentially divide-and-conquering across the object hierarchy rather than across the problem. See figure 5.34. This scheme is similar to that postulated in section 2.2.4.1.

With these changes program 5.35 compiles and runs correctly. The system correctly determines how to partition the data and evaluates expressions when it needs to. When the Kanerva model is run using this regime, the system again correctly picks the partition offering the greatest parallelism (partitioning along the location units). Note that by selecting different intermediate results the correct results are *always* obtained but performance can be worsened or improved. If appropriate intermediate results are made persistent then still further gains can be made. This is essentially the situation we want to be in; where the system always manages to evaluate a problem as best it can, but where a programmer can give hints about better strategies.

5.11 Summary

We have described and developed a D&C class hierarchy for implementing algebraic expressions using delayed evaluation, and have shown that this implementation yields good performance on workstation clusters. We have addressed a number of issues relating to the specifics of this kind of implementation, and done so in a portable and modular way.

```
void type_coerce(Dac& l, Dac& r)
{
    ...
/*
 * If the partition is undefined, find the one which yields maximum size
 */
    if ((l==0 || !l.partition()) && !r.partition()) {
        int s=0, t=0;

        m = P_vertical;
        s = r.change_type(P_vertical).size();

        t = r.change_type(P_horizontal).size();
        if (t > s) {
            m = P_horizontal;
            s = t;
        }

        if (l≠0) {
            t = l.change_type(P_vertical).size();
            if (t > s) {
                m = P_vertical;
                s = t;
            }
            t = l.change_type(P_horizontal).size();
            if (t > s) {
                m = P_horizontal;
                s = t;
            }
        }
    }
/*
 * decompose the left object if necessary
 */
    if (l≠0) {
        l = l.change_type(m);
        if (!sane(l)) l = sanitize(l);
    }
    r = r.change_type(m);
/*
 * decompose the right object if necessary
 */
    if (!sane(r)) r = sanitize(r);
}
```

Figure 5.33: Dynamic partition allocation

The figure demonstrates how D&C operand partitions can be selected intelligently. Successive partitioning strategies are tried, and the one that yields maximum parallelism (largest `size()`*) is selected.*

```
Dac sanitize(const Dac& d)
{
    zCompound *c=ptr_probe<zCompound>(d);
    if (c) {
/*
 * recurse
 */
        if (!sane(c->left)) c->left = sanitize(c->left);
        if (!sane(c->right)) c->right = sanitize(c->right);
/*
 * Both letters are sane but the current object isn't. Therefore
 * evaluate until sanity is obtained.
 */
        if (!sane(d)) {
            c->left = c->left.change_type(c->left.partition());
            c->left = c->left.run();

            if (!sane(d)) {
                c->right = c->right.change_type(c->right.partition());
                c->right = c->right.run();
                if (!sane(d)) error() << "sanitization failure" << fatal;
            }
        }
    }
    else error() << "no compounds to sanitize" << fatal;

    return d;
}
```

Figure 5.34: Sanity checking of D&C objects

The figure demonstrates how compound D&C objects can be forced to have aligned partitions. The compound is scanned for incompatible partition pairs. If one is found then the operands are evaluated and scanning continues.

```
FUNCTOR(double,Fsigmoid,(double d))
{
    return (1.0 / (1.0 + exp(-d)));
}

FUNCTOR(double,Fdsigmoid,(double d))
{
    return ( (1.0 / (1.0 + exp(-d))) * (1.0 - (1.0 / (1.0 + exp(-d)))) );
}

const I=16;
const J=32;
const K=12;

extern "C" void mainCore()
{
    DacVector<double> S_j, S_k, O_j, O_k, d_k, t_k(K), O_i(I);
    DacMatrix<double> W_ij(I,J), W_jk(J,K);

    W_ij = W_jk = t_k = O_i = Random;

    S_j << ~W_ij * O_i;                          // feed forward equations
    O_j << mapf(S_j, Fsigmoid);
    S_k = (~W_jk * O_j);
    O_k = mapf(S_k,Fsigmoid);
                              // feed backward of error equations
    d_k = ((t_k - O_k) & mapf(S_k, Fdsigmoid));
    W_jk += O_j * ~d_k;
    W_ij += (O_i * ~((W_jk * d_k) & mapf(S_j, Fdsigmoid)));
}
```

Figure 5.35: Vertical D&C back-propagation

The figure shows a D&C implementation of the back-propagation algorithm using the classes designed for the Kanerva implementation. As can be seen, D&C and parallelism are not evident at all. Note that expressions are intelligently given good partitioning strategies, but that this is all transparent to the user.

We conclude that delayed evaluation, in conjunction with the techniques that we have described, is a viable implementation strategy for particular classes of parallel problem. We have demonstrated this by applying these techniques to two problems; Kanerva's memory model and the back-propagation algorithm.

Chapter 6

Conclusions and future work

By uniting we stand, by dividing we fall. *John Dickinson*

In this chapter we summarize the contribution and thrust of this thesis. We also examine some future possibilities for the system.

6.1 Summary

A parallel system for implementing D&C problems has been described.

Various aspects of this system have been examined, most notably the associated library, and various problems have been implemented demonstrating the system's ability to achieve useful speedups on real world problems.

The implementing technology we have used has been that of object-orientation. A mapping is made between D&C problems and objects. Having discussed the implementation, it is clear that this technology is not only a useful and viable tool in this context, but that D&C is essentially object-oriented in nature. We assert that although D&C can be described in a process-oriented way, it is more naturally described in object-oriented terms. This is because D&C is invariably concerned with state as well as function. In fact in a parallel context it is usually the abundance of data that leads to long serial execution times. When we talk about the partitioning of parallel problems this invariably means a partitioning of data, and this draws us away from a simple process-oriented description of problems.

Although parallel data-flow languages make a similar assertion (and our description of object-oriented D&C within the actor model in section 3.1.5 is reminiscent of data-flow), object-oriented D&C is not data-flow for complex problems. Different D&C objects have different data *and* process.

However, our system is not just object-oriented data-flow as used by the UFO project [93, 94]. We have shown how the system exhibits object-oriented features that enable the overall issues of complexity to be tackled:

Hierarchical. One of the main advantages of the object-oriented D&C system is its hierarchical structure. This means that complex D&C objects can be built from smaller sub-units. The complexity of the objects is reflected by the complexity of the problems, and enabling the former to be constructed easily similarly aids the latter.

We have constructed systems to compute the Kanerva model and back-propagation algorithm using such a hierarchical approach. We have also demonstrated how delayed evaluation can be used to maintain efficiency in these systems.

Abstraction and polymorphism. Abstraction focuses on the essential characteristics of entities. The system enables a programmer to focus on the D&C aspects of a problem, without regard to evaluation, by presenting entity abstractions encapsulating the solution of standard problems in a D&C fashion. The system enables the evaluation machine to run and schedule D&C problems, without regard to their implementation, by presenting action abstractions [14] which encapsulate all D&C problems. The action abstractions are only possible through polymorphism which enables each D&C object to present the same abstract interface to the evaluator.

Since the evaluator sees all D&C objects in the same way it can evaluate them, using a stack, in an efficient, both in time and space, and generalized fashion. For the same reasons other parallel processing issues, like scheduling and load-balancing, can be addressed in a generic and efficient fashion. The abstractions allow us to really tackle the problems of concern.

Polymorphism is also key to parallel delayed evaluation. It is *only* because each D&C object has a generic interface that aggregate representations can be treated as a single object. And it is only because of this uniformity of interface that methods can be recursively called for aggregate objects.

Encapsulation. Each D&C object encapsulates only state and function that is relevant to its evaluation. Intra-component linkages are much stronger than inter-component linkages, as is required for a stable complex system [14]. The logical separation and encapsulation of D&C program units leads to a more modular system, with a greater chance of operational success. Encapsulation means that individual components can be updated and improved without system failure.

Modularity. The run-time system itself is constructed in a highly modular way, again using object-oriented techniques. This means that the run-time system itself is likely to continue to work well and to be improved easily in the future.

We have presented various classes that tackle general issues of D&C evaluation. Classes that enable objects to be evaluated concurrently. Classes that provide iteration. Classes that cause D&C objects to persist, and so on. Each class puts a little more flexibility and power in the hands of the programmer. The provision of these classes emphasises the point that object-oriented D&C is only one side of the coin. It is the provision of suitable D&C libraries that is also a major strength of the system. The run-time system and basic D&C classes are an implementing technology only, it is the library classes that turn this technology into a useful tool.

Finally, all of these features have been provided without writing a new compiler. This is a point which must not be underestimated. Object-oriented technology is rich in meta-features. Much of the ethos behind object-oriented programming is that new features can be seamlessly provided using the base technology. By adhering to existing standards the system is guaranteed a safe ride through language enhancements, something that cannot be said for a derived language. Where programming tediousness has begun to creep in we have provided a CASE tool for addressing this issue.

We can conclude therefore that object-oriented D&C provides a viable way of achieving parallel performance. Any conclusions about the usability of the system are necessarily subjective, however, our experience leads us to believe that object-oriented D&C provides a convenient and powerful way of expressing, and implementing, parallel programs.

6.2 Contribution

This thesis has taken a new look at the well-known D&C paradigm for parallel processing.

The fundamental concept of the thesis is the encapsulation of D&C problems in objects. This encapsulation allows problems to be evaluated in a generic way.

A stack-based algorithm has been devised for evaluating D&C objects, that draws on conventional computational evaluation. The stack approach allows object evaluations to be nested, and is efficient in both space and time.

The composition of aggregate D&C objects has been demonstrated, and it has been shown how these aggregates can be evaluated efficiently, and used to implement algebraic expressions in parallel.

A mathematical model of D&C has been examined and has been used to demonstrate how problem size scaling is important in D&C programs, just as it is for conventional parallel programs.

The fundamental physical contribution of this work is the Beeblebrox system which provides the described run-time system as well as a rich variety of D&C classes. The system can be ftp'd from `svr-ftp.eng.cam.ac.uk`.

6.3 Future work

We have demonstrated the soundness and viability of D&C objects as a concept. Since the whole system revolves around this idea, it is unlikely that future work would require modification of the basic concept. However, there are two areas where major additions could be made.

The first is obviously the Beeblebrox library. We have described a rich set of Beeblebrox classes, but the number of possibilities is virtually boundless. More fundamental D&C classes could be provided using different underlying data structures such as trees. Additionally, the delayed evaluation subsystem that was used to implement the Kanerva and back-propagation algorithms could be extended to cover further matrix and vector operations. There is also great scope for further optimization of these expressions at run-time so that maximum computation is achieved by every D&C pass.

The second area is that of general run-time efficiency. The scheduler we have employed has proved sufficient for our needs, but there is scope for more intelligent scheduling based on the estimated gain from offloading work. It should be possible to characterize each D&C object with a metric giving its particular computation / communication requirements. This metric could then be used by an intelligent scheduler.

Additionally, there is scope for improving persistent D&C object performance. Persistent objects could be allowed to migrate and again some intelligence could be employed as to when this is a beneficial process.

Finally, D&C partitions could be made dynamic allowing improved performance over time.

Appendix A

Using accepted language extensions

C++ is a changing language. The X3J16 ANSI committee was set up to standardize the language. As a result of this standardization effort a number of new extensions to the language have been proposed and accepted. Many of these extensions are already present in other object-oriented implementations, and their presence would greatly simplify and enhance the design of the D&C system as it now stands. Unfortunately, the standards committee is not expected to have a draft ready until September '94, with an actual standard coming in '95 or '96. Fully conformant compilers can be expected sometime after this. However, it is useful to survey these features and see where they might help.

A.1 Co-variance of virtual return types

In the language's current implementation virtual functions must return the same type for all implementations in an inheritance lattice. This means that if we want to return an object of the current class' type it needs to be of the base-class' type. Co-variance of return types allows virtual functions to return a derived type of the type that the base class function returned. It is possible that this feature might be useful in reducing the need for dynamic casting of D&C objects.

A.2 Run-time type information

Stroustrup [102] has demonstrated a way of providing run-time type information in C++ classes. This feature has now been incorporated into the language standard and will be provided in future compilers. Essentially RTTI is similar to the type identification provided in our D&C classes, however when it is implemented by compiler providers the tedium of setting up RTTI will be eliminated. The term describes exactly what it does: it provides type identification for classes that can be accessed at run-time. This is important for the case where a base class (like our `Dac` class) is used to generically pass information around, but where the actual type of the object needs to be known when any real processing is to be performed on it. Getting the real object from its interface specification goes hand in hand with RTTI and is called *type narrowing*.

A.2.1 Down-casting

As explained in section A.2 it is often desirable to obtain an actual object from its visible interface - usually a base class. However, we cannot just naively cast the base class interface to a derived class. We have to be sure that the object we are casting is really of the type we desire. Then we need to correctly process objects that are in complicated inheritance lattices. We have done this in the past using `ptr_cast<T>`, however, ANSI C++ will provide this feature automatically in the function `dynamic_cast<T>`. `dynamic_cast<T>` is little different from `ptr_cast<T>` except that the template formal is actually a pointer or reference, giving more flexible type selection. Additionally `dynamic_cast<T>` will be a compiler feature rather than implemented in the language itself.

A.3 Summary

Fortunately none of these features involve namespace collisions with D&C features. Thus, although it would be advantageous to use these features, there is no reason why the system as it stands cannot be used with an ANSI conformant compiler.

Appendix B

Hardware and software architecture

The D&C system described in this thesis is only the final layer in a complex system. In this appendix we describe the underlying hardware and software that was used.

B.1 Trollius^{TM}

The cornerstone of the D&C system is the Trollius^{TM} operating system [79, 17]. Trollius^{TM} originated as a transputer operating system for managing process control and message passing. The aim of the Trollius project was to implement UNIX^{TM}-like functionality for transputer-based hardware. Trollius^{TM} transparently provided multi-hop routing, process control, multi-processor file I/O and a whole package of tools.

The other enormous advantage with Trollius^{TM} is that it provides a uniform message-passing interface across different hardware and software architectures. So programs can be compiled for transputers or workstation networks. This provides a convenient way of debugging programs intended for transputers, since the user-friendliness of UNIX^{TM} can be used for development and then programs can be run, unaltered, on transputer hardware.

For transputer-based hardware Trollius^{TM} *is* the operating system. For UNIX^{TM} based machines, Trollius^{TM} is an additional operating system layer.

B.2 Transputer hardware

The parallel hardware used for the experiments in this thesis was a 64-processor, T800 transputer machine. The transputer network was interconnected by configurable link chips that allowed almost all of the combinations possible with the transputer's 4 communication links. The transputer network was connected to a Sun 4/330 workstation via a polling 8-bit link adaptor card. The bandwidth of this card proved rather limiting for some of the experiments in this thesis, and it perhaps would have been useful to try using the 2 Mb dual-port memory that is also provided on the card. Figure B.1 shows schematically the setup used.

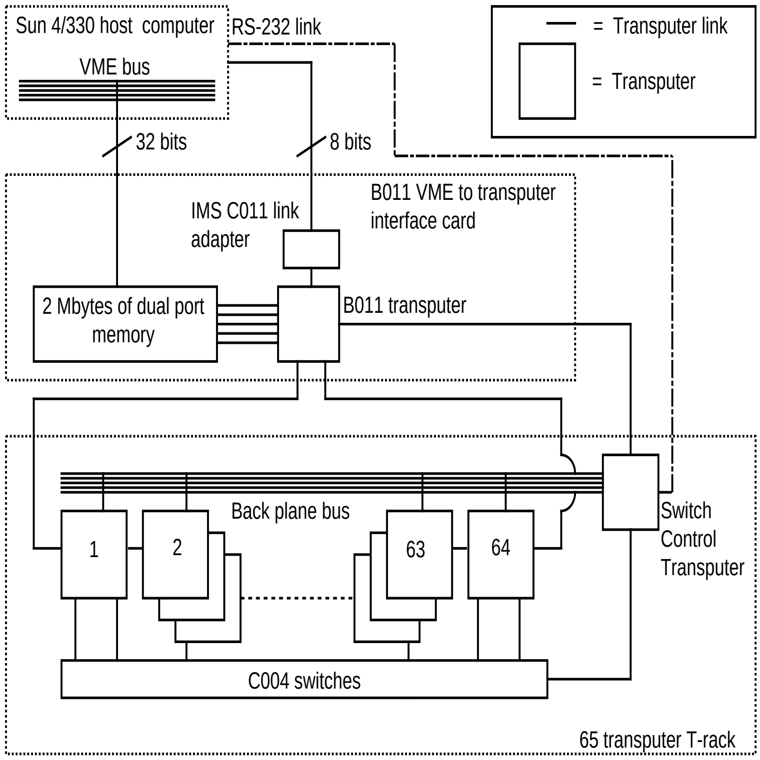


Figure B.1: Transputer farm architecture

B.3 The C++ compiler

The T800 transputer has a rather strange stack-based hardware architecture [57]. As such there is no native C++ compiler for the chip, and a C++ to C translator must be used instead. The compiler used for the purposes of this thesis was the USL "cfront" v3.0.2 compiler. The compiler actually had to be ported slightly to work with the transputer C compiler, but this was not too difficult. A full port was possible, largely due to the flexibility of Trollius^{TM}, so that all the standard C++ classes were available. The same compiler was used for workstation network implementations.

The USL compiler supports many new C++ features including templates, but not exception handling and RTTI. It is hoped that the GNU C++ compiler can be supported in the future, but at the time of writing it did not have the necessary features to make this possible.

B.4 Implementation issues

In this final section we discuss some issues peculiar to a workstation based implementation. On the whole, the workstation implementation is identical to that for transputers, thanks to the generic interface presented by the Trollius^{TM} operating system. However, there are a number of low-level issues that need to be addressed.

B.4.1 Shared memory management in workstation clusters

In section 3.4.3 we described a memory manager for the D&C system. This component managed blocks in shared memory, so that processes could share D&C structures, most notably the execution stack. When using UNIX^{TM} computer clusters, we have a problem since this type of computer has a per-process protected address space. In order to make data available to more than one process we need to use SVR4 shared memory features, which are available on most UNIX^{TM} workstations. This facility allows blocks of shared memory to be allocated and mapped into processes' address space. We can then manage these blocks using a suitable algorithm.

A similar scheme to that given by [63] was used, except that the global operator `new` was overridden with the shared memory system, rather than individual classes' free store allocators. By mapping shared memory blocks contiguously we can imitate the behaviour of `sbrk()`, which normally changes the size of a process' address space. If we have an `sbrk()` equivalent then we can use a conventional memory manager to manipulate the memory blocks. In our case we used the freely available GNU malloc system. Since we have overridden global `new`, we can arrange for the buddy system of section 3.4.3 to allocate its blocks using this operator.

The system is complicated by the need for mutual exclusion of memory accesses. However, this can be provided using SVR4 semaphores. Since we are in a object-oriented environment it is trivial to encapsulate these in a general monitor class. This means that the transputer equivalents can be used on transputers, while still maintaining a uniform interface. In fact very little conditional compilation is needed in switching between target hardware. Most system specific behaviour is encapsulated within modules, and selecting the appropriate memory management is simply a matter of linking in the appropriate libraries.

B.4.1.1 Buddy system extensions

Some UNIX architectures do not allow shared memory segments to be mapped contiguously, this is also true of transputer-based Trollius^{TM}. Instead they are mapped at positions determined by the system. In this case we have a requirement to manage large blocks of non-contiguous memory. Most normal memory managers are inappropriate since they manage one large contiguous address space (section B.4.1). However, the buddy system of section 3.4.3 is ideal. This system manages a large block and allocates other blocks as needed. In the system we described, all the largest blocks are the same size. However, this is no good for general memory management, since we might require a single allocation larger than the largest block. In this case we modify the system, so that when additional blocks are required they are constrained to be of a size no less than the original large block, but can be large enough to satisfy the allocation requirement.

This creates a problem, since it means that the system will be managing blocks of different sizes. When freeing a block we must know the size of its parent, otherwise we might try and coalesce the parent with a non-existent block. In order to do this we keep a record of memory blocks we have allocated and store the index of the appropriate block in an allocation's header record.

Appendix C

Beeblebrox manual pages

NAME

Dac-mode - Beeblebrox formatting functions for emacs

SYNOPSIS

M-x dac-mode

DESCRIPTION

Dac-mode is an emacs lisp mode that provides a series of functions for augmenting C++ class definitions so that they are usable by the Beeblebrox run-time system. *Dac-mode* takes a damage limiting approach and will only add required functions that have no prototypes. If a function prototype exists then *dac-mode* will make no attempt to add a function definition or modify an existing one. The buffer that must be parsed is that containing class declarations. *Dac-mode* assumes that the corresponding source file has the same name but with an extension of "dac-source-ext". This defaults to "C".

dac-mode only works with cc-mode.el and will install command keystrokes into the c++-mode-map keymap. To set up *dac-mode* simply load the library and type M-x dac-mode.

COMMANDS

C-c d f	format the buffer divide-and-conquer style.
C-c d c	format the class which point is in.
C-c d a	format the class which point is in as an abstract class. prefix arg makes it inline.
C-c d m	format the class which point is in as a most derived class requiring make() and a default constructor. prefix arg makes it inline.
C-c d p	parse the class which point is in.
C-c d d	insert a comment delimiter to identify the class members that need to be transmitted for object coherency.
C-c d /	insert a divide() stub.
C-c d +	insert a combine() stub.
C-c d ?	insert a simple() stub.

C-c d = insert an evaluate() stub.

NAME

dac - command line parser for Beeblebrox programs

SYNOPSIS

dac [-rhcPdDv] [-V:cn] [-n <nice>] [-o <file>] [-p <part>] [-b <buff>]
[-w <slowness>] <nodes>

DESCRIPTION

dac is the command line parser for Beeblebrox programs. Arguments are automatically taken from main() and interpreted. Remaining arguments are passed on to mainCore().

The following switches are provided:

-h	Help message.
-P	Pause for a key-press.
-r	This node is the origin.
-c	Cube-up the problem for evaluation.
-Vc	Virtual-circuits on.
-Vn	Virtual-neighbours on.
-d	Debugging mode (no parallel processes).
-D	No Trollius support (implies -d).
-v	Verbose mode. Very verbose with -d.
-n <nice>	<priority> to run process at.
-w <slow>	Delay in seconds for each D&C operation.
-p <size>	<size> partition.
-s <size>	<size> problem.
-b <size>	<size> buffer.
-o <file>	Output timing results to <file>.

Nodes taking part in the evaluation must also be specified as the last argument n<list>, eg., n0-3,5,0xa,12-15.

NAME

Dac - mother of all divide-and-conquer classes.

SYNOPSIS

```
Dac();
Dac(DacRep*);
Dac(const DacRep&);
run() const;
recurse() const;
breadth(int) const;
```

DESCRIPTION

Dac forms the base class for all D&C objects. It is an envelope class in the style of James O. Coplien's "Advanced C++". As an envelope * this class can be copied, assigned, stacked, returned etc. with a minimum of overhead. Protection is provided to make sure DacReps get made into Dacs automatically. The primary function of a *Dac* is to be run and this can be accomplished depth-first (run() and recurse ()) or breadth-first (breadth()). recurse() provides a simple non-parallel evaluation which is useful for debugging.

IMPLEMENTATION NOTES

Dac uses the envelope / letter idiom given in Coplien ch. 5. In this case the letter is derived from the envelope. This means that the envelope can be used to construct letters easily, and also gives a cleaner interface. It throws up a few questions though:

(1) How does the mix-in idea fit into this? ANS: virtually derive *Dac* Rep from Dac saving on an abstract base class. The mix-ins then fit in nicely. To separate these classes still further an intermediate class is added with everything declared pure virtual. This prevents *Dac* functionality from being called accidentally.

(2) How is the letter base kept abstract while allowing envelopes with null letters? ANS: all letter functions are called by envelope functions in a klein bottle type way, Dacs can be instantiated without letters. Letters can be defined as abstract in DacPure.

PUBLIC INTERFACE

```
class Dac : public Buddy {
  public:
    Dac() ;
    Dac(const DacRep&);
    Dac(DacRep* r) ;
    Dac(const Dac& d);
    Dac(ibstream& i) ;
    virtual ~Dac();
    const Dac&            operator= (const Dac& d);
    DacRep*               operator= (DacRep* d);
    const boolean         operator==(const DacRep* d) const ;
    const boolean         operator!=(const DacRep* d) const ;
    const boolean         simple() const;
    Dac                   divide(Dac& l, Dac& r) const;
    Dac                   evaluate() const;
    Dac                   combine(const Dac& d) const;
    const int             size() const;
    void                  size(const int);
    Dac                   make() const;
    Dac                   make_copy(const Dac&) const;
    void                  copy(const DacRep*);
    const Partition       partition() const;
    static const Partition    mask(const Partition p) ;
    Dac                   change_type(const Partition) const;
  public:
    Dac                   run(const boolean=true) const;
    Dac                   recurse() const;
    Dac                   breadth(int=0, const boolean=true) const;
    const DacRep*         rep() const     ;
```

```
    int                node() const;
    const TypeString&  type() const;
    void*              get_this_ptr(int type=0, int probe=0) const;
    void xlate(DacRep* d=0);
  public:
    static DacStack    *p, *q;
    static Zschd       *schd;
  public:              /* protected but for compiler bug */
    union {
        DacRep         *d_rep;
        DacRep         *next;
    };
    int        d_size;
    Dac(DacRep*, BaseConstructor);
    DacRep*    peel() ;
};
```

NAME

DacList - a divide-and-conquer list of divide-and-conquer objects.

SYNOPSIS

```
DacList(const Dac&);
DacList(const Dac&, const Dac&);
DacList DacList::operator+(const Dac&) const
DacList DacList::operator,(const Dac&) const
```

DESCRIPTION

A *DacList* is a special type of D&C list. Each element of the list is in fact a D&C object, and is therefore a candidate for parallel execution. The list is divided and combined in the same way as a zList<T>. However, evaluate() is special in that it arranges for each list element to be run() itself. This is quite a good way of evaluating multiple D&C objects in parallel. The other way to do this is by using a DacPar. However, there is a subtle distinction between the two. A DacPar runs objects as soon as they are available. This means that the farm workers will be totally preoccupied with evaluating the first object. With a DacList, however, the D&C objects are distributed before they are run. This results in better load balancing of the system.

The operators + and , are provided for concatenating DacLists together.

PUBLIC INTERFACE

```
class DacList : public zList<Dac>,
                public zCombineList<Dac>
{
  public:
    DacList();
    DacList(const Dac&);
    DacList(const Dac& a, const Dac& b);
```

```
    DacList(const DacList&);
    const DacList& operator= (const DacList& d) ;
    const Dac& operator= (const Dac& d);
    DacList append(const Dac& d) ;
    DacList operator, (const Dac& d) const ;
    DacList operator+ (const Dac& d) const ;
    Dac     dac(size_t n) ;
    ~DacList();
};
```

NAME

DacManip - a D&C wrapper class for combining manipulators

SYNOPSIS

DacManip(const zManipulator&, const zManipulator&, const zManipulator&, const zManipulator&, const Dac&);
DacManip(const zManipulator&, const zManipulator&, const zManipulator&, const Dac&);
DacManip(const zManipulator&, const zManipulator&, const Dac&);
DacManip(const zManipulator&, const Dac&);

DESCRIPTION

This D&C wrapper enables zManipulators to be combined in a sensible fashion. The constructor detects which manipulator is which and creates a compound object using all of them in the right order. The last argument is the D&C object to be manipulated.

SEE ALSO

zEvaluateManip(3), zCombineManip(3), zSimpleManip(3), zDivideManip(3).

PUBLIC INTERFACE

```
class DacManip : public Dac {
  public:
    DacManip() ;
    DacManip(const Dac& d) ;
    DacManip(const zManipulator&, const zManipulator&,
            const zManipulator&, const zManipulator&,
            const Dac&);
    DacManip(const zManipulator&, const zManipulator&,
            const zManipulator&, const Dac&);
    DacManip(const zManipulator&, const zManipulator&,
            const Dac&);
    DacManip(const zManipulator&, const Dac&);
    ~DacManip();
};
```

NAME

DacMatrix<T> - a D&C matrix class wrapper.

SYNOPSIS

```
DacMatrix(const int, const int);
DacMatrix(const Matrix<T>&);
```

DESCRIPTION

The *DacMatrix* class allows standard matrix operations to be performed, * in parallel, using divide-and-conquer. Many standard matrix operations are provided, although some operations are only partially robust. The package uses delayed evaluation to build up an evaluation tree at run-time. This tree is subsequently modified to determine * how data is going to be partitioned - in the case of matrices, row-wise or column-wise. This transformation is achieved using the change_type() function, after an appropriate type has been determined for the resulting expression. For instance a matrix * vector operation can be partitioned row-wise with the resulting sub-vector units being concatenated together. Alternatively, the partitioning can be column-wise in which case the resultant vectors must be added together. Which makes a better choice is dependent upon the size of the operands in question, and their context in the evaluation structure as a whole.

A *DacMatrix* can be used almost interchangeably with Matrixs, although there are some caveats in their use. In addition, DacMatrixs can be created from Matrixs as well as in more conventional ways.

A *DacMatrix* is strictly a wrapper, as is a Dac. It serves merely to arrange actual D&C objects in a suitable format for processing. As such it means that the internals of evaluation can be kept well away from the user interface.

PUBLIC INTERFACE

```
TEMPL class DacMatrix : public Dac {
 public:
  DacMatrix() ;
  DacMatrix(const int, const int);
  DacMatrix(DacRep* r) ;
  DacMatrix(const Matrix<T>& m);
  DacMatrix(const DacMatrix<T>& m) ;
  ~DacMatrix();
  DacMatrix(DacRep& r) ;
  const abstractType& operator= (const abstractType&);
  const DacMatrix<T>& operator= (const DacMatrix<T>&) ;
  DacMatrix<T> operator* () const;
  const DacMatrix<T>& operator<<(const DacMatrix<T>&) ;
  DacMatrix<T> operator~ () const;
  const DacMatrix<T>& operator+=(const DacMatrix<T>& d) ;
  const DacMatrix<T>& operator-=(const DacMatrix<T>& d) ;
  DacMatrix<T> operator- (const DacMatrix<T>& d) const;
  DacMatrix<T> operator+ (const DacMatrix<T>& d) const;
  DacRep*          op(const DacMatrix<T>&, const Operation) const;
```

```
    const DacMatrix<T>&    op_eq(const DacMatrix<T>&, const Operation);
    DacVector<T>    operator* (const DacVector<T>&) const;
    DacMatrix<T>    operator* (const DacMatrix<T>&) const;
};
```

NAME

DacPar - parallel execution D&C object wrapper

SYNOPSIS

```
DacPar(const Dac&);
DacPar(const DacRep&);
DacPar(DacRep*);
```

DESCRIPTION

A *DacPar* provides a wrapper for DacRep derived objects in the same way that Dac does. However, instantiation of a DacRep object causes the letter part to be pushed onto the evaluation stack and a record kept of the relevant envelope. When one comes to access the letter the envelopes are updated with the evaluated letters. In a parallel * execution environment most letters should already have been evaluated and any remaining ones are evaluated through the use of run(). This means that several independent D&C objects can be evaluated concurrently.

In allowing letters to be pushed onto the execution stack we have to consider a number of possible scenarios in order to ascertain whether all evaluations will complete correctly:

(1) Another object is run() before all DacPars have been evaluated. In this case evaluation will complete correcly as run() save's the current level of the execution stack. No objects can be added to the stack while this run() takes place as it waits for completion before proceeding.

(2) DacPars are evaluated internally by a D&C object. In this case a run() is in progress, however only objects with a DacStack::parallel key will be evaluated, thus confusion on the stack is eliminated. Any run() invoked by *DacPar* evaluation will be a recursed call and the stack state will be saved.

(3) Some combination of (1) & (2). Since DacPars are evaluated only when there are parallel objects at the top of the stack any interleaving of (1) & (2) will unwind correctly.

PUBLIC INTERFACE

```
class DacPar : public Dac {
  public:
    DacPar() ;
    DacPar(const Dac& d)            ;
    DacPar(const DacRep& d)            ;
    DacPar(DacRep* d)            ;
    ~DacPar();
    const Dac& operator=(const Dac& d)        ;
    const DacRep*        rep() const;
};
```

NAME

DacPure - abstract base class for D&C letters

SYNOPSIS

Use as a virtual base for mix-ins.

DESCRIPTION

This class serves to make the primary D&C functions pure again so that users are forced to define them or use mix-ins. This cannot be done in DacRep as it leads to ambiguous usage with the dominance mechanism unable to tell which function to use.

All mix-ins should virtual'ly inherit this class. The C++ compiler will then arrange for the right functions to be called when accessed through the *DacPure* interface.

DacPure inherits from Dac privately so that objects derived from *DacPure* and, more importantly, DacRep *cannot* be up cast to a Dac as this would be disastrous from a reference counted / evaluation point-of-view. Instead a constructor is provided in Dac - 'Dac(DacRep&)' - which wraps the DacRep up to avoid such a disaster.

PUBLIC INTERFACE

```
class DacPure : private Dac {
  public:
    virtual Dac        run(const boolean=true) const=0;
    virtual Dac        recurse() const=0;
    virtual Dac        breadth(int=0, const boolean=true) const=0;
    virtual Dac        make() const;
    virtual void       copy(const DacRep*);
    virtual void*      get_this_ptr(int=0,int=0) const;
    virtual int        node() const=0;
  public:
    Dac::next;
    Dac::d_rep;
    Dac::d_size;
    Dac::mask;
  public:
    Buddy::operator new;
    Buddy::operator delete;
};
```

NAME

DacRep - base class for all letter classes.

SYNOPSIS

```
DacRep(const int size);
DacRep(Exemplar);
```

Use as a base class for all D&C letter classes.

DESCRIPTION

All D&C classes which perform actual evaluation should be derived from this class. DacReps are managed by Dacs to minimise overheads in copying.

Classes which need to be type narrowed using ptr_cast<T>, or classes which are used in a concrete fashion, should define an exemplar and an exemplar constructor the argument of which is passed back to DacReps exemplar constructor. All this and more is performed automatically by dac-mode.el for GNU emacs.

PUBLIC INTERFACE

```
class DacRep : virtual public DacPure {
  public:
    DacRep(const int i);
    DacRep(ibstream&);
    DacRep(const TypeString& e) ;
    ~DacRep();
    const int       size() const;
    void            size(const int);
    Dac             run(const boolean=true) const;
    Dac             recurse() const;
    Dac             breadth(int=0, const boolean=true) const;
    int             node() const;
    obstream&       spawn(obstream&, const boolean tok=true) const;
  public:
    static int      inst_count;
    static void     chk(const DacRep*);
    friend void     chk_list();
  public:
    static struct ExemplarManager ;
    friend class    ExemplarManager;
};
```

NAME

Dinvert - an inversion manipulator

SYNOPSIS

```
Dinvert(const Dac&);
Dac Dinvert::change_type(const Partition) const;
```

DESCRIPTION

Dinvert treats its letter specially for type manipulation. partition() returns the inverse of the letter's partition and change_type() changes the type to the inverse of that which was requested. change_type() is ablative - it no longer has *Dinvert* information.

PUBLIC INTERFACE

```
class Dinvert : public zEnvelope {
  public:
    Dinvert(const Dac& d) ;
    ~Dinvert();
};
```

NAME

car, cdr, cons, () - lisp like functions for Dac objects

SYNOPSIS

```
Dac    car(const Dac&)
Dac    cdr(const Dac&)
Dac    cons(const Dac&, const Dac&)
Dac    operator, (const Dac&, const Dac&)
Dac    operator+ (const Dac&, const Dac&)
```

DESCRIPTION

These functions provide lisp-like functionality for manipulating compound Dac objects. car() and cdr() obtain the left and right members of a compound. cons, '+' and ',' create a compound from two Dac objects.

SEE ALSO

zCompound(3).

NAME

zArray<T> - D&C array class

SYNOPSIS

```
An abstract base class.
provides:    Dac    zArray<T>::divide(Dac&, Dac&) const;
requires:    Dac    zArray<T>::make() const;
```

DESCRIPTION

A *zArray* <T> has a single member of type BuiltinArray<T>, which it arranges to be split up by the divide() function. *zArray* <T> is virtual'ly derived from a zContainer<BuiltinArray<T>> which provides a mix-in interface for accessing the array structure. Thus, it can be used in conjunction with a zMapArray<T> for iterating over the elements of the array.

zArray <T> does not have a built in combine function as this would disallow mixing in the functionality.

PUBLIC INTERFACE

```
TEMPL class zArray : virtual public zContainer<BuiltinArray<T> > {
};
```

NAME

zCombine, zSimple, zUnit, zEvaluate, zDivide.

SYNOPSIS

Mixin with classes requiring a generic operation.

DESCRIPTION

These classes are mix-ins to blank out the various primary functions which a derived class has no need of. They are for convenience only and better performance is achieved by explicitly defining these functions in derived classes. Note the the operation of zUnit is already provided by default in DacRep.

PUBLIC INTERFACE

```
class zCombine : virtual public DacPure
{
  public:
    zCombine() ;
    ~zCombine();
};
```

NAME

zCombineArray<T> - a D&C mix-in array combiner.

SYNOPSIS

A mix-in class.
provides: Dac zCombArray::combine(const Dac&) const;

DESCRIPTION

A *zCombineArray*<T> provides a combine function that concatenates two zContainer<BuiltinArray<T>>s together. It relies on the operator| provided by Array<T>. This functionality is crucial for array related divide-and-conquer evaluations. Unfortunately, the current implementation merely creates a new array and copies the previous two in. This has performance penalties which could be avoided if some way to construct arrays contiguously was found.

PUBLIC INTERFACE

```
TEMPL class zCombineArray : virtual public zContainer<BuiltinArray<T> > {
};
```

NAME

zCombiner - general mix-in D&C combining class

SYNOPSIS

```
requires:     Dac zCombiner::combine(const Dac&) const;
```

DESCRIPTION

zCombiner is a generic combining class. All operations have been done up until this class is used, thus there is no need of simple(), divide() evaluate(). *zCombiner* is mixed in with a class providing the combine() fucntion and is primarily used for returning Dac object results.

PUBLIC INTERFACE

```
class zCombiner : public DacRep
{
  public:
    zCombiner() ;
    ~zCombiner();
};
```

NAME

zCompound - a D&C container class for two D&C letters.

SYNOPSIS

```
zCompound(const Dac&, const Dac&);
provides:     Dac zCompound::divide(Dac&, Dac&) const;
Dac zCompound::combine(const Dac&) const;
```

```
Dac zCompound::evaluate() const;
const boolean zCompound::simple() const;
```

DESCRIPTION

zCompound is a class which combines two D&C objects to form a compound D&C object. In *zCompound* all the primary D&C functions are delegated to the zCompound's members 'left' and 'right'. Compound objects can be made from normal D&C objects through the use of the operator+. Note that compound zCompounds can be formed by making 'left' or 'right' a

zCompound as manipulated by DacMatrix(3) and DacVector(3).

SEE ALSO

car(3), DacMatrix(3), DacVector(3).

PUBLIC INTERFACE

```
class zCompound : public DacRep
{
  public:
    zCompound();
    zCompound(const Dac&, const Dac&);
    ~zCompound();
};
```

NAME

zContainer<T> - a general D&C container for standard classes.

SYNOPSIS

```
zContainer<Class>(const Class&);
```

DESCRIPTION

zContainer is a general template container class that implements object I/O for the contained object. It may well be more efficient to use the contained class directly, but this is provided for aesthetics. *zContainer* is used extensively by derived operation classes like zVector(3).

PUBLIC INTERFACE

```
TEMPL class zContainer : public zNull {
  public:
    typedef zContainer<T>   Container_t;
    zContainer(const T& t) ;
    zContainer(const Dac&);
```

```
    zContainer();
    ~zContainer();
    virtual T          data() const;
    void               init(const T& x) ;
};
```

NAME

zDelayHorizontalMatrix<T> - a horizontally divided delayed assignment D&C matrix class.

SYNOPSIS

As zDelayMatrix<T> but providing:
Dac zDelayHorizontalMatrix<T>::divide(const Dac&, const Dac&) const;

DESCRIPTION

A *zDelayHorizontalMatrix* <T> provides a delayed assignment matrix class like a zDelayMatrix<T>, except that the class can actually be instantiated. The class is defined to partition horizontally in divide(). *zDelayHorizontalMatrix* <T>s are usually created by change_type() applied to a zDelayMatrix<T>.

SEE ALSO

zDelayMatrix(3).

PUBLIC INTERFACE

```
TEMPL class zDelayHorizontalMatrix : public zDelayMatrix<T> {
  public:
    zDelayHorizontalMatrix(const char* f) ;
    zDelayHorizontalMatrix(const int r, const int c, const abstractType& a) ;
    zDelayHorizontalMatrix(const int r, const int c) ;
    ~zDelayHorizontalMatrix();
};
```

NAME

zDelayMatrix<T> - a delayed assignment D&C matrix class.

SYNOPSIS

Used as a base class for zDelayVerticalMatrix<T> etc.
zDelayMatrix(const char*);
zDelayMatrix(const int rows, const int columns);
zDelayMatrix(const int rows, const int columns, const abstractType&);

provides: Dac zDelayMatrix<T>::evaluate() const;
Dac zDelayMatrix<T>::change_type() const;

DESCRIPTION

A *zDelayMatrix* <T> contains a matrix size and initialization description which is realized when evaluate() is called. A

zDelayMatrix <T> can be initialised from a file name, or with size arguments. It can also be initialized with size arguments and an abstractType. *zDelayMatrix* <T> only provides the functionality for initialization and realization. In order to divide the class in a meaningful way, subclasses must be defined with this functionality. See zDelayVerticalMatrix(3) and zDelayHorizontalMatrix(3).

zDelayMatrix <T> provides a change_type() function that allows derived classes to be generated according to a partitioning requirement. See DacMatrix(3) for examples.

PUBLIC INTERFACE

```
TEMPL class zDelayMatrix : public zNull {
  public:
    zDelayMatrix(const char*);
    zDelayMatrix(const int r, const int c, const abstractType& a) ;
    zDelayMatrix(const int r=0, const int c=0) ;
    ~zDelayMatrix();
};
```

NAME

zDelayVector<T> - a delayed assignment D&C vector class.

SYNOPSIS

zDelayVector(const char*);
zDelayVector(const int elems);
zDelayVector(const int elems, const abstractType&);
provides: Dac zDelayVector<T>::evaluate() const;
Dac zDelayVector<T>::divide(const Dac&, const Dac&) const;

DESCRIPTION

A *zDelayVector* <T> contains a vector size and initialization description which is realized when evaluate() is called. A

zDelayVector <T> can be initialised from a file name, or with size arguments. It can also be initialised with size arguments and an abstractType. *zDelayVector* <T> divides its vector up in much the same way as zVector<T> does - although the vector doesn't actually exist untill evaluate is called. *zDelayVector* <T>s can be generated by DdelayVector<T>s which are used bu the DacVector(3) wrapper.

SEE ALSO

DacVector(3).

PUBLIC INTERFACE

```
TEMPL class zDelayVector : public DacRep, private zCombine {
  public:
    zDelayVector(const char*);
    zDelayVector(const int i, const abstractType& a) ;
    zDelayVector(const int i=0) ;
    ~zDelayVector();
};
```

NAME

zDelayVerticalMatrix<T> - a vertically divided delayed assignment D&C matrix class.

SYNOPSIS

```
As zDelayMatrix<T> but providing:
Dac     zDelayVerticalMatrix<T>::divide(const Dac&, const Dac&) const;
```

DESCRIPTION

A *zDelayVerticalMatrix*<T> provides a delayed assignment matrix class like a zDelayMatrix<T>, except that the class can actually be instantiated. The class is defined to partition vertically in divide(). *zDelayVerticalMatrix* <T>s are usually created by change_type() applied to a zDelayMatrix<T>.

SEE ALSO

zDelayMatrix(3).

PUBLIC INTERFACE

```
TEMPL class zDelayVerticalMatrix : public zDelayMatrix<T> {
  public:
    zDelayVerticalMatrix(const char* f) ;
    zDelayVerticalMatrix(const int r, const int c, const abstractType& a) ;
    zDelayVerticalMatrix(const int r, const int c) ;
    ~zDelayVerticalMatrix();
};
```

NAME

zEnvelope - a concrete D&C wrapper class for D&C objects.

SYNOPSIS

```
zEnvelope(const Dac&);
provides:       Dac zCompound::divide(Dac&, Dac&) const;
Dac zCompound::combine(const Dac&) const;
Dac zCompound::evaluate() const;
const boolean zCompound::simple() const;
```

DESCRIPTION

zEnvelope is a class which holds a single D&C object. All the primary D&C functions are delegated to this letter. Since the primary functions are all inline, this doesn't prove to much of an efficiency problem. zEnvelopes are used extensively for dynamic inheritance.

PUBLIC INTERFACE

```
class zEnvelope : public DacRep
{
  public:
    zEnvelope(const int=0);
    zEnvelope(const Dac&);
    ~zEnvelope();
};
```

NAME

zEvTemplate<T> - a parameterized evaluation manipulator.

SYNOPSIS

```
zEvTemplate<T>(const Dac&);
provides:       Dac zEvTemplate<T>::evaluate() const;
```

DESCRIPTION

A template D&C zEnvelope that transforms its letter to its formal parameter upon evaluation.

SEE ALSO

zEvWrap(3).

PUBLIC INTERFACE

```
TEMPL class zEvTemplate : public zEnvelope {
  public:
    zEvTemplate(const Dac& d) ;
    ~zEvTemplate();
```

```
};
```

NAME

zEvaluateManip, zDivideManip, zCombineManip, zSimpleManip - D&C manipulators for dynamic inheritance.

SYNOPSIS

```
DEFEVAL(name)   { ... }
zEvaluateManip(void(*)(Dac&,const Dac&));
Dac     zEvaluateManip::create(const Dac&) const;
DEFDIV(name)    { ... }
zDivideManip(Dac(*)(const Dac&,Dac&,Dac&));
Dac     zDivideManip::create(const Dac&) const;
DEFCOMB(name)   { ... }
zCombineManip(void(*)(Dac&,const Dac&,const Dac&));
Dac     zCombineManip::create(const Dac&) const;
DEFSIMP(name)   { ... }
zSimpleManip(const boolean(*)());
Dac     zSimpleManip::create(const Dac&) const;
```

DESCRIPTION

These D&C classes enable class hierarchies to be built up with dynamic inheritance. Each class works by having a D&C letter to which all functions, except the one being defined, are forwarded. The normal use of these objects is through the macros DEFEVAL(), DEFDIV(), DEFCOMB(), DEFSIMP(). These macros provide parameters labelled result, self, left and right depending on the function. An object of the required type is automatically created and labelled <name>. This object can be used to create more objects of the same type. It is generally used by DacManip(3). Function definitions should use ptr_probe<T> for accessing members of result etc. This is because objects may be nested arbitrarily deeply, and the user will not be able to tell at which level objects may be found.

These classes are good for rapid prototyping of functionality, however, they may not be as efficient as a properly defined D&C object.

PUBLIC INTERFACE

```
class zEvaluateManip : public zManipulator
{
  public:
    zEvaluateManip(const PFD dp) ;
    zEvaluateManip(const PFD dp, const Dac& d) ;
    ~zEvaluateManip();
    Dac         create(const Dac& d) const ;
};
```

NAME

zEvaluateWrap - a polymorphic D&C evaluation manipulator.

SYNOPSIS

```
zEvaluateWrap(const Dac& w, const Dac& d);
provides:      Dac zEvaluateWrap::evaluate() const;
```

DESCRIPTION

A *zEvaluateWrap* wraps the return value of its argument d.evaluate() in its argument w. This functionality is the polymorphic equivalent of zEvaluateTemplate(3). The wrapper is reference counted since only one is required per processor. The wrapper itself is not actually used, it merely manufactures the required object. change_type() is arranged to operate on the wrapper as well as the letter.

PUBLIC INTERFACE

```
class zEvaluateWrap : public zEnvelope
{
  public:
    zEvaluateWrap(const Dac& w, const Dac& d) ;
    ~zEvaluateWrap();
};
```

NAME

zGraphic - a D&C visualization class.

SYNOPSIS

```
zGraphic(const Dac&);
```

DESCRIPTION

A *zGraphic* is a zEnvelope which contains its construction argument and delegates all primary functions to this argument. However, at the same time *zGraphic* arranges for the current execution state to be displayed in the form of a tree. divide() expands this tree and combine() prunes it.

Each processor has a window in which nodes are displayed. Nodes are represented by black boxes. When a node is spawned to another processor, the box becomes hollow. When the node is returned, the box is made solid once more. Remote nodes are positioned according to their position in the overall tree. This makes it easier to determine where a node has been spawned to.

Since there is no physical tree it is impossible for zGraphic to react to expose events. The window must be visible all of the time for the tree to be displayed correctly. Execution can be paused by clicking mouse button 2 in a window.

PUBLIC INTERFACE

```
class zGraphic : public zEnvelope
{
  public:
    zGraphic();
    zGraphic(const Dac&);
    ~zGraphic();
};
```

NAME

zHorizontalMatrix<T> - a horizontal D&C matrix class.

SYNOPSIS

```
zHorizontalMatrix(const Matrix<T>&);
zHorizontalMatrix(const char*);
provides:     Dac zHorizontalMatrix<T>::divide(Dac&, Dac&) const;
Dac zHorizontalMatrix<T>::combine(Dac&) const;
```

DESCRIPTION

A *zHorizontalMatrix* <T> has a single member of type Matrix<T>, which it arranges to be split up by the divide() function.

zHorizontalMatrix <T> is derived from a zMatrix<T> which provides a mix-in interface for accessing the matrix structure. An exemplar is provided for dynamic casting with ptr_cast<T> Note that several possible division strategies are possible for matrices, these should be provided by mix-in type classes

PUBLIC INTERFACE

```
TEMPL class zHorizontalMatrix : public zMatrix<T> {
  public:
    zHorizontalMatrix(const Matrix<T>& a) ;
    zHorizontalMatrix(const Dac& d);
    zHorizontalMatrix(const char* fn);
    ~zHorizontalMatrix();
  public:
    static const Dac     exemplar;
};
```

NAME

zIterator - a D&C parallel iteration class.

SYNOPSIS

Abstract base class.
provides: Dac zIterator::iterate(int from, int to);
requires: Dac zIterator::mapfunc(int) const=0;

DESCRIPTION

A *zIterator* enables parallel iteration using D&C. mapfunc(), which must be defined in derived classes, is executed for each value in the range FROM to TO. evaluate() returns a Dac object, the type of which is determined by the return value of mapfunc().

PUBLIC INTERFACE

```
class zIterator : public DacRep, public zUnit {
  public:
    zIterator(int sa=0, int stop=0) ;
    ~zIterator();
    Dac        iterate(int, int);
};
```

NAME

zMapArray<T> - a mix-in D&C class for iterating over an zArray<T>.

SYNOPSIS

Mix-in with classes of the zArray<T> family.
provides: Dac zMapArray<T>::evaluate() const;
requires: Dac zMapArray<T>::mapfunc(const T&)=0

DESCRIPTION

This class can be used to iterate over the elements of a zArray(3) of any type. It requires derived classes to define the virtual function mapfunc(), which it arranges to be called for all elements of the zArray<T>. No evaluate() function is necessary and mapfunc() must return a Dac object like evaluate() would have done.

PUBLIC INTERFACE

```
TEMPL class zMapArray : virtual public zContainer<BuiltinArray<T> >
{
};
```

NAME

zMapf<T,F>, mapf() - map over a vector's elements applying a function.

SYNOPSIS

```
zMapf<F,T>(const DacVector<T>& v, const F(*)(T));
zMapf<F,T>(const DacVector<T>& v, const F(*)(T,T), T);
DacVector<T> mapf(const DacVector<T>& d, T (*p)(T));
DacVector<T> mapf(const DacVector<T>& d, T (*p)(T,T), T);
DacVector<R> mapf(const DacVector<T>& d, R (*p)(T), Resolve<R>);
DacVector<R> mapf(const DacVector<T>& d, R (*p)(T,T), T, Resolve<R>);
provides:       Dac zMapf<T,F>::evaluate() const;
```

DESCRIPTION

zMapf <T,F> provides a mechanism for applying a function, in parallel, to all the elements of a zVector<T>. The function applied can take 0 or 1 arguments, and can return an arbitrary type. Global template functions are provided for easy construction of *zMapf* <T,F>'s from zVector<T>'s. If the resultant zVector<T> is of a different type to the evaluated

zMapf <T,F> then a type resolving parameter must be included when using the global functions. The template formals T and F represent the type of the vector being mapped, and the type of the vector returned, respectively.

Note that all functions used must be delared as Functors so that their definitions can be found portably.

COMMENTARY

The problem with this class is the handling of functions with differing numbers of arguments. The requirement is to be able to use standard functions so <stdarg.h> cannot be used. As a compromise we hold the number of arguments we need. If we made this a template formal then the code should be compiled a lot more optimally. Another consideration must be that the actual information held is common to every instantiation by the D&C machine - thus reference counting would take out a lot of the tedium.

SEE ALSO

DacVector(3).

PUBLIC INTERFACE

```
template <class T, class F> class zMapf : public zEnvelope {
  public:
    zMapf(const DacVector<T>& v, const PFT p);
    zMapf(const DacVector<T>& v, const PFTT p, T a);
    ~zMapf();
};
```

NAME

zMatrix<T> - generic D&C matrix class.

SYNOPSIS

A delayed evaluation base class.
provides: Dac zMatrix<T>::change_type() const;

DESCRIPTION

This class purely exists for change_type() to work properly - it should never actually be evaluated. It is used extensively by DacMatrix(3) for instantiating D&C matrix objects the partition of which is determined later. A *zMatrix* <T> will change to either a zHorizontalMatrix(3) or a zVerticalMatrix(3).

PUBLIC INTERFACE

```
TEMPL class zMatrix : public zContainer<Matrix<T> > {
};
```

NAME

zNull - a NOP D&C class.

SYNOPSIS

```
zNull(const int=1);
zNull(const Dac&);
NullDac;
```

DESCRIPTION

zNull is a concrete D&C class that doesn't actually do anything. It is most useful where a D&C class needs to be returned from a function but for which the value will be ignored. It is also useful as a base class for classes that merely provide additional storage. *zNull* can be dynamically cast to, transmitted, received and used in most places where a D&C class is required. A constructor from a Dac is provided so that a Dac class may be converted into nothing. A NullDac is a *zNull* allocated in heap with a Dac wrapper.

PUBLIC INTERFACE

```
class zNull : public DacRep
{
  public:
    zNull(const int =1);
    zNull(const Dac&);
    zNull(const TypeString&);
```

```
    ~zNull();
};
```

NAME

zOpTemplate<L,R> - a concrete D&C general operation class.

SYNOPSIS

```
zOpTemplate<L,R>(const Dac&, const Dac&, const Operation);
provides:       operators +,-,*,+=,-=,*= in evaluate.
```

DESCRIPTION

zOpTemplate <L,R> is a Dac class which provides D&C equivalents for the operators +,-,*,+=,-=,*=. The template formals are the operands for the required operation. This class can be used in conjunction with zEvaluateTemplate(3) to change the return type of the evaluation. In general this is required as evaluate simply returns a zContainer<L> by default. So for example a zEvaluateTemplate<zVerticalMatrix<T> > might be used to concatenate resultant zContainer<Matrix<T> >s into a result. Note that for delayed data types it may be necessary to use a zEvaluateTemplate<zNull>. *zOpTemplate* <L,R>s are used extensively by DacMatrix(3)s and DacVector(3)s.

SEE ALSO

DacMatrix(3), DacVector(3).

PUBLIC INTERFACE

```
class zOpTemplate : public zCompound {
  public:
    zOpTemplate() ;
    zOpTemplate(const Dac& l, const Dac& r, const Operation);
    ~zOpTemplate();
};
```

NAME

zPartition - a partition manipulation D&C mix-in class.

SYNOPSIS

```
zPartition(int);
provides:       const boolean zPartition::simple() const;
```

DESCRIPTION

zPartition arranges for its construction argument to be used as the partition for D&C evaluation of the object it is a part of. The partition defaults to args::partition which can be provided on the command-line. The partition data is transmitted, so it only needs to * be set on the origin processor.

PUBLIC INTERFACE

```
class zPartition : virtual public DacPure
{
  public:
    zPartition(int p=args;;
    ~zPartition();
};
```

NAME

zPersistent - a persistent D&C wrapper.

SYNOPSIS

```
zPersistent(const Dac& d);
const Dac& persist(Dac&);
const Dac& flush(Dac&);
```

DESCRIPTION

A *zPersistent* makes its D&C argument d persistent. Persistence is implemented by caching persistent D&C objects in a hash table when evaluate() is called for the first time. The D&C object is then replaced with a null object. When evaluate() is called again the required D&C object is retrieved from the hash table. Notice that evaluate() is usually called when D&C objects have been distributed, and so the persistent objects will be distributed as well. Persistent, virtual D&C objects are a good way of preventing very large evaluations on the root processor.

zPersistent objects require special handling by the D&C scheduler so that scheduled objects are always scheduled consistently. Note that this may well imply a performance penalty. Persistent objects are registered on the root node for the first call of divide().

PUBLIC INTERFACE

```
class zPersistent : public zEnvelope {
  public:
    zPersistent() ;
    zPersistent(const Dac& d) ;
    ~zPersistent();
    void real() ;
    void flush() ;
    void init(unsigned int o, int b) ;
```

```
    const Partition     lookup_partition() const;
  public:
    static Dac  exemplar;
};
```

NAME

zQuicksort<T> - a D&C quicksort class.

SYNOPSIS

```
zQuicksort(const BuiltinArray<T>&);
provides:       Dac zQuicksort<T>::divide(Dac& l, Dac& r) const;
Dac zQuicksort<T>::evaluate() const;
```

DESCRIPTION

A *zQuicksort* <T> is a D&C object for quicksorting a zArray(3). The efficiency of the implementation is limited by the the efficiency of zCombArray<T>.

PUBLIC INTERFACE

```
TEMPL class zQuicksort : public zArray<T>, public zCombineArray<T>
{
  public:
    zQuicksort(const BuiltinArray<T>& a)   ;
    const Dac& operator=(const Dac& d) ;
};
```

NAME

zSum<T> - a D&C object summator.

SYNOPSIS

```
zSum(const T&);
provides:       Dac zSum<T>::combine(const Dac&) const;
```

DESCRIPTION

zSum is a D&C containet class which sums its member in combine(). The member comes from *zSum* <T>'s superclass, zContainer<T>. This assumes that its member actually has a + operator.

PUBLIC INTERFACE

```
TEMPL class zSum : public zContainer<T>
{
  public:
    zSum(const T& t) ;
    zSum(const Dac& d) ;
    ~zSum();
  public:
    static const Dac    exemplar;
};
```

NAME

zTemplate, zCombineTemplate, zEvaluateTemplate - a generic D&C hierarchy.

SYNOPSIS

```
zTemplate<Class>(const Class&);
```

DESCRIPTION

These classes provide a class hierarchy for performing general D&C on a standard C++ class. This class is used as the formal parameter for these parameterized classes. Most functionality is provided in the base class zBaseTemplate (an alias to zContainer). The other classes give an outline structure, the user must provide specializations for the functions divide() and combine(). Note that only g++ will handle general template specializations which are more useful in this instance.

The derived classes zCombineTemplate and zEvaluateTemplate are designed to be used as mix-in classes.

PUBLIC INTERFACE

```
TEMPL class zTemplate : virtual public zBaseTemplate<T> {
  public:
    zTemplate(const T& x) ;
    zTemplate(const zTemplate<T>&) ;
    ~zTemplate();
    T*          operator-> () ;
};
```

NAME

zTpTemplate<T> - a transpose D&C class.

SYNOPSIS

```
zTpTemplate(const Dac&);
provides:      Dac zTpTemplate<T>::evaluate() const;
```

DESCRIPTION

zTpTemplate <T> arranges for its member to be transposed and returned in a container when evaluate()ed. It is used in conjunction with DacMatrix<T> and DacVector<T>. The template formal is the type of the object to be transposed.

zTpTemplate <T> treats its letter specially for type manipulation. partition() returns the inverse of the letter's partition and change_type() changes the type to the inverse of that which was requested.

SEE ALSO

DacMatrix(3), DacVector(3).

PUBLIC INTERFACE

```
TEMPL class zTpTemplate : public zEnvelope {
  public:
    zTpTemplate(const Dac& d) ;
    ~zTpTemplate();
  public:
    static Dac  exemplar;
};
```

NAME

zVector<T> - a D&C vector class.

SYNOPSIS

```
zVector(const Vector<T>&);
provides:      Dac zVector<T>::evaluate() const ;
Dac zVector<T>::divide(Dac& l, Dac& r) const;
Dac zVector<T>::combine(const Dac& r) const;
```

DESCRIPTION

A *zVector* <T> provides containership for Vector<T> objects. It also provides D&C functions for splitting and combining

zVector <T> objects. It is very similar to zArray(3) in operation but is specifically for maths oriented processing.

SEE ALSO

zArray(3).

PUBLIC INTERFACE

```
TEMPL class zVector : /* virtual */ public zContainer<Vector<T> >
{
  public:
    zVector() ;
    zVector(const Vector<T>& v) ;
    zVector(const Dac&);
    ~zVector();
    static const Dac exemplar;
};
```

NAME

zVerticalMatrix<T> - a D&C matrix class that partitions vertically.

SYNOPSIS

```
zVerticalMatrix(const Matrix<T>&);
zVerticalMatrix(const char* fn);
```

DESCRIPTION

A *zVerticalMatrix* <T> has a single member of type Matrix<T>, which it arranges to be split up columnwise by the divide() function. *zVerticalMatrix* <T> is derived from a zMatrix<T> which provides a mix-in interface for accessing the matrix structure. An exemplar is provided for dynamic casting with ptr_cast<T>. A *zVerticalMatrix* will recombine columnwise although generally evaluate() returns an object of differing type, as defined by the programmer.

For matrix operations see DacMatrix(3). zVerticalMatrix's are normally generated indirectly by the change_type() function from DacMatrix<T>s or DacVector<T>s.

PUBLIC INTERFACE

```
TEMPL class zVerticalMatrix : public zMatrix<T> {
  public:
    zVerticalMatrix(const Matrix<T>& m);
    zVerticalMatrix(const Dac& d);
    zVerticalMatrix(const char* fn);
    ~zVerticalMatrix();
  public:
    static const Dac    exemplar;
};
```

NAME

zVirtualArray<T> - a D&C delayed evaluation array class.

SYNOPSIS

An abstract base class containing a virtual array.
provides: Dac zVirtualArray<T>::divide(Dac&, Dac&) const;
requires: Dac zVirtualArray<T>::make() const;

DESCRIPTION

A *zVirtualArray* <T> is derived from zArray<T> and so inherits the functionality of that class. However, it differs from the zArray<T> * class in that the array data is not instantiated until the array() function is called. In the meantime, a filename, position and size are used to represent the array.

SEE ALSO

zArray(3).

PUBLIC INTERFACE

```
TEMPL class zVirtualArray : public zArray<T> {
  public:
    BuiltinArray<T> realise();
    BuiltinArray<T> data() const;
};
```

Bibliography

[1] D. Adams. *The Hitch-hiker's Guide to the Galaxy.* Pan, 1979.

[2] Gul A. Agha. *Actors.* MIT Press, 1986.

[3] A. V. Aho, J. E. Hopcraft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[4] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading Divide-and-Conquer: A technique for Designing Parallel Algorithms. *SIAM Journal of Computing*, 18(3):499–532, June 1989.

[5] T. H. Axford. An Elementary Langauge Construct for Parallel Programming. *ACM Sigplan Notices*, 25(7):72–80, 1990.

[6] T. H. Axford. An Abstract Model for Parallel Programming. Technical Report CSR-91-5, The University of Birmingham School of Computer Science, 1991.

[7] T. H. Axford. Position Paper: Using a general Divide-and-Conquer Function as a Basic Mechanism for Parallelism. In *Workshop on Abstract Machine Models for Highly Parallel Computers*, volume 2, pages 95–99. Univeristy of Leeds, 1991.

[8] F. Baiardi, M. Jazayeri, M. Mackey, F. Petrini, T. Sullivan, and M. Vanneschi. P^3M: An Abstract Architecture for Massively Parallel Machines. In *Workshop on Abstract Machine Models for Highly Parallel Computers*, volume 2. Univeristy of Leeds, 1991.

[9] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[10] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Computer Science. Prentice-Hall International, 1990.

[11] Brian N. Bershad et al. PRESTO: A System for Object-Oriented Parallel Programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[12] C. R. Birchenhall. Matrix Methods in C++. Discussion Paper in Computing CP105, University of Manchester, 1990.

[13] G. Blair, J. Gallagher, D. Hutchison, and D. Shepherd, editors. *Object-Oriented Languages, Systems and Applications.* Pitman, 1991.

[14] Grady Booch. *Object-Oriented Design with Applications.* Benjamin / Cummings, 1991.

[15] P. A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke. μuC++: Concurrency in the Object-oriented Language C++. *Software - Practice and Experience*, 22(2):137–172, February 1992.

[16] G. D. Burns. A Local Area Multicomputer. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[17] Greg Burns et al. All About Trollius. *Occam User's Group Newsletter*, 1990.

[18] F. W. Burton. Storage Management in Virtual Tree Machines. *IEEE Transactions on Computers*, 37(3), March 1988.

[19] F. W. Burton and M. M. Huntbach. Virtual Tree Machines. *IEEE Transactions on Computers*, 33(3):278–280, March 1984.

[20] B. Carpentieri and G. Mou. Compile-Time Transformations and Optimization of Parallel Divide-and-Conquer Algorithms. *ACM SIGPLAN Notices*, 26(10):19–28, October 1991.

[21] P. P. S. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), March 1976.

[22] A. A. Chien. Supporting Modularity in Highly-Parallel Programs. In *Research Directions in Object-Based Concurrent Systems*. MIT Press, 1993.

[23] A. A. Chien and W. J. Dally. Concurrent Aggregates (CA). In *Second Symposium on Principles and Practice of Parallel Programming*. ACM, March 1990.

[24] A. A. Chien and W. J. Dally. Experience with Concurrent Aggregates (CA): Implementation and Programming. In *Fifth Ditributed Memory Computer Conference*, April 1990.

[25] A. A. Chien, W. Feng, V. Karamcheti, and J. Plevyak. Techniques for Efficient Execution of Fine-Grained Concurrent Programs. In *Yale Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, August 1992.

[26] Andrew Chien, Vijay Karamcheti, and John Plevyak. The Concert System - Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. Technical report, University of Illinois, 1993.

[27] Roger S. Chien and Samuel T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[28] P. A. Chou. Optimal Partitioning for Classification and Regression Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4):340–354, April 1991.

[29] A. R. Clare. A Generic Divide-and-Conquer Kernel for the Meiko Computing Surface. Unpublished technical report, 1990.

[30] A. R. Clare and A. M. Day. Experiments in the Parallel Computation of 3D Convex Hulls. Unpublished paper, August 1992.

[31] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[32] M. Cosnard and A. D. Ferreira. Reflexions on the Real Power of the Hypercube Model. In *Workshop on Abstract Machine Models for Highly Parallel Computers*, volume 2. Univeristy of Leeds, 1991.

[33] R. B. Davies. Newmat04, An Experimental Matrix Package In C++. Documentation of the Newmat C++ class library, 1991.

[34] E. W. Dijkstra. Solution of a Problem in Concurrent Programming. *Communications of the ACM*, 8:569–570, September 1965.

[35] Raimund K. Edge. *Programming in an Object-Oriented Environment*. Academic Press, 1991.

[36] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[37] M. J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[38] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall International, Englewood Cliffs, NJ, 1988.

[39] N. Gehani and A. D. McGettrick. *Concurrent Programming*. Addison-Wesley, 1989.

[40] N. H. Gehani. Broadcasting Sequential Processes. In N. Gehani and A. D. McGettrick, editors, *Concurrent Programming*. Addison-Wesley, 1989.

[41] G. A. Geist and V. S. Sunderam. Network-Based Concurrent-Computing on the PVM System. *Concurrency - Practice and Experience*, 4(4):293–311, 1992.

[42] D. Gill and E. Tadmor. An $O(N^2)$ Method for Computing the Eigensystem of $N \times N$ Symmetric Tridiagonal Matrices by the Divide and Conquer Approach. *SIAM Journal of Statistical Computing*, 11(1):161–173, January 1990.

[43] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA, 1989.

[44] Andrew S. Grimshaw. The Mentat Run-Time System: Support for Medium Grain Parallel Computation. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 1064–1073. Computer Society Press, 1990.

[45] Andrew S. Grimshaw. An Introduction to Parallel Object-Oriented Programming with Mentat. Technical Report TR-91-07, Department of Computer Science, University of Virginia, 1991.

[46] Andrew S. Grimshaw. Meta Systems: An Approach Combining Parallel Processing and Heterogeneous Distributed Computing Systems. In *International Parallel Processing Symposium*, Beverly Hills CA, 1992.

[47] Andrew S. Grimshaw and Virgilio E. Vivas Jr. FALCON: A Distributed Scheduler for MIMD Architectures. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 149–163, Atlanta, GA, March 1991.

[48] Andrew S. Grimshaw and Edmond C. Loyot. ELFS: Object-Oriented Extensible File Systems. Technical Report TR-91-14, Department of Computer Science, University of Virginia, July 1991.

[49] D. H. Grit and J. R. McGraw. Programming Divide and Conquer for a MIMD Machine. *Software-Practice and Experience*, 15(1):41–53, January 1985.

[50] A. Gupta. Stanford Dash Multiprocessor - The Hardware and Software Approach. *Lecture Notes in Copmuter Science*, 605:801–805, 1992.

[51] R. Gupta. A Hierarchical Approach to Load Balancing in Distributed Systems. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 1000–1005. Computer Society Press, 1990.

[52] J. L. Gustafason. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[53] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

[54] J. Hendler. Enhancement for Multiple Inheritance. In P. Wegner and B. Shriver, editors, *Proceedings if the Object-Oriented Programming Workshop*, volume 21 of *ACM SIGPLAN Notices*, October 1986.

[55] W. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[56] C. A. Hoare. Communicating Sequential Processes. In N. Gehani and A. D. McGettrick, editors, *Concurrent Programming.* Addison-Wesley, 1989.

[57] M. Homewood, D. May, D. Shepherd, and R. Shepherd. The IMS T800 Transputer. *IEEE Micro*, 7(5):10–26, October 1987.

[58] E. Horowitz and A. Zorat. Divide-and-Conquer for Parallel Processing. *IEEE Transactions on Computers*, 36(6):582–585, 1983.

[59] P. Hudak and B. Goldberg. Serial Combinators: "Optimal" Grains of Parallelism. In de Bakker et al., editors, *PARLE.* Springer-Verlag LNCS, Vol. 258, 1987.

[60] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[61] K. Ikudome, G. C. Fox, A. Kolawa, and J. W. Flower. An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 1105–. Computer Society Press, 1990.

[62] A. K. Jones and P. Schwartz. Experience Using Multiprocessor Systems – A Status Report. *Computing Surveys*, 12(2):121–165, 1980.

[63] David Jordan. Instantiation of C++ Objects in Shared Memory. *Journal of Object-Oriented Programming*, pages 21–28, March 1991.

[64] Pentti Kanerva. *Self-Propagating Search: A Unified Theory of Memory.* PhD thesis, Center for the Study of Language and Information, 1984.

[65] Vijay Karamcheti and Andrew Chien. Concert - Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. In *Supercomputing '93*, Portland, Oregon, November 1993.

[66] D. King and E. J. Wegman. Hypercube Dynamic Load Balancing. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 962–964. Computer Society Press, 1990.

[67] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, 2nd edition, 1973.

[68] T. J. LeBlanc and E. P. Markatos. Shared Memory vs. Message Passing in Shared-Memory Multiprocessors. In *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing.* IEEE Computer Society Press, December 1992.

[69] T. G. Lewis and H. El-Rewini. *Introduction to Parallel Computing.* Prentice-Hall, 1992.

[70] J. Lim and R. E. Johnson. The Heart of Object-Oriented Concurrent Programming. *SIGPLAN Notices*, 24(4), April 1989.

[71] Barbara Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 23(5), 1988.

[72] D. L. McBurney and M. R. Sleep. Transputer-Based Experiments with the ZAPP Architecture. In de Bakker et al., editors, *PARLE.* Springer-Verlag LNCS, Vol. 258, 1987.

[73] D. L. McBurney and M. R. Sleep. Transputers + Virtual Tree Kernel = Real Speedups. In G. C. Fox, editor, *The Fourth DMCC.* Association for Computing Machinery, 1988.

[74] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[75] J. del R. Millán and P. Bofill. Learning by Back-Propogation: Computing in a Systolic Way. In de Bakker et al., editors, *PARLE.* Springer-Verlag LNCS, Vol. 365, 1989.

[76] Z. G. Mou. Divacon: A Parallel Language for Scientific Computing Based on Divide-and-Conquer. In *1990 Third Symposium On The Frontiers Of Massively Parallel Computation*, pages 451–461, 1990.

[77] Z. G. Mou and P. Hudak. An Algebraic Model for Divide-and-Conquer and Its Parallelism. *The Journal of Supercomputing*, (2):257–278, 1988.

[78] NeXT Computer Inc. *The Objective C Language*, 3.0 edition, 1992.

[79] The Ohio State University. *Trollius User's Reference*, 1990.

[80] J. Padget and S. Merrall. Bridging the MIMD - SIMD Gap. In *Workshop on Abstract Machine Models for Highly Parallel Computers*, volume 1, pages 83–. Univeristy of Leeds, 1991.

[81] A. J. Piper and R. W. Prager. A High-Level, Object-Oriented Approach to Divide-and-Conquer. Technical Report CUED/F-INFENG/TR 98, Cambridge University Engineering Department, 1992.

[82] A. J. Piper and R. W. Prager. A High-Level, Object-Oriented Approach to Divide-and-Conquer. In *The Fourth IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, December 1992.

[83] A. J. Piper and R. W. Prager. Generalized Parallel Programming with Divide-and-Conquer: The Beeblebrox System. Technical Report CUED/F-INFENG/TR 132, Cambridge University Engineering Department, July 1993.

[84] John Plevyak, Vijay Karamcheti, and Andrew Chien. Analysis of Dynamic Structures for Efficient Parallel Execution. In *Sixth Workshop on Languages and Compilers for Parallel Machines*, August 1993.

[85] R. W. Prager and F. Fallside. The Modified Kanerva Model for Automatic Speech Recognition. *Computer Speech and Language*, (3):61–81, 1989.

[86] F. A. Rabhi. *Designing a Parallel Graph Reduction Machine with Dynamic Granularity Control*. PhD thesis, University of Sheffield, Department of Computer Science, 1991.

[87] F. A. Rabhi and G. A. Manson. Experimenting with Divide-and-Conquer Algorithms on a Parallel Graph Reduction Machine. Technical Report CS-90-2, University of Sheffield, Department of Computer Science, 1990.

[88] F. A. Rabhi and G. A. Manson. Divide-and-Conquer and Parallel Graph Reduction. *Parallel Computing*, (17):189–205, 1991.

[89] M. Rosing and R. P. Weaver. Mapping Data to Processors in Distributed memory Computations. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 884–. Computer Society Press, 1990.

[90] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Internal Representations by Error Propogation. In Anderson and Rosenfield, editors, *Neurocomputing*. MIT Press, 1988.

[91] A. Saha and M. D. Wagh. Algorithms for Determining Optimal Partitions in Parallel Divide-and-Conquer Computations. In *1991 International Conference on Parallel Processing*, volume 3, pages 75–82, 1991.

[92] V. A. Saletore. A Distributed and Adaptive Dynamic Load Balancing Scheme for Parallel Processing of Medium-Grain Tasks. In D. W. Walker and Q. F. Stout, editors, *The Fifth Distributed Memory Computing Conference*, pages 994–999. Computer Society Press, 1990.

[93] John Sargent. United Functions and Objects: An Overview. Technical Report UMCS-93-1-4, Department of Computer Science University of Manchester, 1993.

[94] John Sargent. Uniting Functional and Object-Oriented programming. Technical report, Department of Computer Science University of Manchester, 1994.

[95] Jerry Schwartz. Iostreams Examples. In *C++ Language System Release 3.0 Library Manual*. Unix Systems Laboratories, 1991.

[96] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[97] D. R. Smith. Top-Down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27:43–96, 1985.

[98] I. Sommerville. *Software Engineering*. International Computer Science Series. Addison-Wesley, 1985.

[99] Guy L. Steele Jr. *Common Lisp The Language*. Digital Press, 2nd edition, 1990.

[100] V. Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13:354–356, 1969.

[101] B. Stroustrup. Parameterized types for C++. In *C++ Selected Readings*. Unix Systems Laboratories, 1991.

[102] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2nd edition, 1992.

[103] B. Stroustrup and D. Lenkov. Run-time Type Identification in C++. In *USENIX C++ Proceedings*, pages 313–339. USENIX Association, 1992.

[104] B. Stroustrup and J. E. Shopiro. *A Set of C++ Classes for Co-routine Style Programming*.

[105] Thinking Machines Corporation, Cambridge, MA. *C* Programming Guide Version 6.0*, November 1990.

[106] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *SIGPLAN Notices*, 22(12), December 1987.

[107] David Ungar, Randall B. Smith, and Craig Chambers. Object, Message and Performance: How they Coexist in Self. *IEEE Computer*, 25(10):53–64, 1992.

[108] Nick Waegner and Steve Young. A Trellis-Based Language Model for Speech Recognition. In *Proceedings of the 2nd International Conference on Spoken Langauge Processing*, 1992.