CHAPTER 2

The Divide-and-Conquer Paradigm as a Basis for Parallel Language Design

Tom Axford

University of Birmingham, Birmingham B15 2TT, U.K.

Published in *Advances in Parallel Algorithms*, L. Kronsjö and D. Shumsheruddin (eds.), Blackwell 1992.

1 Introduction

1.1 A Brief History

Since the earliest days of computer programming, algorithms have been known which follow the so-called 'divide-and-conquer' method. Many of the best and most widely used computer algorithms are these divide-and-conquer (d-c) algorithms. The binary search algorithm and Hoare's quicksort are two very well known examples. There are many, many others for all types of computing problems, both simple and complicated, both general and specialised.

Earlier references to d-c in the programming literature do not attempt to formalise it (e.g. Aho et al. 1974), but simply use the idea informally to help with the explanation of an algorithm. In (Horowitz and Sahni 1978), on the other hand, d-c is described as a *control abstraction* and a program for it is given in a Pascal-like pseudocode, calling four other procedures to supply the details needed to implement any specific algorithm within the general d-c family. Although Horowitz and Sahni gave this completely general d-c program, they never thereafer used it as a program, but simply as a model to be copied when constructing programs for more specific d-c algorithms. More recently, many other authors have followed a similar approach and formally defined the d-c paradigm as an algorithm which can be discussed in its own right, rather than considering only particular instances of it.

D-c algorithms have long been recognised as highly suitable for parallel implementation because the sub-problems generated can be solved independently and hence in parallel. Many d-c algorithms generate very large numbers of sub-problems in typical real applications and hence can potentially be implemented with a high degree of parallelism. There have been various proposals for parallel architectures designed specifically for the d-c family of algorithms (Peters 1981; Preparata and Vuillemin 1981; Horowitz and Zorat 1983; McBurney and Sleep 1987) and the suggestion of basing a parallel programming language on the d-c paradigm appears to have first been made over ten years ago (Preparata and Vuillemin 1981), although the first, and so far only, language to have been developed in detail appeared quite recently (Mou 1990). Much of the recent work on using the general d-c paradigm for parallel processing has been done in the context of functional programming languages (Mou and Hudak 1988; Cole 1989; Kelly 1989; Mou 1990; Rabhi and Manson 1990), which are perceived to have some fundamental advantages over conventional procedural languages. In an influential and well known paper (Backus 1978), John Backus argued strongly in favour of functional languages instead of the conventional procedural languages, and these arguments

have been echoed many times by more recent authors (e.g. Hudak 1989; Hughes 1989). The main reasons for preferring functional languages are discussed later.

1.2 The Divide-and-Conquer Paradigm

The general d-c paradigm may be described informally as follows: partition the problem into separate, smaller sub-problems, all of which are essentially the same but with different data; find solutions to these sub-problems; and then combine these solutions into a solution for the whole. This approach is used recursively, so that the sub-problems may themselves be further partitioned into smaller sub-problems and so on, until the sub-problems obtained are so small that a solution to each is easy to find directly.

In a Pascal-like pseudocode, the general d-c paradigm may be defined more formally as:

```
procedure divcon(p)
begin
    if simple(p)
    then
        return solve(p)
    else
        (p1,...,pn) := divide(p);
        return combine(divcon(p1),...,divcon(pn))
    fi
end
```

where *p* is the problem data and *p1,...,pn* are the problem data for each of the sub-problems into which the problem is partitioned. The function *simple* tests to see if the problem is sufficiently simple that it can be solved directly (by the function *solve*) in which case the partitioning need not be done. The function *divide* partitions the problem into *n* smaller problems, while the function *combine* takes the solutions to the *n* sub-problems and combines them into a solution to the whole problem.

2 Divide-and-Conquer in Procedural Languages

2.1 What is Wrong With Existing Languages?

Procedural languages such as Fortran, Pascal, C, Ada, etc. are well-established and highly successful vehicles for programming conventional computers based on the von Neumann machine architecture, which is an essentially sequential, uniprocessor architecture. The only really common use of parallel processing is for a few very specialised and heavily used functions of computer operating systems for which dedicated hardware is built (e.g. peripheral device controllers for handling input and output in parallel with other processing).

General purpose parallel computer systems have been readily available for at least a decade now. It is widely appreciated that these computers offer far superior price/performance ratios than conventional uniprocessor machines, yet they are used far less. As Geoffrey Fox has said:

> "... essentially all successful reasonable performance uses of parallel machines have used explicit user decomposition which is low level and machine dependent. We expect that we must find more portable attractive methods if parallel computers are to take over from the conventional architectures" (Fox 1989)

And on the same theme, D.B. Skillicorn says:

> "It is scarcely surprising that most potential users have avoided making the transition to parallel machines and development of parallel software. ... The solution to these problems lies in finding concurrent programming languages or programming models that are architecture independent." (Skillicorn 1991)

Many others have expressed similar views (e.g. Pancake, 1991). Indeed, it is widely recognised that conventional programming languages have proved disappointingly unsuitable for automatic parallelisation, and that new languages are needed. Attempts to extend existing languages have been highly architecture dependent, and they also fail to solve the problem that the underlying language is fundamentally sequential in nature. This means that programs must use the (machine-dependent) language extensions to achieve worthwhile parallel speedups. Existing programs (entirely within the base language) fail to benefit unless they are substantially redesigned and rewritten.

Why are conventional languages so difficult to implement efficiently in parallel? A major reason for this is the type of control structures used in these languages. In essence, the control structures can usually all be reduced to three fundamental types: (i) simple sequencing of statements (denoted by the semicolon in Pascal), (ii) the **if...then...else...** construct, and (iii) the **while...do...** construct.[1] There are no obvious opportunities for parallelism in any of these. The compiler is totally unable to extract any parallelism from a program by simply looking at its overall control structure, it must inevitably look deeper (and that is very much more difficult to do).

2.2 A Divide-and-Conquer Construct

In this section we define a control structure that implements the d-c paradigm. This control structure can be used as an alternative to the usual **while...do...** construct (or similar loop constructs). Any computable function can be programmed in a conventional procedural language using only three control structures: (i) sequencing, (ii) conditional statements (**if...then...else...**) and (iii) d-c statements (i.e. the new construct). A proof of this statement is not given here, although it is not difficult to construct a proof: essentially all that is required is to show that any instance of the old **while...do...** statement can be programmed in terms of the new d-c construct (of course, some further assumptions are needed about the programming language, but nothing out of the usual).

[1] Indeed, the **if...then...else...** construct is itself unnecessary, as it is easy to represent it in terms of the other two. That is going to extremes, however, as no real programming language limits the programmer so severely. We wish to consider the simplest set of constructs that allow us to retain the essential flavour and style of most conventional programs.

2.2.1 Definition of the Divide-and-Conquer Construct

A general d-c algorithm could be defined in the form of a procedure as in Section 1.2. This requires four user-defined procedures to specify the details. A less cumbersome way to do it is to define a new control construct, in a similar vein to the traditional control constructs such as **if, for, while, case,** etc. The d-c construct that we define here has five parts to it, separated and bounded by six keywords: **divcon, test, solve, ordivide, andcombine** and **nocvid** (the last terminates the complete construct and is **divcon** written backwards).

The general form is:

divcon $P_1\{v_1, ..., v_m/\}$
test $B\{v_1, ..., v_m\}$
solve $P_2\{w_1, ..., w_n/v_1, ..., v_m\}$
ordivide $P_3\{v'_1, ..., v'_m, v''_1, ..., v''_m/v_1, ..., v_m\}$
andcombine $P_4\{w_1, ..., w_n/w'_1, ..., w'_n, w''_1, ..., w''_n\}$
nocvid

where $P_i\{x, ..., y/u, ..., v\}$ denotes any piece of program (consisting of complete statements and constructs only) that computes the variables $x, ..., y$ from the variables $u, ..., v$ (and from constants and any other variables, such as global variables, that are in scope at the beginning of the **divcon** construct); and $B\{x, ..., y\}$ denotes any boolean expression using the variables $x, ..., y$ (and constants and global variables).

It is important that no side effects are permitted within the d-c construct; i.e. the code for *B* does not update any variables at all (whether local or global) and the code for $P_i\{x, ..., y/u, ..., v\}$ does not update any variables other than the $x, ..., y$ explicitly mentioned.

The meaning of this construct is defined by the equivalent program:

procedure *divcon(***in** $v_1, ..., v_m$, **out** $w_1, ...w_n$*)*
begin
  if $B\{v_1, ..., v_m\}$ **then** $P_2\{w_1, ..., w_n/v_1, ..., v_m\}$
  else
    $P_3\{v'_1, ..., v'_m, v''_1, ..., v''_m/v_1, ..., v_m\}$*;*
    divcon($v'_1, ..., v'_m, w'_1, ..., w'_n$*);*
    divcon($v''_1, ..., v''_m, w''_1, ..., w''_n$*);*
    $P_4\{w_1, ..., w_n/w'_1, ..., w'_n, w''_1, ..., w''_n\}$*;*
  fi
end*;*
$P_1\{v_1, ..., v_m/\}$*;*
divcon($v_1, ..., v_m, w_1, ..., w_n$*);*

where the two recursive calls of *divcon* (between P_3 and P_4) can be taken in either order (or concurrently).

So, the construct

divcon P_1 **test** B **solve** P_2 **ordivide** P_3 **andcombine** P_4 **nocvid**

can be understood as follows:

P_1 is code to declare and initialise the variables $v_1, ..., v_m$ that will be used in the first phase of the d-c method. B is a conditional expression (in terms of $v_1, ..., v_m$). If this expression is true, the problem will not be sub-divided further, but a solution found directly using P_2, which is the code to compute the result (represented by the variables $w_1, ..., w_n$) directly from $v_1, ..., v_m$. If the conditional expression B is false, however, the problem is sub-divided into two parts using P_3, which is code to compute $v'_1, ..., v'_m$ and $v''_1, ..., v''_m$ from $v_1, ..., v_m$. The singly primed variables represent one sub-problem, while the doubly primed variables represent the second sub-problem. The two sub-problems are themselves solved using the same d-c approach applied recursively. The results of these two sub-problems will be represented by $w'_1, ..., w'_n$ and $w''_1, ..., w''_n$ respectively. The final result is computed by P_4, which is code to compute $w_1, ..., w_n$ from $w'_1, ..., w'_n$ and $w''_1, ..., w''_n$, i.e. to combine the solutions to the two sub-problems.

Two very simple examples serve to illustrate the style of programming required by the **divcon** construct. Both examples are problems that would normally be programmed very easily with a single loop using **for** or **while** loops.

2.2.2 Example: Summation

A program to add up an array of numbers (i.e. $a[1] + ... + a[n]$) can be written as a single application of the d-c construct, in which the variables i and j denote the first and last index values of the array to be summed (initially *1* and *n*), and *sum* denotes the sum of the array (the result of the computation):

```
divcon
    i := 1;
    j := n;
test
    i = j
solve
    sum := a[i]
ordivide
    i' := i;
    j' := (i+j) DIV 2;
    i'' := j' + 1;
    j'' := j
andcombine
    sum := sum' + sum''
nocvid
```

The first part (after **divcon**) defines the initial values of the variables *i* and *j* which will be used as working variables in the d-c process. The next part (after **test**) is a boolean expression that tests if the problem is sufficiently simple that it can be solved directly without the need for partitioning. In this example, no partitioning is done if the array contains only one element.

The third part (after **solve**) solves the problem in the case that no partitioning is to be carried out. This occurs only if the array contains just one element, so the sum is that element. The new variable *sum* is defined in terms of *i* and *j* (and any constants or globals, such as the array *a* which is treated as a global variable as far as the d-c construct is concerned), and *sum*

will eventually contain the final result, i.e. the sum of all elements of the array. The fourth part (after **ordivide**) divides the problem into exactly two sub-problems: i', j', i'' and j'' are computed from *i* and *j*. The pair (i', j') is the data for one sub-problem (summing the bottom half of the array), while (i'', j'') is the data for the other sub-problem (summing the top half of the array).

The final part (after **andcombine**) combines the results of the two sub-problems to give the result of the whole problem: *sum* is computed from *sum′* and *sum″*, which denote the results of the two sub-problems. In this example, the sum of the whole array is simply the sum of the sums of the two halves of the array.

2.2.3 Another Example: Find the First Zero

A program to find the first zero in the array *a[1..n]* can also be implemented as a single simple application of the d-c construct. The variables *i* and *j* denote the first and last index values, as before. The result is denoted by two variables: *found* is true if a zero element is found, and false if all elements of the array are non-zero; *posn* is the index of the first zero element if there is one (*posn* is undefined if all elements are non-zero).

divcon
    i := 1;
    j := n;
test
    i = j
solve
    found := (a[i] = 0);
    posn := i;
ordivide
    i' *:= i;*
    j' *:= (i+j) DIV 2;*
    i'' *:=* j' *+ 1;*
    j'' *:= j*
andcombine
    found := found′ **or** *found″;*
    posn := **if** *found′* **then** *posn′* **else** *posn″* **fi**
nocvid

This program works by dividing the array into two halves and solving for each half separately. If the array contains only one element, however, no sub-division is necessary and the result can be computed directly simply by looking at the only element to see if it is zero. The solutions to the two halves of the array are combined by (i) taking the logical OR of the two values of *found*, and (ii) if a zero was found in the first half then it is the first zero in the whole array, otherwise the zero found in the second half is the first zero of the whole array.

2.2.4 Discussion

To the author's knowledge, no programming language with this d-c construct (or anything very similar) has ever been implemented. A purely sequential implementation of the d-c

construct is likely to be less efficient than equivalent loop constructs, although not seriously so. This is due to the somewhat greater overheads associated with the d-c method. In the two examples given in the previous section, if *n* is the number of elements in the array, then the execution time is $O(n)$ for serial execution, which is the same as for the normal loop implementations, the only difference being that the constant multiplicative factor may be somewhat greater for d-c than for simple loops.

For a parallel implementation, however, the gains should be striking. A shared-memory (PRAM) architecture is required. Whenever the problem is sub-divided in two by the d-c algorithm, the two sub-problems can then be solved in parallel on separate processors. Although they use their own versions of the working variables, all global variables (i.e. all variables in scope at the beginning of the **divcon** construct) are readable by all the sub-problems generated, so the obvious way to implement these variables is in shared memory. The potential parallelism is limited only by the size of the problem. In practice, the optimum amount of parallelism is likely to be considerably less than the maximum, but the important point is that it is easy to implement the program with any specified degree of parallelism (up to the maximum) without requiring help from the programmer.

For our two simple examples, if the maximum degree of parallelism is implemented (which requires $O(n)$ processors), the execution time is $O(\log n)$, a speedup of $O(n/(\log n))$. Of course, the speedup depends on the problem, but many common processing problems are subject to speedups of this order. If n is very large, the parallelism will be limited by the number of available processors (say p), but the speedup will typically be approximately p in these cases (i.e. close to the ideal speedup).

It should be completely feasible to implement a programming language containing the d-c construct described above (or an equivalent construct). The most difficult part is in prohibiting side effects within the d-c construct. Although it is possible to leave this entirely to the programmer's self-discipline, that would permit the development of unsafe and hence potentially unreliable software.

Another dilemma is whether or not to include conventional loop constructs (such as **for, while,** etc.) also. It is not necessary for these to be included; all computable functions can be programmed without them. Nevertheless, for a small minority of programming problems it may be inconvenient and cumbersome if they are not available at all and absolutely everything has to be programmed in terms of only conditional statements and d-c.

On the other hand, if conventional loop constructs are retained in the language, most programmers are so familiar with using loops that they will probably continue to use them rather than make the effort to learn the techniques for using d-c instead. If this happens, programs will continue to be written in the same old way and will completely fail to obtain the benefits of the parallelism available through d-c.

The ideal situation would be to have a language which provided d-c constructs in addition to the usual loops, but for the great majority of programs to be written using solely d-c and conditionals. These programs could then be implemented easily and efficiently either in the usual serial form on uniprocessors, or in parallel on shared memory multiprocessors.

3 Divide-and-Conquer in Functional Languages

3.1 Are Functional Languages Inherently More Parallel Than Procedural Languages?

Procedural language programs consist of a sequence of operations on the state of the machine. There are three main features of such languages that hinder their parallel implementation. The first is that any program is a sequence of operations and the programmer has to impose an order on the operations in his program, whether or not that order is logically necessary. It is very difficult for the compiler to deduce that some statements must be kept in the order written whereas other statements may be safely re-ordered.

The second feature that causes difficulties is the state itself. A construct such as the d-c construct introduced earlier allows the execution sequence to be split into two parallel sequences, but it does not split the state. In general, both execution sequences can access all variables in scope when the d-c construct is entered. Indeed, procedural languages routinely allow communication between different routines to occur via global or outer-block variables. Programmers are used to exploiting this facility and hence tend to think of shared variables as a natural and familiar means of communication. Most theoretical work on parallel algorithms has been in conventional procedural languages and it is no coincidence that far more work has been done on shared memory algorithms than on message passing algorithms. It is this fundamental rôle of state in procedural languages that makes shared memory parallelism seem easier and more natural.

Thirdly, the fact that most procedural languages permit side effects (e.g. global and outer-block variables may be updated by any procedure) makes it practically impossible for a compiler to change the order of execution of many parts of a program that would otherwise be perfectly safe to change. Hence automatic parallel implementation is very difficult to achieve.

Functional programming languages, on the other hand, have rather different characteristics. The programmer does not have to impose a sequential order on all operations whether or not it is required. In a functional program, there is no explicit order of operations. The compiler must deduce any necessary sequencing from the data dependencies (e.g. if x depends on y, then the computation of y must precede the computation of x).

Furthermore, there is no state, as such, in a functional language and hence there is less bias towards shared memory parallelism. Message passing approaches seem to fit comfortably into a functional language framework.[2]

Because of these potential advantages, and the more general benefits of functional programming (Hudak 1989; Hughes 1989), most of the recent work on the use of d-c as a general procedure or function which can be used as a building block in program construction has been done in a functional language context (Burton and Sleep 1981; Mou and Hudak 1988; Cole 1989; Kelly 1989; Mou 1990; Rabhi 1990). We follow the same approach here, and the remainder of this chapter discusses d-c as a basis for parallelism in functional languages. This is not to say that a similar approach cannot be used to achieve parallelism in procedural

[2] Most functional languages retain some bias towards shared memory, however, because outer block variables are usually accessible within function definitions and other program structures. As variables in functional programs cannot be changed after definition, access to outer block variables is equivalent to read-only shared memory.

languages. Any functional language program can be translated into a procedural language program, and usually the overall style and structure of the original program can be largely preserved (just as it is possible to translate a Fortran program into Pascal, and the result still has a recognisably Fortran-like style).

Functional programming languages do not include loop constructs such as **while...do...**, nor do they allow any explicit sequencing of operations, although there is implicit sequencing arising from the data dependencies. The programmer defines the control structure of his program by using **if...then...else...** expressions and recursive function definitions.[3] Whether or not parallelism can be easily found in such programs depends very much on the particular recursive definitions used. The d-c paradigm often permits a high degree of parallelism and can very often be used as a convenient alternative to the more familiar programming structures.

3.2 A Simple Introduction to Functional Languages

For the benefit of readers who are not familiar with modern lazy functional programming languages such as Miranda (Turner 1985) and Haskell (Hudak et al. 1990), we informally introduce a very simple functional language, sufficient for the purposes of illustration in the rest of this chapter. The language is approximately a common subset of Miranda and Haskell, but a few small differences have been introduced to achieve a high degree of simplicity, clarity and generality. The language is untyped and the pattern-matching features of Miranda and Haskell have been omitted.

3.2.1 Expressions

Expressions in the language can be constructed in the following ways:

1. As the **application** of a function to a single argument (in functional languages a function always has exactly one argument, hence the brackets around the argument can be omitted):

   $f\,x$

   denotes $f(x)$ in mathematics. Functions can give other functions as their results, and these may be applied to further arguments in turn (the default bracketing is from the left):

   $f\,x\,y\,z$ means $((f\,x)\,y)\,z$

   i.e. f is applied to its argument x, the result (another function) is applied to y and the result of that (yet another function) is applied to z. This is normally how we represent a function which would conventionally be defined to have three arguments.

   In general, all of f, x, y and z may themselves be expressions (bracketed if necessary).

[3] Strictly speaking, not even recursion is necessary. Any program can be written solely in terms of non-recursive function definition, provided higher order functions are allowed (the so-called Y combinator of combinatory logic can be used to simulate recursion). It is conventional to use recursion, however. In any event, the same arguments would apply to the use of the Y combinator or any other way of representing recursion.

2. As a **function definition** of the form:

   $x \rightarrow e$

   where x denotes any identifier and is the formal argument of the function, and e may be any expression (which usually involves the formal argument x). The whole expression denotes the function f where $f(x) = e$ in mathematics. It is a way of defining a function without having to give it a name. Default bracketing of these expressions is from the right, so:

   $x \rightarrow y \rightarrow x + y$ means $x \rightarrow (y \rightarrow x + y)$

   which defines a function which takes an argument x and gives as its result another function, which takes an argument y and gives as its result the sum of x and y.

3. As a **conventional expression** with infix operators, **if** expressions, brackets, etc. For example:

   if $x > 0$ **then** $x + 6$ **else** $2 * (x - 6)$ **fi**

   has its usual meaning. Prefix function application always binds more tightly than infix operators, so:

   $f\,x + g\,y$ means: $(f\,x) + (g\,y)$

   and both denote the mathematical expression $f(x) + g(x)$. The infix expression $x + y$ is interpreted as equivalent to $(+)\,x\,y$ where $(+)$ denotes the prefix form of $+$: $(+) = x \rightarrow y \rightarrow x + y$.

4. As a **tuple** containing two or more components which are separated by commas and enclosed in brackets, e.g.

   $(x+y, f\,a)$

   is the tuple whose first component is $(x+y)$ and whose second component is $(f\,a)$.

5. As a **where** clause which is of the form: any expression followed by the keyword **where**, followed by a list of definitions. These definitions are local to the expression. For example:

   $x + y$ **where** $x = 3; y = 7$ equals $3 + 7$

   $f = x \rightarrow g$ **where** $g = y \rightarrow a$ equals $f = x \rightarrow y \rightarrow a$

   Program layout (i.e. the indentation) is used to indicate the extent of the list of definitions in a **where** clause.

3.2.2 Definitions

Expressions are used to construct **definition statements** which have the form:

$x = e;$

where x denotes any identifier and e is any expression. Although definitions are superficially like assignment statements in procedural languages, the identifiers in a functional languages do not denote variables, and once an identifier has been defined it cannot be redefined to a new value (unless the identifier is being used in a different scope or context, in which case it

denotes a logically distinct object). For this reason, definitions may be written in any order and this order does not indicate the order in which they will be evaluated.

Tuples of identifiers are allowed on the left-hand side of definition statements:

(x,y,z) = e; is equivalent to *x = e 1; y = e 2; z = e 3;*

Tuples are treated as equivalent to partial functions over the integers, so that any tuple applied to the integer *i* gives the *i*-th component of that tuple, e.g. *(a,b,c)3* is equivalent to *c*.

3.2.3 Function Definitions

Functions can be defined using the definition statements already introduced. For example:

factorial = n → **if** *n=0* **then** *1* **else** *n* ∗ *factorial(n–1)* **fi**

An alternative syntactic form of function definition statement is permitted, which many programmers prefer:

f x = ... is synonymous with *f = x* → *...*

f x y = ... is synonymous with *f = x* → *y* → *...*

and so on. For example, the same factorial function definition may be written:

factorial n = **if** *n=0* **then** *1* **else** *n* ∗ *factorial(n–1)* **fi**

and function composition *(.)* may be defined in any of the following ways:

(.) = f → *g* → *x* → *f(g x)*

(.) f g x = f(g x)

f . g = x → *f(g x)*

(f . g) x = f(g x)

3.3 A Function for Divide-and-Conquer

A divide-and-conquer function is easy to define in functional programming languages. The first proposal to use such a function as a general basis for parallelism was made in (Burton and Sleep 1981), and others have developed and extended this approach. Following the approach we used in the procedural language context, a simplified form of the general d-c algorithm is used in which the sub-division is always into exactly two parts at a time. Of course, this entails no significant loss of generality as subdivision into any number of parts is possible, it simply requires more steps in which to do it.

Using our simple functional language, we define the d-c function *divcon* as follows:

```
divcon simple solve divide combine data =
if simple data then solve data else combine result1 result2 fi
        where
        result1 = divcon simple solve divide combine data1;
        result2 = divcon simple solve divide combine data2;
        (data1, data2) = divide data;
```

The function *divcon* takes five arguments: the first three are unary functions (*simple, solve* and *divide*); the fourth is a binary function (*combine*); and the last (*data*) is a data object of arbitrary complexity. This last argument is the data for the particular problem to be solved using the d-c method. The first four arguments effectively define the particular d-c algorithm to be used, and correspond to the four functions with the same names introduced in the procedural form of d-c in Section 1.2.

The parallel implementation of *divcon* in a functional language is not fundamentally different from its parallel implementation in a procedural language. Both rely on the property that the two sub-problems can be solved independently and in parallel (*result1* and *result2* in the above program).

4 Lists With Divide-and-Conquer

The basic data structure used in most functional languages is the list, following the lead set by Lisp in the 1960s (McCarthy 1978a, 1978b). The conventional definition of a list makes use of the primitive functions *head, tail, cons* and *null*[4] and is well-suited to a particular style of programming which is illustrated by the following programs taken from the Haskell Report (Hudak et al. 1991) (with syntax changes to remove the pattern matching) which define: *s++t*, the list formed by concatenating two lists *s* and *t*; and *s!!i*, the *i*-th element of the list *s*:

s ++ t = **if** *null s* **then** *t* **else** *head s : (tail s ++ t)* **fi**

s !! i = **if** *i = 0* **then** *head s* **else** *tail s !! (i–1)* **fi**

This style of programming in which operations on a list are defined recursively by operating on the head of the list and recursively applying the definition to the tail of the list does not contain any obvious parallelism. Furthermore, the primitive functions *head, tail, cons* and *null* are not well suited to use with the d-c paradigm. An alternative set of primitive functions for lists has been proposed (Axford and Joy 1991) which is well suited to use with d-c algorithms and easily parallelisable. These primitives are described in the next section.

4.1 A Divide-and-Conquer Model of Lists

The model contains six primitives (the first is a constant, the rest are functions): (i) *[]* is the empty list; (ii) *singleton x* (usually written as *[x]*) is the list containing the single element *x*; (iii) *s++t* is the list formed by concatenating lists *s* and *t*; (iv) *split s* is a pair of lists obtained by partitioning the list *s* into two non-empty parts (*s* must contain at least two elements); (v) *#s* is the number of elements in the list *s*; and (vi) *element s* is the only element in the list *s* (defined only if *s* contains exactly one element).

The algebraic specification of this model is:

#[] = *0*

#[x] = *1*

[4] *head s* is the first element of the list *s*; *tail s* is the list obtained by removing the first element of *s*; *cons x s* (more commonly written *x : s*) is the new list obtained by adding the item *x* to the beginning of *s* (so *x* is the head of the new list); *null s* is true if the list *s* is empty and false otherwise.

$\#(s \mathbin{++} t) = \#s + \#t$

$element\ [x] = x$

$s \mathbin{++} [\,] = s$

$s \mathbin{++} (t \mathbin{++} u) = (s \mathbin{++} t) \mathbin{++} u$

$split\ ([x] \mathbin{++} [y]) = ([x],[y])$

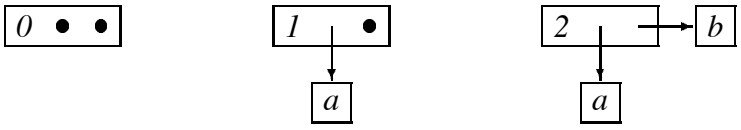
$\#u \geq 2$ **and** $split\ u = (s,t)$ **implies:**
$s \mathbin{++} t = u,\ \#s \geq 1,\ \#t \geq 1$

In this model of lists, the primitive operation for building up lists is concatenation (instead of *cons* in the traditional model), and the primitive function for breaking down lists is *split* (instead of *tail*).

4.2 Representation in the Computer

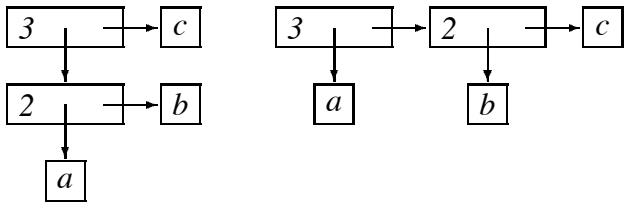
Suppose that a list is represented as a binary tree. Elements of the list are stored in the leaves of the tree, each leaf node containing exactly one element. No elements are stored in branch nodes, but each branch node contains the size of the list and two pointers: the left one points to the first part of the list, while the right points to the second part of the list. Ideally these two parts of the list should be approximately equal in length (i.e. the tree should be be balanced), but that is not a requirement for correctness of the representation, it affects only the performance.

The representations of the empty list (*[]*), a singleton list (*[a]*), and a list of two elements (*[a,b]*) are, respectively:



The symbol • denotes a nil pointer and occurs only in lists containing either no elements or just a single element.

Two alternative representations of the list *[a,b,c]* are:



For lists of three or more elements, the representation is not unique, but several different structures are possible. All are equally valid and will give exactly the same results, although the performance of a program may depend upon how well-balanced the representation is.

The above representation allows straightforward and obvious implementations of all five primitive functions. All execute in constant time, irrespective of the lengths of the lists involved. None of these primitives offers any scope for parallelism, however. That comes with their use in d-c programs and, in particular, their use in the implementation of higher-level list-processing functions.

This representation of lists is somewhat less efficient than the traditional representation in typical circumstances on sequential von Neumann architectures. The binary tree structure uses more pointers than a simple chain (up to twice as many) and this requires more memory space and increases the execution time for typical list-processing operations. This loss of efficiency (of up to a factor of two in both memory space and execution time) in the sequential implementation is insignificant in comparison to the gains that can be made with parallel implementation.

4.3 Some Examples of Use

4.3.1 Some Higher-Level List Functions

Programs involving lists rarely need to use the primitive functions directly, most things can be done much more easily using general high-level list-processing functions. One such function that is very widely applicable is *map*, defined as:

map f [x1,...,xn] = [f x1, ..., f xn]

This function may be programmed in the d-c style as:

```
map f s =
    if #s=1 then [f(element s)] else map f s1 ++ map f s2 fi
                where (s1,s2) = split s;
```

or, using the function *divcon* explicitly:

```
map f s = divcon single solve split (++) s where
                single s = (#s=1);
                solve s = [f(element s)]
```

Another very useful function is *reduce* which has the following definition, in which $\oplus$ denotes any infix operator:

reduce ($\oplus$) [x1,...,xn] = x1 $\oplus$ x2 $\oplus$ ... $\oplus$ xn

The operator $\oplus$ must be associative for the result to be uniquely defined. If the list is empty, the result is undefined. If the list contains only one element, the result is that element.

The function *reduce* may be programmed using d-c:

```
reduce f s =
    if #s=1 then element s else f(reduce f s1)(reduce f s2) fi
                where (s1,s2) = split s;
```

A third very useful high-level function is *filter*, which finds the sublist obtained by removing all elements which do not satisfy a given condition. It can be programmed using d-c (this time using *divcon* explicitly):

```
filter p s = divcon single solve split (++) s where
                single s = (#s=1);
                solve s = if p(element s) then s else [] fi
```

All three functions, *map, reduce* and *filter*, have sequential execution times of $O(n)$ provided that the list primitives execute in constant time. In a parallel implementation on a shared memory architecture, if the d-c parallelism is fully used, all three have execution times of $O(\log n)$, requiring $O(n)$ processors.

Many other common functions of lists have similar performance characteristics, with similarly large parallel speedup. In practice, if *n* is much larger than the number of available processors *p*, then *p* will limit the parallelism rather than *n*, and the speedup will be close to the ideal (i.e. close to *p*).

4.3.2 Other Applications

Instead of programming in terms of the primitive list-processing functions directly, it is much better to use high-level functions such as *map, reduce* and *filter*. All of these functions can be implemented using either the traditional *head, tail* and *cons* primitives or the new primitives, or any other suitable set of primitives for that matter. Furthermore, they may be implemented on many different computer architectures, using many different types of list representation. Programs that are written in terms of these high-level functions alone are thus immediately portable across all these different methods of implementation, both serial and parallel.

A good example of a problem that can be easily programmed purely in terms of high-level list functions is the ray tracing problem described next.

Example: Ray Tracing

Suppose we have a list of objects and a list of rays (representing physical objects and light rays in three dimensional space). The problem is to compute the first impact of each ray on an object.

Each ray is represented as a starting point and a direction. An impact with an object is simply represented as a distance, which is the distance the ray travels from its starting point before it hits the object. The first impact for a given ray is then the impact represented by the minimum distance. We do not go into the details of how objects are represented, or how the point of impact of a ray with an object is computed, but concentrate solely on the overall structure of the program.

One possible solution is as follows, expressed as a program for the function *impacts* which takes two arguments, a list of rays and a list of objects, and gives a list of impacts as its result (this list is in the same order as the input list of rays):

impacts rays objects = *map firstimpact rays* **where**
firstimpact ray = *reduce min (map impact objects)*
where
impact object = <distance travelled by *ray*
to hit *object*>;
min x y = **if** $x \leq y$ **then** *x* **else** *y* **fi**

The program uses *map* to apply the function *firstimpact* to each separate ray in the list; *firstimpact* being a function which takes a single ray as its argument and finds the first impact of that ray with any of the objects. That is done by first using *map* to apply the function *impact* to each separate object. The function *impact* takes a single object as its argument and

finds the impact of the current ray with that object. When the list of impacts of one ray with all the objects has been found, the function *reduce min* is used to find the first impact of that ray.

The outermost structure of this program is an application of *map* to the list of rays. This will take $O(\log n)$ time for a fully parallel implementation (on $O(n)$ processors), where *n* is the number of rays. The computation of the first impact of a single ray is an application of *map* to all the objects, followed by an application of *reduce* to all the objects, both of which take $O(\log m)$ time in a fully parallel implementation (on $O(n)$ processors), where *m* is the number of objects. Thus the execution time of the complete program is $O(\log m + \log n)$ for a fully parallel implementation (on $O(m \times n)$ processors).

5 Arrays with Divide-and-Conquer

The common use of arrays in programming languages typically provides very little to support parallelism. The primitive operations on arrays are accesses to single elements of the array, the element being identified by its index. For d-c style programming, operations are required which can partition arrays and put them back together again. Such operations should ideally be defined as primitive operations, so that the implementor is free to implement them in whatever manner is most efficient.

The programming language Divacon adopts just such an approach, using a functional language as the framework.

5.1 Divacon

Divacon (Mou, 1990) is a parallel functional programming language which has been designed particularly for architectures with parallel processor networks in the form of a hypercube or closely related topologies such as the butterfly, perfect shuffle or cube-connected-cycles. Divacon has been implemented in prototype form on the Connection Machine at Yale.

5.1.1 Overview of the Language

Divacon is a functional programming language which is conventional in its basic data types (integer, float, character and boolean) and operators ($+, -, *, \div, >, <$, etc.). The main data structures are tuples and arrays. Tuples are used statically (there are no operators to change the size of tuples), while arrays are dynamic and can be divided into smaller arrays or combined to form larger arrays (these are the operations needed for d-c programming). In fact, Divacon tuples are more like conventional arrays in many respects (except that their syntax is not at all the same), while Divacon arrays are really much more sophisticated data structures than normal arrays, having primitives which operate on the array as a whole, unlike the primitives for normal arrays which operate on single elements only.

In addition to the operations required by d-c algorithms for sub-dividing and re-combining arrays, some 'communication' operations are also provided on arrays. These permute the elements of an array according to a specified rule (defined in terms of the array indices only, not the values of the elements). For example, a simple rule would be to move the *i*-th element to the *(i+1)*-th position (modulo the size of the array).

An analog of the function *map* on lists (as defined in Section 4.3.1) is defined as a primitive function on both tuples and arrays. In Divacon this function is called 'distribution' and is denoted by the symbol *!* (used as a prefix operator).

The other main feature of Divacon is the divide-and-conquer function itself. Three forms are provided, all are variations on the general d-c paradigm.

It would occupy too much space to attempt a full description of Divacon here. The basic data types and operators, and many other basic features of the language are much like those in many other programming languages. The important new features of the language are its treatment of arrays (effectively a new type of data structure) and the d-c functions.

In the description which follows, many details of the syntax of Divacon are glossed over as they are not relevant to the present discussion. In some examples, the syntax has been changed to bring it more into line with our earlier examples and to avoid the need to describe the syntax of Divacon in detail.

5.1.2 Arrays in Divacon

An array is a function *A:I→V*, where *I* is a set of integer tuples called its index set, and *V* its indexed set. Each index (i.e. each element of *I*) has the form $(i_0, ..., i_{m-1})$ where $0 \leq i_j < S_j$ for all $0 \leq j \leq m-1$. This defines an *m*-dimensional array of size S_j in the *j*-th dimension. Primitive functions are provided in Divacon to give the size and shape of any array (hence requiring that this information be stored as part of the array).

Divide Functions Several primitive functions are provided to partition arrays into smaller arrays. The binary divide function d_b *m* divides a vector (i.e. a one-dimensional array) into two subvectors, the first consisting of the first *m* elements of the original, and the second consisting of the remainder. The left-right divide function d_{lr} is a special case of this for which *m = n/2*, where *n* is the number of elements in the vector. The even-odd divide function d_{eo} divides a vector into two subvectors, the first consisting of the even-numbered elements, while the second consists of the odd-numbered elements (the order of the elements otherwise remains unchanged).

For example,

d_b *4 [a b c d e] = ([a b c d], [e f])*
d_{lr} *[a b c d e f] = ([a b c], [d e f])*
d_{eo} *[a b c d e f] = ([a c e], [b d f])*

Similar operators are defined for multi-dimensional arrays also. These operators to sub-divide arrays into smaller arrays are fundamental to the use of d-c algorithms on data which is represented in the form of arrays. All of these division operators are 'polymorphic' in the sense that the way in which the array is sub-divided is determined purely by its size and shape, never on its contents.

Combine Functions Primitive combine functions are the inverse of the primitive divide functions. The combine functions c_{lr} and c_{eo} are defined to be the inverses of d_{lr} and d_{eo}, respectively.

So, for example,

c_{lr} *([a b c], [d e f]) = [a b c d e f]*
c_{eo} *([a b c], [d e f]) = [a d b e c f]*

Similar operators are defined for multi-dimensional arrays also. All the combine operators are polymorphic in the same sense as the divide operators.

Distribution Operator The distribution operator *!* is equivalent to the function *map* introduced in Section 4.3.1. So, for a function *f*, the new function *!f* applies *f* separately to each element of an array (or tuple).

For example,

!+ [(3,5) (1,4) (2,2)] = [8 5 4]
! [(3,5) (1,4) (2,2)] = [15 4 4]*

(Notice that Divacon regards +, *, etc. as prefix operators without enclosing them in brackets, and the argument of each is a single pair of numbers: they are not curried; e.g. *+(3,4)=7*.)

As with most of the Divacon primitives, the distribution operator can be used with multi-dimensional arrays also.

Communications Operator This operator (denoted #) permutes the elements of an array. It is applied to a function which specifies the particular permutation required. If *f* is a function, and $[a_1 ... a_n]$ is an array, then $\#f\,[a_1 ... a_n] = [(a_1, a_{f\,1}) ... (a_n, a_{f\,n})]$. This definition permits not only permutations of the array, but also the broadcasting of a single element into all positions in the array, etc.

For example, if *const m x = m*, and *mirr i = –i*, (interpreted modulo the size of the array),

#(const 0)[3 5 7 9 11] = [(3,3) (5,3) (7,3) (9,3) (11,3)]
#mirr [3 5 7 9 11] = [(3,11) (5,9) (7,7) (9,5) (11,3)]

and so *#(const m)* broadcasts element *m* to all positions in the array, while *#mirr* defines a mirror image reflection about the centre of the array.

5.1.3 Divide-and-Conquer Functions

Divacon defines three versions of the general d-c paradigm.

The function *Naive-PDC* can be defined as follows:

Naive-PDC (divide, combine, simple, solve) = f
where *f data =* **if** *simple data* **then** *solve data*
else *(combine . !f . divide) data* **fi**

(where . denotes function composition). The arguments to *Naive-PDC* are in a different order to those used in the d-c functions introduced earlier in this chapter, but are otherwise very similar except that the function *divide* partitions the problem not just into two sub-problems, but into an array (of arbitrary size) of sub-problems, and *combine* combines an array of answers to those sub-problems.

Another d-c function ('parallel d-c') is defined as follows:

PDC (divide, combine, pre, post, simple, solve) = f
where *f data* = **if** *simple data* **then** *solve data*
else *(combine . post . !f . pre) data* **fi**

The functions *pre* and *post* have been included to provide pre-adjustment and post-adjustment of the data within the d-c construct.

A third form of the general d-c paradigm is provided. It imposes a linear order on the recursive operations and hence is called sequential divide-and-conquer:

SDC (divide, combine, $(\mu_0, ..., \mu_{k-2})$*, simple, solve) = f*
where *f data* = **if** *simple data* **then** *solve data*
else *(combine . h . divide) data* **fi**
where $h\ (x_0,...,x_{k-1}) = (y_0,...,y_{k-1})$
where
$y_0 = f\ x_0$;
$y_1 = (f\ .\ \mu_0)\ (x_1,y_0)$;
. . .
$y_{k-1} = (f\ .\ \mu_{k-2})\ (x_{k-1},y_{k-2})$

The function *divide* partitions the problem into k parts, and the functions $\mu_0, ..., \mu_{k-2}$ define the way in which the solutions to these k parts interact.

This sequential d-c function is more general than the d-c algorithms previously discussed, but it has the disadvantage of requiring the sub-problems to be solved in a specified order (the solution to each must be available before the next in the sequence can be solved), hence no structural parallelism is possible. The only possible parallelism is within the separate sub-problems (or by transforming the program into one of the more restricted forms of d-c).

5.1.4 Implementation of Divacon

A prototype implementation of Divacon has been constructed in *Lisp on the Connection Machine at Yale University. The implementation uses both data parallelism and control parallelism. Arrays are distributed over the available processors and hence represented in a highly parallel form. Operations on arrays, such as their creation, the divide, combine and index translation operations can then be performed in parallel. All these operations take constant time provided the size of the array does not exceed the number of processors.

The parallel d-c functions are also implemented in parallel, and provide the only source of control parallelism.

It is non-trivial to implement Divacon in *Lisp, which is essentially a data-parallel language and provides only flat parallel data structures. It is not entirely clear how much the implementation has constrained the performance and design of Divacon, and whether or not its limitations are due to the implementation language or to the underlying machine architecture. Later implementations have been completed for the MasPar MP-1 (2D mesh architecture) and WaveTracer DTC (3D mesh), both massively parallel machines using bit-processors.

5.1.5 Programming in Divacon

The d-c model provided by Divacon is both powerful and convenient. A very wide range of applications can be readily programmed with the constructs provided giving programs which

execute in parallel with high performance. Some examples of such problems, which have all been programmed in Divacon, are polynomial evaluation, matrix multiplication, monotonic sort, FFT and banded linear systems.

The divide and combine functions for arrays in Divacon are all polymorphic in the sense defined earlier, i.e. the subdivision of an array is independent of the values of the elements of the array. This means that only 'polymorphic' d-c algorithms can be programmed easily. This rules out an important class of d-c algorithms which use non-polymorphic divide and combine operations. A simple example is the divide operation in quicksort, in which the array is split into two parts, one part being all elements less than a guessed median value, and the other part being the remainder. Clearly, this operation depends not on the indices of the elements, but on their values.

6 Sets and Mappings With Divide-and-Conquer

The basic concepts of set theory mathematics are widely used in computer science, yet relatively few programming languages have incorporated sets as basic data structures. The best known procedural language based on sets is SETL, which has been in use for some two decades and has had some success as a very high-level language for general software development and prototyping (Schwartz et al. 1986). It has remained very much a minor language, however, and does not seriously compete with the major programming languages.

The use of set theory in program specification is much more universal. The major formal specification methods, such as Z (Spivey 1989) and VDM (Jones 1986), are based on set theory mathematics, and use sets and relations as basic data structures out of which everything else is constructed. Experience with these methods has shown that set theory provides a powerful and convenient way in which the programmer can express the organisation of his data. Recently there has been increasing interest in representing sets and relations directly in programming languages so that they can be used as tools for further software development (North 1990; Treadway 1990; Marino and Succi 1991).

The common programming languages have not found it appropriate to include sets and relations as basic data structures because they are rather more difficult and somewhat less efficient to implement than arrays or lists, and their conceptual advantages were felt to be less important than maximising performance. Furthermore, why change something that has worked well for years?

The balance of advantage *is* changing, however. Not only are sets and relations mathematically more powerful and elegant that arrays and lists, but they probably offer easier and more efficient parallel implementations. In the rest of this section, we introduce the idea of using mappings as basic data structures and some primitive functions for operating on such structures using the d-c style of programming (Axford 1991).

For simplicity, we choose to define a single type of data structure only (although it is quite possible to provide other types of structures as well, or instead). The chosen structure is called a 'mapping' and corresponds to a finite partial function in set theory mathematics. It is a convenient choice because it is reasonably easy to implement efficiently on most computer architectures, whether serial or parallel, and it is convenient to use in programming. Sets need not be provided separately as they are very easily represented in terms of mappings.

Sequences and arrays are simply special cases of mappings. Relations are more general, but they are also more difficult to implement efficiently, so they appear to be a less attractive choice as the basic type of structure. This is an area in which there is rather limited experience, however, particularly of parallel implementations for general-purpose use, and a great deal more research is needed to find optimum solutions.

6.1 Primitive Functions of Mappings

Any abstract data structure can be defined by giving an algebraic specification of the properties of the primitive functions which operate on it. In choosing a suitable set of primitive functions, mathematicians will usually give priority to simplicity and elegance in the specification itself. Here, we are more concerned with choosing a set of primitive functions that are convenient for the programmer and can be efficiently implemented on both serial and parallel architectures. The set of primitives is not intended to be the simplest or most elegant from a mathematical viewpoint. Furthermore, a complete formal specification is rather long and tedious, and not necessary for our purposes here, so the primitives will be defined more informally for brevity and ease of understanding.

A mapping may be thought of as a data structure which represents a function of a single argument over a finite domain. The representation stores the function values for all possible argument values. We will refer to the argument value as the *key* and to the corresponding function value as its *attribute*. The set of keys is the domain of the function, each key being 'mapped' to its attribute.

The following functions are considered as primitives:

CreateMap	*MapUnion*	*ForAll*
IntegerSequence	*MapSize*	*MapMap*
ApplyMap	*SubMap*	*ReduceMap*
MapIntersection	*IsIn*	*ReduceList*
MapDifference	*ForOne*	

Each is now defined in turn.

6.1.1 CreateMap

This takes a single argument which is a tuple[5] of pairs. The pairs are the key-attribute pairs which define the mapping. If any duplicate keys are present in the tuple, then the result is undefined. The order in which the pairs occur is irrelevant. For example:

CreateMap ((3,6),(1,4),(0,1))
CreateMap ((1,4),(0,1),(3,6))

each represent the mapping $\{0 \mapsto 1, 1 \mapsto 4, 3 \mapsto 6\}$. We will usually write *{(x1,y1),...,(xn,yn)}* as a shorthand for *CreateMap ((x1,y1),...,(xn,yn))*.

[5] We assume that tuple structures already exist in whatever programming language we are using. If not, it is quite straightforward, if more tedious and less efficient, to build up mappings of any size by adding elements one at a time.

6.1.2 IntegerSequence

This function takes two arguments, both integers, and creates the mapping whose domain is the sequence of integers from the first argument to the second. The attribute values created are the same as the keys. The infix operator form *m..n* will normally be used in preference to the prefix function *(IntegerSequence m n)*. So, for example:

3..7 has the value {*(3,3),(4,4),(5,5),(6,6),(7,7)*}

6.1.3 ApplyMap

This operation applies the mapping as if it were a function, i.e. it maps a given key to its attribute. It takes two arguments, the first is a mapping and the second is the key value to be mapped. Thus *(ApplyMap s k)* is the attribute associated with the key *k* in the mapping *s*. For example:

ApplyMap {*(3,6),(1,4),(0,1)*} *1* has the value *4*.

6.1.4 MapIntersection

This function takes the set intersection of the domains of two mappings as the domain of the resultant mapping. The attribute of a given key in the result is a function of the attributes of that key in the two argument mappings. The first argument is this function, the second and third arguments are the two mappings. So, *(MapIntersection f s t)* is the mapping in which every key, *x*, that is present in this mapping is also a key in *s* and *t*; and if *(x,y)* is in *s* and *(x,z)* is in *t*, then *(x, f y z)* is in *(MapIntersection f s t)*. For example:

MapIntersection (*) {*(1,7),(2,45),(3,2)*} {*(0,3),(1,2),(3,3)*}

has the value {*(1,14),(3,6)*}.

6.1.5 MapDifference

This function takes two mappings as its arguments. The difference of the domains of the two mappings is the domain of the result. The attributes remain unchanged. In other words, *(MapDifference s t)* is the mapping which contains all the elements of *s* for which the keys differ from all keys in *t*. For example:

MapDifference {*(1,7),(2,45),(3,2),(4,7)*} {*(0,3),(1,2),(3,3),(4,4)*}

has the value {*(2,45)*}.

6.1.6 MapUnion

This function takes the set union of the domains of two mappings as the domain of the resultant mapping. The attribute of a given key in the result is the attribute of that key in the argument in which is occurs (if it occurs in only one), otherwise it is a function of the two attributes if that key occurs in both. MapUnion is applied to three arguments: the first is a binary function, the second and third are mappings. For example:

MapUnion (*) {*(1,7),(2,45),(3,2),(4,7)*} {*(0,3),(1,2),(3,3),(4,4)*}

has the value {*(0,3),(1,14),(2,45),(3,6),(4,28)*}.

6.1.7 MapSize

The size of a mapping is the number of elements it contains. For example:

MapSize {*(1,7),(2,45),(3,24),(7,17)*} has the value *4*

6.1.8 SubMap

This function gives the sub-mapping that consists of all elements of a given mapping whose keys satisfy a given condition. It takes two arguments, the first is a unary function (the condition), while the second is the mapping. For example:

SubMap (x→x>2) {*(1,7),(2,45),(3,24),(4,17)*}

has the value {*(3,24),(4,17)*}, i.e. all elements with keys greater than 2.

6.1.9 IsIn

This function tests if a given value is present in the domain of a given mapping. It takes two arguments, the first is the key value to the tested for, the second is the mapping in which the key is to be looked for. The result is a boolean. For example:

IsIn 3 {*(1,7),(2,45),(3,24),(4,17)*}

has the value *true* because the mapping contains the key *3*.

6.1.10 ForOne

This function tests whether or not a given condition is true for at least one member of the domain of the mapping. It takes two arguments: the first is a monadic function whose result is a boolean, the second is a mapping; the result is a boolean.

ForOne (x→x>40) {*(1,7),(2,45),(3,24),(4,17)*}

has the value *false* because there is no key greater than *40*.

6.1.11 ForAll

This is the similar function which tests whether or not the condition is true for all values in the domain of the mapping. For example:

ForAll (x→x>0) {*(1,7),(2,45),(3,24),(4,17)*}

has the value *true* because every key in the mapping is greater than zero.

6.1.12 MapMap

This function performs the same operation on every element in a mapping. So, *(MapMap f s)* is the mapping in which every element of the form *(x,y)* in *s* becomes *(x, f x y)* in the result. The first argument is a binary function, the second is a mapping and the result is a mapping. For example:

*MapMap (x→y→2*y)* {*(1,7),(2,45),(3,24),(4,17)*}

has the value {*(1,14),(2,90),(3,48),(4,34)*}, i.e. every attribute is doubled.

6.1.13 ReduceMap

This function reduces a mapping to a single value by applying a binary function repeatedly to all the elements. It takes four arguments: the first and second are binary functions, the third is any type of object, and the fourth is a mapping. *(ReduceMap f g z s)* is defined to have the value *z* if *s* is empty; it has the value *(g x y)* if *s* contains only one element $s=\{(x,y)\}$; and in all other cases (i.e. two or more elements in *s*) it is defined recursively to be

f (ReduceMap f g z s1) (ReduceMap f g z s2)

where *s1* and *s2* are any two parts of the mapping *s*, such that both of the following equations are true (where *f* is any function):

MapIntersection f s1 s2 = $\{\}$
MapUnion f s1 s2 = *s*

The first argument, *f*, is not significant in either case.

Proof Obligations: *ReduceMap* is a powerful and useful function, but it has the rather dangerous property of being potentially non-deterministic, i.e. its definition is sometimes ambiguous, and when evaluated in different ways it can lead to different results. To avoid non-determinism (i.e. to ensure that its value is always uniquely defined), the function *(ReduceMap f g z s)* should only be used when *f* is commutative and associative, and when *z* is an identity element of *f*, i.e. when all three of the following equalities hold:

f a b = *f b a*
f (f a b) c = *f a (f b c)*
f z a = *a*

for all *a, b* and *c*. These must be regarded as proof obligations on the programmer whenever he or she uses *ReduceMap*. If these properties are not satisfied by the arguments of *ReduceMap*, then the results may be non-deterministic. Non-deterministic functions are dangerous because they are not referentially transparent and hence program analysis is made very much more difficult and counter-intuitive. For example, we can no longer make the apparently obvious statement that:

ReduceMap f g z s = *ReduceMap f g z s*

simply because *ReduceMap* may give different values on different occasions even though all its arguments remain the same.

Another way to avoid this problem is to tighten up the specification by defining precisely the order of evaluation. In the functional language Haskell, the function *foldr* is similar to *ReduceMap* except that the order of evaluation is precisely defined. This is certainly a simple solution which avoids the problem of non-determinism, but it also makes general parallel implementation very much more difficult and less efficient, so we rule it out on those grounds. The definition of *ReduceMap* given above allows the implementor a great deal of freedom to choose the most efficient order of evaluation, at the expense of putting an obligation on the programmer to make sure that he or she uses *ReduceMap* only when its arguments have the required properties. If the programmer chooses to ignore this obligation, then he has only

himself to blame if either (i) some implementations of his program give incorrect results, or (ii) mathematical analysis of the program is misleading.

With the increasing trend towards more formal analysis and verification of programs, and with the increasing availability of software tools to aid the programmer at this task, imposing such proof obligations on the programmer is likely to become much more acceptable in the future. The great majority of programmers today would probably regard any such proof obligations as thoroughly undesirable, but these attitudes are slowly changing. Already, it is widely recognised that formal methods of program analysis and proof are becoming essential in some specialised fields of programming such as concurrency and safety-critical systems. The wider use of formal methods will inevitably follow.

6.1.14 ReduceList

Suppose it is required to reduce a mapping with a function that is non-commutative. In this case it is necessary to define the order in which the elements are enumerated, although the order in which the reduction operations are carried out may still be left undefined provided the function is associative. Suppose that the data is in the form of a list and the order of elements in the list is specified to be the order of enumeration for the reduction.

In this context, a list *[x1,...,xn]* is taken to be simply a shorthand for the mapping $\{(1,x1),...,(n,xn)\}$ with the domain $1,...,n$. The function *(ReduceList f g z s)* is defined to be equal to *z* if *s* is empty; to *(g x)* if *s* is the list *[x]* (i.e. the list containing the single element *x*); and in all other cases it is defined recursively as:

f (ReduceList f g z s1) (ReduceList f g z s2)

where *s1* and *s2* are any two non-empty lists such that *s* is equal to the concatenation of *s1* and *s2* (in that order).

Proof Obligations: The requirements on the arguments of *ReduceList* for the result to be deterministic are less stringent than before. They are simply that *f* be associative:

$f\ (f\ a\ b)\ c\ =\ f\ a\ (f\ b\ c)$

6.2 Some Applications

6.2.1 Ray Tracing

Consider the ray tracing problem introduced earlier in Section 4.3.2. The data consists of a set of objects and a set of rays (e.g. representing light rays) and we wish to find the first impact of each ray on an object. Suppose the data is supplied in the form of two mappings. In both cases, the domain of the mapping (i.e. the set of keys) is unimportant and may simply be the integers from *1* to *n*. In the first mapping (called *rays*) the attributes are the rays, while in the second mapping (called *objects*) the attributes are the objects. The required result is another mapping over the same domain as *rays*, but with the attributes being the first impacts of the corresponding rays. An impact is represented by the distance along the ray to that impact (a ray is assumed to have a starting point, and the distance may be infinite if there is no impact). We do not concern ourselves with the details of the representation of the individual rays and objects.

Using a similar d-c approach to that adopted in Section 4.3.2, we first sub-divide the problem by sub-dividing the set of rays, and then in the computation of the first impact of a given ray, we sub-divide further by sub-dividing the set of objects. A function to do this can be programmed as follows:

```
impacts rays objects = MapMap firstimpact rays where
    firstimpact key ray = mindistance where
        mindistance = ReduceMap min impact infinity objects;
        impact key object = <distance travelled
                                                    by ray to hit object>;
        min x y = if x≤y then x else y fi;
        infinity = <value to represent no impact>;
```

This program uses *MapMap* to apply the function *firstimpact* to every ray (a parallel implementation could do these computations concurrently). The definition of *firstimpact* then uses *ReduceMap* to apply the function *impact* to every object to find the impact of the current ray with one object, and to reduce these with the function *min* to obtain the first impact (the first impact being the impact at the minimum distance). Note that the definition of *impact* defines *object* to be an argument (along with *key*, which is not used), but *ray* is a free variable (i.e. it is accessed directly as an outer-block variable rather than passed to the function as an argument).

An alternative style of program divides the objects first:

```
impacts rays objects =
    ReduceMap (MapUnion min) getimpacts {} objects
        where
        min x y = if x≤y then x else y fi;
        getimpacts key object = MapMap findimpact rays
            where
            findimpact key ray = <distance travelled
                                                    by ray to hit object>;
```

The structure of this program is rather different. Firstly, *ReduceMap* is used to apply the function *getimpacts* to each object separately (a parallel implementation could do these concurrently). The function *getimpacts* computes the list of impacts of all the rays with a single object (the argument). This is done by using *MapMap* to apply *findimpact* to each ray separately, where *findimpact* is very similar to *impact* in the previous program, but its second argument this time is *ray*, and *object* is a free variable. It computes the impact of a single ray on a single object. Finally, the function *ReduceMap* (in the first line of the program) combines all the sets of impacts, using *(MapUnion min)*, which picks out the first impact for each ray.

Both solutions offer a great deal of parallelism if the primitive functions *MapMap* and *ReduceMap* can be implemented in parallel. Such parallel implementation is discussed further in section 6.3.

In both programs, the use of *ReduceMap* gives rise to proof obligations. It is not difficult to show that the arguments of *ReduceMap* have the required properties so the results are uniquely determined.

6.2.2 Hoare's Quicksort Algorithm

Hoare's 'quicksort' algorithm can be programmed using the general d-c function as shown below. For simplicity we assume that the file contains at least one element and that all elements are different. The input data is assumed to be in the form of a list represented as a mapping with the domain *1,...,n*.

The program below uses the function *divcon*. The final argument of *divcon* is a triple *(m,n,s)* and the program sorts the segment of the sequence *s* between index values *m* and *n*.

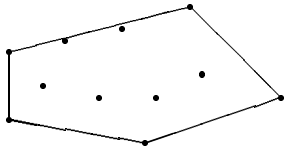
```
sort s = divcon simple solve divide (MapUnion f)
                                        (1, MapSize s, s) where
    simple (m,n,s) = (m=n);
    solve (m,n,s) = s;
    divide (m,n,s) = ((m,p,s1),(p+1,n,s2)) where
        median = if first<second then first else second fi;
        first = ApplyMap s m;
        second = ApplyMap s (m+1);
        s1 = belowmedian m n;
        s2 = abovemedian m n;
        belowmedian m n =
            if m>n then {}
            elsf y≤median then MapUnion f {(m,y)} rest
            else rest fi where
                rest = belowmedian (m+1) n;
                y = ApplyMap s m;
        abovemedian m n =
            if m>n then {}
            elsf y>x then MapUnion f {(n,y)} rest
            else rest fi where
                rest = abovemedian m (n–1);
                y = ApplyMap s n;
        p = m + MapSize s1 – 1;
```

In this program, the expressions *(belowmedian m n)* and *(abovemedian m n)* give sequences of all the elements of the segment of *s* from *m* to *n* which are below or above the median, respectively. The former sequence begins at index *m*, while the latter sequence ends at index *n*. The estimation of a median value is very crude, being simply the smaller of the first two items in the file. (It is easy to improve this.) The combine phase of the d-c algorithm uses *(MapUnion f)* to rebuild the file (*f* is irrelevant).

For example, if the initial data is {*(1,4),(2,3),(3,7),(4,2),(5,6)*} (representing the list *[4,3,7,2,6]*) then the value used for *median* will be *3*, and the subfiles *s1* and *s2* will be computed to be {*(1,3),(2,2)*} and {*(3,4),(4,7),(5,6)*}, respectively (representing the lists *[3,2]* and *[4,7,6]*). Notice that the definitions of *belowmedian* and *abovemedian* are recursive because the algorithm used to compute them is inherently sequential (there is no obvious way in which the computation of these functions may benefit from parallel processing).

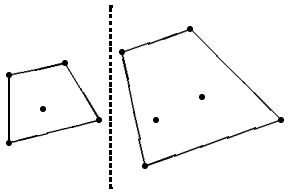
6.2.3 Convex Hull

The convex hull of a set of points in a plane is the smallest enclosing convex polygon. For example, the set of eleven points shown in the diagram has the convex hull indicated:



A program for finding the convex hull of a set of points can be written using a divide-and-conquer algorithm, using the fact that it is relatively easy to combine two non-overlapping convex polygons into a single convex polygon which encloses the original two (Preparata & Hong, 1977). (This involves much less work than combining two overlapping convex polygons.) We can take advantage of this if we divide the original set of points into non-overlapping subsets. An easy way to do this is to order the points in order of their X-coordinates (and if two points have equal X-coordinates, they are ordered on their Y-coordinates). If this ordered list of points is now divided into sublists, each sublist will have a convex hull which does not overlap the convex hull of any other sublist.

For the example given above, the points can be divided into two sets with convex hulls as shown:



A suitable program to do this is shown below. The data is assumed to be in the form of a list of points (represented as a mapping called *points*) which have already been ordered in the required manner.

```
convexhull points = ReduceList f g z points where
    f hull1 hull2 = <compute the convex hull
                                    which encloses both hull1 and hull2>;
    g point = <the polygon consisting of that one point>;
    z = <the empty polygon>;
```

We can easily represent a polygon as an ordered list of points (its vertices) and it is easy to write programs for the functions *g* and *z*. The function *f* which combines two convex hulls is harder, but the details do not really matter here, as we have sufficient of the structure of the program to show the potential for parallel implementation.

A more efficient program is likely to result if we combine the sorting and the convex hull computation into a single application of d-c. This is easy to do, using the quicksort algorithm for the sorting:

```
convexhull points = polygon where
    polygon = divcon lengthOne I divide combine points;
    lengthOne s = (MapSize s = 1);
```

```
I x = x;
divide s = (s1,s2) where
    s1 = SubMap p1 s;
    p1 x = (x≤median);
    s2 = SubMap p2 s;
    p2 x = (x>median);
    median = <estimate the median value>;
combine p1 p2 = <convex hull enclosing p1 and p2>;
```

In this single application of *divcon*, the function *divide* effectively does the sorting, while *combine* does the convex hull computation.

The fact that this quite complex problem can be solved by a program which is in the form of a single application of *divcon* (with all the detailed computation captured in the arguments) illustrates well the power of the abstract d-c function. A good compiler could reasonably be expected to obtain quite large-grain parallelism from such a program. In the next section, the problems of implementation are discussed further.

6.3 Parallel Implementation of Mappings

The primitive functions of mappings can all be implemented in the divide-and-conquer style if mappings are represented as trees, hash tables, or other representation that allows partitioning into two approximately equal parts, preferably in constant time (i.e. independently of the number of elements in the mapping). This allows efficient parallel implementation on multiprocessor shared memory (i.e. P-RAM) architectures, just as can be done for lists.

Mappings are also well suited to parallel implementation on distributed architectures (without shared memory). In this case, the representation of a mapping may be distributed across the local memories of the processors, a different part of the mapping being stored on each processor. This approach is essentially like a very high-level SIMD architecture. Real SIMD machines available at present are generally SIMD at a low level, with each machine instruction being executed simultaneously by all processors on their own local data. In a distributed mapping implementation, the synchronisation is at a much higher level, with each high-level primitive function being executed simultaneously by all processors. For example, consider the following situation.

Suppose the mapping is distributed across all available processors by using a hashing function of the keys. For P processors, an element with key k is assigned to processor i, where i=*hash k* and the function *hash* gives values in the range $1,...,P$ for all arguments. The primitive functions of mappings can be implemented as follows.

6.3.1 ApplyMap

For the function application *ApplyMap s k*, the only part of the mapping that is required is that containing the key k, which resides on processor number *hash k*. All other processors can remain idle.

The execution time for such a parallel distributed implementation is the sum of the following: (i) the time to compute *hash k* on the master processor; (ii) the time to communicate with processor number *hash k*; (iii) the time to evaluate *ApplyMap s k* on the part of the

mapping held on processor number *hash k*; and (iv) the time to return the result to the master processor.

6.3.2 MapIntersection

The function application *MapIntersection f s t* can be implemented by concurrently evaluating *MapIntersection f si ti* on each processor, where *si* and *ti* are those parts of the mappings *s* and *t*, respectively, which reside on processor number *i*.

The execution time is the sum of: (i) the time to broadcast the instruction to all processors; and (ii) the maximum time to compute *MapIntersection f si ti* (the maximum being taken across all values of the processor number, *i*). Assuming that the sequential processing of *MapIntersection* takes $O(n)$ time (where *n* is the total number of elements in the mappings *s* and *t*), and that the broadcast communication time is constant, then the overall parallel execution time is $O(n/P)$, or constant if *P* (the number of processors) is $O(n)$, provided that the hashing function distributes the mapping evenly across the processors. There is insufficient practical experience to be able to say at present just how well this method will work in typical computing applications. Some form of dynamic load balancing may well be needed in addition to the distribution of load provided by the hashing function.

The functions *MapDifference* and *MapUnion* can be implemented similarly, with similar execution time behaviour.

6.3.3 MapSize

The function application *MapSize s* can be implemented by evaluating *MapSize si* concurrently on each processor and then summing all the individual results. The time taken is the sum of: (i) the time to broadcast the instruction to all processors; (ii) the maximum time to compute *MapSize si*; and (iii) the time to sum the results. We assume that the broadcast time is constant, and that the time to compute *MapSize* sequentially is at most $O(n)$, although it may be constant in many implementations. Either way, for an even distribution of the mapping and $O(n)$ processors, the parallel time for (ii) is approximately constant. The time for the final step (summing the results obtained on the individual processors) is $O(\log n)$, which dominates the timing, hence the total parallel execution time of *MapSize* is also $O(\log n)$.

6.3.4 Discussion

The other primitive functions can be implemented in similar ways using d-c algorithms to achieve high parallelism. No programming language has yet been constructed using this approach, so the advantages of mappings as basic data structures for parallel programming are essentially speculative at the present time. Many practical problems remain to be overcome, but the approach looks extremely promising nevertheless.

7 Summary and Conclusions

The development of parallel software is floundering (Pancake, 1991) because there is no universal, architecture-independent model of programming. The purpose of this chapter has been to suggest that there is a promising case for considering the divide-and-conquer

paradigm as a fundamental design principle with which to guide the design of both control structures and data structures for new models of programming. The important characteristic of d-c is that it contains inherent parallelism in a very abstract and general form, and hence is not dependent upon a particular architecture.

Procedural Languages and Side Effects

A d-c control construct for procedural languages can be defined and, although unfamiliar, is not difficult to use. This construct can replace the usual loop constructs (such as **for** and **while** statements) and permits obvious control parallelism. The main problem with such a construct is that side effects must be prohibited for it to be well defined. Very few procedural languages prohibit side effects and hence neither users nor compiler writers are attuned to the side-effect-free style of programming, although it has for long been advocated as good practice in most circumstances. Of course, as freedom from side effects is required only within the d-c construct itself, the rest of the program need not be side-effects free. It would be possible, although unsafe, to put the d-c construct into any normal procedural language and rely solely on the self-discipline of the programmer to avoid side effects.

The toleration of side effects by most procedural languages is itself a serious problem for automatic parallel implementation (as it is an equally serious problem for other automatic optimisations, for the same reasons). The admission of side effects means effectively that an implementation model must be defined as part of the language definition (because without it the side effects are meaningless). This requires all implementations of the language to be consistent with this model, thus severely constraining the implementor. A good principle for any architecture-independent language to follow is to allow as much freedom as possible on methods of implementation.

Functional Languages

Pure functional languages are completely free of side effects, so they avoid one of the main weaknesses of the common procedural languages (although it is true that procedural languages without side effects do exist, but they are not in common use). Functional languages also make it easy to use functions which operate on other functions and generate yet further functions as their results. They provide a very convenient medium in which to define and use d-c functions. A further major advantage for parallel programming is that they do not require the programmer to put all program instructions into sequential order, whether or not the ordering of instructions is logically necessary.

In addition, functional languages provide a very much better medium for program analysis; for example, that required when verifying the proof obligations imposed by the use of non-deterministic functions such as *reduce* and *ReduceMap*.

Nevertheless, it is not essential to use a functional language. Most of the discussion and examples given in this chapter have been in the context of a functional programming language, but there is nothing that could not be done in a similar way in a procedural language, albeit with more difficulty. A functional language gives a number of benefits, not least being greater simplicity and elegance.

Control Parallelism v. Data Parallelism

Many authors have defined d-c procedures or functions, and their inherent control parallelism is well known: whenever the problem is divided into independent sub-problems, the solutions to those sub-problems may be obtained concurrently. The overall efficiency of a d-c algorithm is very dependent upon the efficiency with which the problem can be divided into parts, and the efficiency with which the solutions to the parts can be combined to give the overall solution. Often, large data structures are required to represent the problem data and/or the problem solution. Efficient ways of dividing and combining these data structures are then needed—rather than the efficient single-element operations required for the traditional iterative loop style of programming which we are all so used to.

Hence, the effective use of the control parallelism available in d-c programs often depends very much on having suitable data structures available, ones for which very fast divide and combine operations can be implemented. In this chapter, three classes of data structure have been considered from this point of view: lists, arrays and mappings.

Lists

By choosing a different set of primitive functions to the usual *head, tail* and *cons*, we can define lists in such a way that divide and combine operations are primitives, and everything else is programmed in terms of these. This opens the way to very efficient parallel implementation of many d-c programs using lists on shared memory (PRAM) architectures. As yet, no general programming language has been constructed using this approach, so experience of this style of parallel programming is extremely limited. The method looks very promising, however, particularly for those types of programming languages which have traditionally used lists as their primary data structures, but further investigation is required.

The efficient implementation of lists for d-c programs on distributed (non-shared-memory) architectures is an even more open question, although there appears to be no obvious reason why it should not be possible to find satisfactory representations on these architectures also. More research is needed to determine if this is a useful approach. Of course, lists can easily be represented as 1-D arrays and we can look for suitable ways of defining and representing arrays instead.

Arrays

Arrays can be re-defined to provide divide and combine operations as primitives instead of (or as well as) the usual single-element access operations of conventional arrays. George Mou's work on the language Divacon has shown how this can be done in a way suited to data parallel implementation of the arrays on hypercube and related architectures. Divacon is the first (and so far, only) language whose primary aim is to provide a medium in which programs with a high degree of d-c parallelism can be easily and conveniently written. While the language is quite general-purpose, it is targeted at certain hypercube architectures (in particular, the Connection Machine); and the limitations of the target architecture have somewhat constrained the language design. Nevertheless, Divacon supports efficient programs for the simpler types of d-c algorithms and has demonstrated that these can be programmed simply and conveniently, and then processed fully automatically to achieve very high degrees of parallelism.

Arrays in Divacon are defined with divide and combine primitives, as well as operations to permute the elements in specified ways (dependent only on the array indices, not on the values of the elements) or copy some elements to other positions in the array (again specified only by position, not by value). Such permute and copy operations are needed for the data-parallel representation used in the Connection Machine implementation of Divacon, but would not be necessary if a shared memory implementation was being used instead. On the other hand, Divacon provides no simple and efficient way of dividing (or combining) arrays by content, as needed in programming the quicksort algorithm, for example. This limitation is again due to the need to be able to implement the language easily and efficiently on the Connection Machine.

Sets and Mappings

Probably the most attractive data structures for an architecture-independent programming languages are sets and mappings. They are powerful and convenient for the programmer, as evidenced by their very widespread use in specification languages, for which convenience of use is much more important than convenience of implementation. Of course, data structures based on sets and mappings are more complicated and less efficient to implement sequentially than conventional arrays or lists. The situation changes markedly for parallel implementation, however, for which absence of a specified order to the elements of sets and mappings gives a useful additional degree of freedom to the implementation.

Many programs in conventional languages use lists or arrays to represent data for which the order is unspecified and irrelevant to the computation. Imposing an unnecessary order on the elements of such data structures imposes unnecessary constraints on the optimisations which the language processor can carry out (unless it has a very deep knowledge of the mathematical properties of the program, which is typically far beyond the abilities of any normal compiler or interpreter). The use of sets and mappings avoids the need to impose arbitrary order where it is unnecessary. A parallel implementation then has the freedom to handle the elements in whatever order happens to be convenient, which is often much more efficient than having to maintain a strict ordering.

Data structures based on sets and mappings are very much less common in programming languages than arrays and lists (data structures such as Pascal sets are not counted as they are so limited in practice as to be essentially useless for the type of programming advocated here), so there is relatively little experience of their use, even with sequential implementations. No general purpose programming language yet exists which uses sets or mappings as its basic type of data structure and which is suitable for writing programs with a high degree of d-c parallelism. This is an area in which much more research is needed: (i) to find the best set of primitives; (ii) to investigate the ease of programming with these primitives; and (iii) to find efficient implementations of the primitives on a variety of architectures.

8 References

A.V. Aho, J.E. Hopcroft and J.D. Ullman (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley.

T. Axford (1991) "An Abstract Model for Parallel Programming", *Research Report CSR-91-5*, School of Computer Science, University of Birmingham, Birmingham B15 2TT.

T. Axford and M. Joy (1991) "List Processing in Parallel", *Research Report CSR-91-8*, School of Computer Science, University of Birmingham, Birmingham B15 2TT.
J. Backus (1978) "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Comm. ACM*, **21**(8), pp. 613–41.
F.W. Burton and M.R. Sleep (1981) "Executing Functional Programs on a Virtual Tree of Processors" in *Proc. Conf. Functional Programming Languages and Computer Architecture*, New Hampshire, pp. 187–94.
M. Cole (1989) *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman.
G.C. Fox (1989) "Parallel Computing comes of Age: Supercomputer Level Parallel Computations at Caltech", *Concurrency: Practice and Experience*, **1**(1), pp. 63–103.
P.J. Hatcher, M.J. Quinn, A.J. Lapadula, B.K. Seevers, R.J. Anderson and R.R. Jones (1991) "Data-Parallel Programming on MIMD Computers", *IEEE Trans. Parallel and Distributed Systems*, **3**(2), pp. 377–83.
E. Horowitz and S. Sahni (1978) *Fundamentals of Computer Algorithms*, Pitman.
E. Horowitz and A. Zorat (1983) "Divide-and-Conquer for Parallel Processing", *IEEE Trans. Computers,* **C-32**(6), pp. 582–5.
P. Hudak (1989) "Conception, Evolution and Application of Functional Programming Languages", *ACM Computing Surveys*, **21**(3), pp. 359–411.
P. Hudak et al. (1991) "Report on the Programming Language Haskell – A Non-Strict, Purely Functional Language – Version 1.1", Yale University, Department of Computer Science.
J. Hughes (1989) "Why Functional Programming Matters", *Computer J.* **32**(2), pp. 98–107.
C.B. Jones (1986) *Systematic Software Development Using VDM*, Prentice-Hall.
P. Kelly (1989) *Functional Programming of Loosely-Coupled Multiprocessors*, Pitman.
D.L. McBurney and M.R. Sleep (1987) "Transputer-Based Experiments with the ZAPP Architecture", in *Proc. PARLE 1987*, published as *Lecture Notes in Computer Science* (ed. de Bakker et al.) **258**, pp. 242–59, Springer-Verlag.
J. McCarthy (1978) "A Micro-Manual for Lisp – Not the Whole Truth", *ACM SIGPLAN Notices* **13**(8), pp. 215–16.
J. McCarthy (1978) "History of Lisp", *ACM SIGPLAN Notices* **13**(8), pp. 217–23.
G. Marino and G. Succi (1991) "Functional Programming with Bags", *Internal Report*, DIST, Università di Genova, via Opera Pia 11a, 16145 Genova, Italy.
Z.G. Mou (1990) "Divacon: A Parallel Language for Scientific Computing Based on Divide-and-Conquer", in *Proc. 3rd Symp. Frontiers Massively Parallel Computation*, IEEE.
Z.G. Mou and P. Hudak (1988) "An Algebraic Model for Divide-and-Conquer and Its Parallelism", *J. Supercomputing,* **2**, pp. 257–78.
N.D. North (1990) "An Implementation of Sets and Maps as Miranda Abstract Data Types", *NPL Report DITC 162/90*, National Physical Laboratory, Teddington, TW11 0LW.
C.M. Pancake (1991) "Software Support for Parallel Computing: Where Are We Headed?", *Comm. ACM*, **34**(11), pp. 52–64.
F.J. Peters (1981) "Tree Machines and Divide-and-Conquer Algorithms", in *CONPAR81; Proc. Conf. Analysing Problem-Classes Parallel Computing*, pp. 25–36.
F.P. Preparata and S.J. Hong (1977) "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", *Comm. ACM*, **20**(2), pp. 87–93.
F.P. Preparata and J. Vuillemin (1981) "The Cube-Connected Cycles: A Versatile Network for Parallel Computation", *Comm. ACM,* **24**(5), pp. 300–9.

F.A. Rabhi and G.A. Manson (1990) "Experimenting with Divide-and-Conquer Algorithms on a Parallel Graph Reduction Machine", *Research Report CS-90-2*, Department of Computer Science, University of Sheffield, Sheffield S10 2TN, U.K.

J.T. Schwartz, R.B.K. Dewar, E. Dubinsky and E. Schonberg (1986) *Programming with Sets: An Introduction to SETL*, Springer-Verlag.

D.B. Skillicorn (1991) "Practical Concurrent Programming for Parallel Machines", *Computer J.*, **34**(4), pp. 302–10.

J.M. Spivey (1989) *The Z Notation: A Reference Manual*, Prentice-Hall.

P.L. Treadway (1990) "The Use of Sets as an Application Programming Technique", *ACM SIGPLAN Notices*, **25**(5), 103–16.

D.A. Turner (1985) "Miranda: A Non-Strict Functional Language with Polymorphic Types", in *Proc. 1985 Conf. Functional Programming Languages and Computer Architecture*, published as *Lecture Notes in Computer Science* **201**, pp. 1–16, Springer-Verlag.

Contents