A Higher-Order Approach to Parallel Algorithms

P. G. HARRISON
Department of Computing, Imperial College, London SW7 2BZ

A unified approach to the development of algorithms tailored to various classes of parallel computer architecture is presented. The central theme is to identify a small set of higher-order functions that can be implemented efficiently on the target architecture and which can be used to express parallel algorithms – in general via mechanised program transformation from some higher-level specification. Such higher-order functions enable generic programs to be written in which much parallelism may be explicit. Although the analysis uses purely functional languages, it is the functional paradigm that is important and not the particular syntax. The proposed methodology is illustrated with a numerical problem which is solved directly by a non-recursive program. We also describe schemes that map programs onto both static and dynamic MIMD architectures which have communication links which are fixed and changeable at run-time respectively.

Received March 1992, revised May 1992

1. INTRODUCTION

The main obstacle to the successful exploitation of parallel computer systems is the ease with which they can be programmed. Existing software technology is inappropriate and for such systems ever to become viable general-purpose products, it is essential that software tools be developed to reduce the inherent complexity of parallel program development. There are three aspects of software development for parallel systems:

- The potential parallelism available in a program must be identified. In some cases, the parallelism may be explicit, having been provided by the programmer, but in general it is necessary to analyse the program to establish the potential parallelism.
- The available parallelism must have a suitable granularity; that is, each task must be of an appropriate size for the type of machine being programmed. The grain size must be small enough to utilise several processors at once yet large enough to avoid excessive communication overhead. Large grain also makes a program more portable, enabling it to be executed efficiently on a wide range of multi-processors.
- The resulting parallel program must be partitioned into sequential units of computation which must be scheduled to run efficiently on the available processors.

The present paper applies the functional programming paradigm and a program transformation methodology to address these issues in a uniform way. There exist a number of classes of parallel algorithm and a number of widely differing types of parallel architecture, ranging from SIMD, through tightly coupled MIMD with shared memory to loosely coupled multi-processors. The central idea of this paper is to identify forms of functional programs that suit particular architectures, to generate parallel algorithms by composing them and to develop program transformations to synthesise such algorithms from higher-level specifications. Here, the higher-level specifications will also be functional programs, but the objective in the longer term is to develop the paradigms which would become applicable to any flavour of source language.

In section 2, we demonstrate how the use of higher-order functions in a functional program can produce non-recursive forms for many non-trivial algorithms. This results in clear and concise programs which often solve a generic class of problems through their parameterised form; the genericity comes from the function-valued arguments of the higher-order functions. We also describe a transformational methodology for tailoring functional programs to parallel architectures. We then consider in the following two sections more concrete problems. We show how to transform divide-and-conquer algorithms into both dynamic and static MIMD forms, in which the communication structure is respectively variable and fixed throughout a program's execution. In the former case, considered in section 3, the resulting abstract architecture is essentially parallel graph reduction of the type realised by ALICE.[13] An example is given which throttles the available parallelism. In section 4, it is shown how divide-and-conquer algorithms can also be mapped onto pipelines, a special case of the static MIMD architecture. The paper concludes in section 5.

We present much of our analysis in a combinator-based language in the FP style.[1] The notation is reminiscent of APL, and for our numerical example of section 2, we introduce several more of the APL built-in functions, hopefully to appeal to a wider readership. The primitive combinators – higher-order functions – include function composition (denoted by $_\circ_$), conditional (denoted by $_\rightarrow_;\ _$) and function-tupling called construction (denoted by $[_,_\ldots_,_]$). Function application is denoted by juxtaposition, e.g. $f x$. Although this syntax is essentially first-order, those higher-order functions which have function-valued arguments but do not return functions as results can be expressed by including the apply function as a primitive. Apply takes a function and an object as arguments and returns the result of applying the function to the object.

2. A HIGHER-LEVEL PARADIGM

2.1. Use of higher-order functions

Functional languages provide an ideal medium for implementing many of the principles of good software engineering. This is largely a consequence of their high-

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

level, declarative nature and the hierarchical structure of functional programs.[6] However, such properties are not unique to functional languages and the same (functional) paradigm can also be applied to more conventional languages which may include imperative features. The paradigm merely requires a language to possess some means for defining functions (including higher-order functions). Certainly, other features of functional languages such as strong typing and pattern matching are desirable since they assist the rapid production of correct, easy-to-read programs. However, they may be too expensive for a large organisation to incorporate into their software production process, in terms of staff retraining, development of new systems software and interaction with other software components written under conventional methodologies. In the sequel, we will talk about functional programming, but it is the paradigm that we advocate above all.

Whilst a first-order functional language can be used to produce good software, far greater expressive power is available to a language with higher-order functions. These can take functions for arguments and return functions as results, and the former capability alone can endow a language with the power to define generic algorithms, parameterised by some operation. For example, consider the factorial function. Whilst its usual recursive definition is mathematically rigorous and concise, it is not the intuitive definition which is something more like 'multiply all the numbers from 1 up to the number given as argument'. When defining an operation on a data structure – here a list of numbers – we frequently would like to iterate over it in some way with an operator. Moreover, we might want to iterate over it in the same way with various different operators, for example we might want to add up all the numbers from 1 to the argument value. It would be nice to be able to avoid writing a whole new program for each such operator; we would like to have the operator as a parameter. This capability is exactly what a higher-order function (of the first kind above) provides.

To summarise, we would like to avoid the use of explicit recursion, which should be replaced by a higher-order function. Of course, we do not really remove the recursion in this way; it just becomes hidden in the higher-order function which itself has a recursive definition, although typically is implemented as a primitive. In fact just a few higher-order functions, together with some well-chosen first-order primitives, suffice to solve a wide range of problems for each composite data type. This philosophy is not new; it is very close to that of APL and later FP. The only real difference from the programming point of view is the ability for users to define arbitrary higher-order functions. Even then, we favour the extensive use of a small suite of such functions, corresponding to a small number of program-forming structures in 'structured programming'. However, a major advantage, if a functional language or a pure functional style is used, is that we have better scope for optimisation. This is based upon the extensive set of meaning-preserving transformation techniques developed for functional languages, which derive from the simple semantics of these languages and in particular their referential transparency; see ref. 6 for example.

Two of the most important higher-order functions which iterate over linear structures such as lists, sequences or vectors are map and reduce which we will often denote respectively by * (in postfix form) and / (in prefix form). To be concrete, we will take lists for our linear structure and define map: $(\alpha \to \beta) \to \text{list } \alpha \to \text{list } \beta$ by

$$\text{map f nil} \equiv \text{f}^* \text{ nil} = \text{nil}$$
$$\text{map f (x::xs)} \equiv \text{f}^* \text{ (x::xs)} = \text{(f x)::(f}^* \text{ xs)}$$

where nil denotes the empty list and :: is the infix form of the list constructor function 'cons'. Similarly, right-fold or right-reduce, written $/ : \beta \to (\alpha \to \beta \to \beta) \to \text{list } \alpha \to \beta$, is defined by

$$\text{/b f nil} = \text{b}$$
$$\text{/b f x::xs} = \text{f x (/b f xs)}$$

(We adopt the convention that :: has the highest precedence in expressions where there is ambiguity due to absence of brackets.) For lists of integers, we define, as in APL, the function 'iota', denoted by ι, that returns the list of integers from 1 to n when applied to argument n. We omit its obvious recursive definition. Then, for example, factorial (!) is defined by

$$! \text{ n} = / \text{ 1} \times (\iota \text{ n})$$

and the function that sums the numbers up to n (sum) is defined by

$$\text{sum n} = / \text{ 0} + (\iota \text{ n})$$

(Actually, since we are using right-fold, we are multiplying or adding the numbers from n down to 1, but since + and × are commutative, this is the same as working upwards from 1. If + and × were not commutative, we could define a left-fold function similarly and use it to work upwards from 1 instead.)

If we also introduce the function-composition higher-order function, $\circ$, these definitions become even simpler, namely

$$! = (/1 \times) \circ \iota \text{ and sum} = (/0+) \circ \iota$$

Denoting the constant function which always returns the result x by $\bar{\text{x}}$ (following Backus's notation[1]), the function length which returns the length of a list is defined by

$$\text{length} = (/0+) \circ \bar{1}*$$

Not only are these definitions intuitive (and easy to read when you have got used to the notation, recall APL!), they also avoid duplication of work since they allow the definition of generic functions which iterate over a data structure with whatever functions and values are supplied as arguments. Here we have parameterised reduce to multiply numbers, add them or count the number of items in a list. A more complex example which triangulates a matrix is given in the next sub-section.

We advocate this approach to software development whether or not the target machine is a parallel computer. However, for the parallel case, the absence of recursion can simplify greatly the identification of parallelism and partitioning. For example, wherever map is applied to a known function, we can distribute the computations over its list argument. It will be noted that reduce is inherently sequential in nature and so cannot be parallelised in general. However, if it is applied to an associative operator, it can be applied to segments of a list independently and the results combined in a divide-and-conquer fashion.

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

To sum up, our paradigm for parallel (but not only parallel) functional programming is to:

(*a*) Define the appropriate data structures, e.g. vectors, matrices, trees, as in normal programming practice.
(*b*) Iterate over these structures using higher-order functions rather than explicit recursion. The higher-order functions must be tailored to the structures and should be few in number. Typically they will include variants of map or reduce and for standard data structures, such as vectors or lists, should be provided as primitives. The same applies to commonly required first-order functions; see the example in the next section. This reduces the need for user-defined functions and increases efficiency.

2.2. A numerical example

To exemplify the previous discussion, let us consider operations on vectors and matrices. First, we introduce the data type vector, denoted by angle brackets $\langle ,\ldots, \rangle$, which is equivalent for our purposes to the type list. We assume the following functions are primitive:

- cons (with infix form ::)
- index (with infix form of)
- append (with infix form ||)
- length

Of course, all but the constructor cons could be defined recursively by the programmer. We also introduce the following functions for manipulating vectors. These too could be defined by the user but more likely would be primitive, as in APL, for example, whence many are taken. All of these functions are undefined on arguments which do not have the type specified.

- take (with infix form ↑), defined by
  $n \uparrow \langle x_1, \ldots, x_m \rangle = \langle x_1, \ldots, x_n \rangle \quad (n \leqslant m)$
  $= \langle 0, \ldots, 0, x_1, \ldots, x_m \rangle \quad (n > m)$
  (n − m zeroes)
- drop (with infix form ↓), defined by
  $n \downarrow \langle x_1, \ldots, x_m \rangle = \langle x_{n+1}, \ldots, x_m \rangle \quad (n < m)$
  $= \langle\,\rangle \quad (n \geqslant m)$
- copy, defined by
  copy n x = $\langle x, \ldots, x \rangle$ (a list of length n, each item equal to x)
- inner_product (with infix form •), defined by
  $\langle x_1, \ldots, x_n \rangle \bullet \langle y_1, \ldots, y_n \rangle > = (x_1^* y_1) + \ldots + (x_n^* y_n)$
  $(n \geqslant 0)$

transpose, also written Γ, defined by

$$\Gamma\langle\langle x_{11}, \ldots, x_{1n}\rangle, \langle x_{21}, \ldots, x_{2n}\rangle, \ldots, \langle x_{m1}, \ldots, x_{mn}\rangle\rangle = \langle\langle x_{11}, \ldots, x_{m1}\rangle, \ldots, \langle x_{1n}, \ldots, x_{mn}\rangle\rangle \quad (m, n \geqslant 0)$$

- ι and revι, defined by
  $\iota n = \langle 1, \ldots, n \rangle$ and $\text{rev}\iota\, n = \langle n, \ldots, 1 \rangle \quad (n \geqslant 0)$

We will also use the higher-order function zip (with postfix form $*^2$), which is dyadic map, defined by

$$\text{zip } f\ (\langle x_1, \ldots, x_n\rangle, \langle y_1, \ldots, y_n\rangle) = \langle f\langle x_1, y_1\rangle, \ldots, f\langle x_n, y_n\rangle\rangle \quad (n \geqslant 0)$$

It is then easily shown that

$$\bullet = (/+) \circ (\text{zip} \times)$$

by applying each side of the equation to an arbitrary pair of equal length vectors, and that

$$\Gamma \text{ pair} = (\text{zip id}) \text{ pair}$$

when pair is a vector of two (equal length) vectors and id is the identity function.

We define a matrix to be a vector of vectors, and similarly for higher rank objects, and introduce the following additional utility functions.

- index n (a 'curried' selector function partially applied to n) to select the nth item in a list, e.g. the nth row of a matrix is defined by
  index n $\langle x_1, \ldots, x_m \rangle = x_n \quad (1 \leqslant n \leqslant m)$
  index n z = ⊥ (undefined) for any other form of argument
- col to select a column, defined by
  col = map ∘ index
- replace-row, defined in (partly) curried form by
  replace-row (n, v) m = (n − 1) ↑ m || (v :: n ↓ m)
- replace-col, defined similarly by
  replace-col (n, v) = Γ ∘ (replace-row (n, v)) ∘ Γ
- matrix-multiply (infix form ⊗) can now be defined simply by
  m1 ⊗ m2 =
  Γ (map (constr (map inner-product m1)) (Γm2))
  or
  m1 ⊗ m2 =
  constr (map map (map inner-product m1)) (Γm2)
  where constr : list($\alpha \to \beta$) → $\alpha \to$ list(β) is a coercion function defined by
  constr $\langle f_1, \ldots, f_n \rangle\, z = \langle f_1 z, \ldots, f_n z \rangle \quad (n \geqslant 0)$
  In words, constr converts a list of functions – which cannot be applied to an argument – into a construction which applies each function in the list to the supplied argument.

Notice that

constr = Λ ((map @ ∘ distr)

where disr, @ and Λ are the functions distribute right, apply and curry respectively, defined by

distr $\langle\langle x_1, \ldots, x_n\rangle, a\rangle = \langle\langle x_1, a\rangle, \ldots, \langle x_n, a\rangle\rangle$
@ $\langle f, x \rangle$ = f x
Λ f x y = f(x, y)

With a little more practice, it can also be seen that

Γ = @ ∘ [constr ∘ col* ∘ ι ∘ length, id]

We conclude this section with a concise, non-recursive function, tri, which converts a matrix into lower triangular form. It follows the conventional Gaussian elimination algorithm without pivoting, but there is no conceptual problem in writing a non-recursive version with pivoting. The definition of tri is the following:

```
tri matrix  = / matrix f (revι (d − 1))
  where d = length matrix
        f n m = (n ↑ m) || vsub* (n ↓ m)
        where vsub r = zip− ⟨r, zip × ⟨row, copy d
                                           (index n r)⟩⟩
              row = zip÷ ⟨index n m, copy d (index n
                                           (index n m))⟩
```

2.3. Principles of transformational parallel programming

In transformational programming, first a clear application-oriented solution to a problem is written, with emphasis on ease of understanding rather than

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

efficiency of execution. Meaning-preserving (i.e. semantically sound) transformations are then applied successively to this program to produce a sequence of semantically equivalent, but ever more efficient, programs. Ideally, the transformation process would be supported by some environment which would ensure that only semantically verified steps could be applied and many of these steps would be applied automatically. Although this has been achieved for some simple transformations, e.g. ref. 5, in general the process is at best only semi-automatic. Typically, sub-expressions are selected by the programmer who also decides which transformations to apply and judges their success (or otherwise). Moreover, new transformation steps can be introduced by the programmer as 'lemmas' which are specific to his particular application. This is often necessary to allow progress to be made, but relies for its correctness on that of the new lemmas. This type of interactive transformation may be supported by meta-programming, i.e. a programming language that allows the user to manipulate programs in a controlled way that ensures their correctness is preserved. This leads to a formal approach to software development.

The best-established transformation methodology of this type is the unfold/fold methodology of Burstall and Darlington,[3] which is based solely on partial (symbolic) evaluation and the substitution of equal expressions. Being based on such simple principles, its range of applicability is very wide and typically it forms the framework for a transformation environment which also includes more powerful but more specific techniques. One such technique which we will make use of is algebraic program transformation. This consists of simply rewriting combinator-expressions using equational reasoning (as an inference system) based on some set of axioms. The algebraic style has been followed by a number of researchers, for example Bird,[2] and Williams *et al.*,[17] as well as the author,[9,10] and a non-trivial application of it is required in our example of section 4. Before such a transformation system can be applied, it is necessary to prove the semantic soundness of the axioms by showing that the application of either side of an axiomatic equation to an arbitrary object yields the same result. The same approach could be applied to the expressions of a conventional functional language in which object-variables appear explicitly, but these variables often impede the transformation process since they need to be instantiated to enable certain steps to proceed, e.g. folding. We therefore abstract all object-variables to attain a form which resembles the style of FP.[1] For example, two axioms state that

$$[f, g] \circ h = [f \circ h, g \circ h] \text{ and } (p \rightarrow q;\ r) \circ h = p \circ h \rightarrow q \circ h;\ r \circ h$$

and can be verified very simply by applying each side of an axiom to an arbitrary object x and demonstrating that the results are equal. The axioms may be primitive, relating the general combining forms of the language (i.e. its basic higher-order functions) as above. Alternatively, they may be data type dependent, relating the type's constructors, selectors and higher-order functions. From the axioms, we may derive theorems, i.e. commonly used compositions of axioms. Two examples are a result defining a loop which is equivalent to a linear function and an expression for the inverse of a recursive function. Such theorems have been developed for sequential implementations; see for example refs 9, 10, 12.

In the next sections we obtain results which define equivalent parallel computations for certain kinds of function. This derivation of axioms in terms of data types and associated higher-order functions is consistent with the programming style we have advocated and the parser-based analysis which is required to determine the applicability of theorems is conducive to mechanisation. Transformational programming is particularly relevant to the exploitation of parallel machines. There is great diversity in the current range of parallel machine types available. However, there is no accepted software development methodology for parallel computation – not even for specific machines, let alone a common one. Application-oriented solutions typically do not account for the properties of the machine on which they run, and neither should they. Moreover, current parallel optimisers are too specific to both application and machine. Hence there is no real alternative to explicit control of parallelism. Of course, this is disastrous for software productivity, reliability, evolution and portability, but it is necessary, in compilers' object code, for satisfactory performance – which is the motivation behind parallel computation in the first place. Successful exploitation of parallel systems has required a return to low-level programming: the hardware is presented with manuals describing processors' languages and communications protocols and the rest is up to the user. Although it helps when the languages are at quite a high level, software tools for performing non-trivial logical tasks are lacking.

We propose to use the functional paradigm to address these problems. The methodology comprises three phases. First, a class of functions that correspond to efficient code for the target architecture is identified. Secondly, transformation techniques are developed that map application-level programs and/or specifications into functions in that class. A suite of such transformations is necessary for each class, i.e. for each type of architecture. As well as conventional source-to-source transformation, this phase includes the annotation of the transformed program to indicate such execution characteristics as process placement and scheduling. In the third phase, the applications programmer should now be able to write the most 'natural' program for the problem in question. This may already involve higher-order functions, if the preceding methodology is adhered to, and so may be suitable for parallel evaluation immediately. In general, however, we will require transformation to synthesise an equivalent program that contains more parallelism. As usual, this program would be guaranteed correct by the soundness of the transformation rules.

3. TRANSFORMATIONS FOR DYNAMIC MIMD

There are two types of function which we consider for parallel execution. The first is linear in the sense of the graph representing a partially evaluated function application. This graph grows down its left spine, reflecting a sequential computation with a number of nodes proportional to the size of the argument. Such a function application gains nothing from execution on a parallel

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

machine since there is never any work to distribute, with the possible exception of some sub-expressions in the body of the function which do not call that function recursively. However, certain linear functions can be transformed into parallel form if, in some way to be clarified, they can be 'distributed' over the set of all the arguments passed in a recursive call. On the other hand, non-linear functions often generate parallelism directly by having more than one sub-expression in their body containing a recursive call. Such functions are called divide-and-conquer functions. The evaluation graph of such a function is a tree and the number of parallel function applications grows exponentially as the graph unfolds. It is therefore relatively simple to find sufficient parallelism and the problem becomes one of throttling the generation of parallelism which, if it becomes too excessive, will cause a communication bottleneck and loss of performance. Nonlinear functions of this type are considered in the next section where parallelism is controlled by mapping the functions' applications onto a pipeline. Here we briefly address linear functions, showing how they can be transformed into divide-and-conquer form under certain conditions, whereupon throttling will again be required. Throttling is typically implemented through annotations to the compiled code of a program which indicate, for example, the smallest size of argument for a function that makes remote evaluation advantageous. The information conveyed by the annotation is based on some form of complexity analysis performed at compile time. Moreover, the run-time decision also depends on the expected communication time overhead which should be predicted from the current load on the system using some performance model. These issues of partitioning and scheduling are vitally important and difficult problems, but we do not address them further in this paper.

To be concrete in our parallelisation of linear functions, we consider list-manipulation functions which have a tendency to be sequential because of the linear (sequential) definition of lists:

list α = nil ++ cons (α, list α)

The transformation of such a function into a parallel form is exemplified by the recursive function that computes the length of a list. Let us call this count to distinguish it from the already defined length function. Its definition is:

count nil = 0
count x::xs = 1 + count xs

which is equivalent to the FP-like definition

count = is_nil $\rightarrow \bar{0}$; succ $\circ$ count $\circ$ tl

where is_nil is the function that tests for an empty list, succ is the integer successor function and tl is the function that takes the tail of a non-empty list. To achieve a parallel computation, we need a non-linear tree-structure instead of a list, and we define

tree α = leaf(list α) ++ node(tree α, tree α)

For example, one tree corresponding to the list ⟨A, B, C, D, E, F⟩ is shown in Figure 3.1. Of course, this tree is not unique, and typically we choose a balanced tree, with list segment sizes as near to equal as possible at the tips, for an efficient parallel computation. The optimal segment size is a function of the partitioner. It should be just small enough so that the communication overhead of distributing the function application on it makes it cheaper to evaluate locally. As we have said, determination of this size requires complexity and performance analysis.

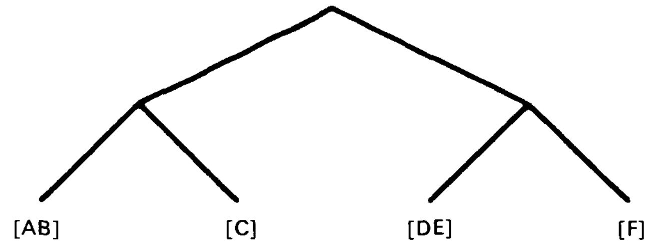

Figure 3.1. A tree of segments of a list.

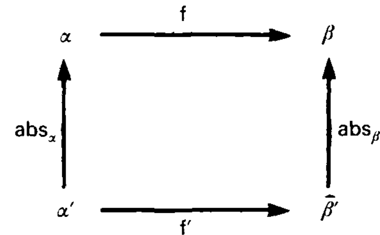

Figure 3.2. General data type transformation.

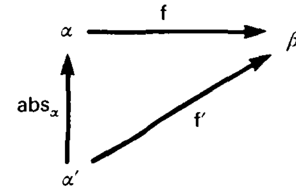

Figure 3.3. 'Triangular' data type transformation.

For parallel evaluation of the application of a function on lists, count in our case, we need to find a corresponding function on trees. This type of synthesis is often called a data type transformation. In the general problem, we are given abstraction functions, $\text{abs}_\alpha: \alpha' \rightarrow \alpha$, which link the concrete forms of type α' with the abstract forms of type α. We can then draw a commutative square, shown in Figure 3.2, which defines a concrete function $f': \alpha' \rightarrow \beta'$ corresponding to the given (abstract) function $f: \alpha \rightarrow \beta$. Thus, $f' = \text{abs}_\beta^{-1} \circ f \circ \text{abs}_\alpha$ and the problem is to express the right-hand side of this equation in a form such that no function involves an abstract type;[11] as it stands, all three do so! In many cases either $\alpha = \alpha'$ or $\beta = \beta'$ and the square becomes a triangle (see Figure 3.3), greatly simplifying the transformation to $f' = f \circ \text{abs}_\alpha$.

In our case, $\beta = \beta'$ = num, the type of non-negative integers, and abs_α, abbreviated by abs, is defined by

abs leaf(x) = x
abs node(s, t) = (abs s) ‖ (abs t)

which has equivalent FP definition

abs = is_leaf $\rightarrow$ id; append $\circ$ abs* $\circ$ [lefttree, righttree]

where is_leaf is the predicate that tests if a tree-object is a leaf and lefttree, righttree are selector functions on trees formed by the node constructor. (Notice that if we

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

generalised * to map over the tree constructor, node, we could just write abs* instead of abs* ∘ [lefttree, righttree]). Now, it is easy to see that if, given $f: \alpha \to \beta$,

f ∘ append = a ∘ f* for some function a (which depends on f)

we can synthesise a function $f' \alpha' \to \beta$ independently of α. In our case, this condition holds with a = +, integer addition, since

count (x ‖ y) = (count x) + (count y)

The concrete function f′ is defined by

f′ = f ∘ abs =
is_leaf → f ∘ id; f ∘ append ∘ abs* ∘ [lefttree, righttree]

by the basic FP law f ∘ (a → b; c) = a → f ∘ b; f ∘ c for all functions f, a, b, c. Thus, if the condition stated above is satisfied,

f′ = is_leaf → f; a ∘ f* ∘ abs* ∘ [lefttree, righttree]
  = is_leaf → f; a ∘ (f ∘ abs)* ∘ [lefttree, righttree]

since (f ∘ g)* = f* ∘ g* for all functions f, g (see section 4.2). Thus we arrive at the definition

f′ = is_leaf → f; a ∘ f′* ∘ [lefttree, righttree]

Although this definition still contains a reference to f and so is not entirely independent of the type α, this reference occurs only in the 'base case' and not in the recursive branch. Thus we only need to retain the abstract function for computations at the leaves of a tree. In fact we could even avoid this if we chose a tree representation with only singleton lists at the leaves. Then we would need only the base case of f for application to the values in the leaves. However, this would lead to too fine a grain of parallelism for our purposes.

To be rigorous, the function we have derived for f′ is the least fixed point of the functional

λg . (is_leaf → f; a ∘ g* ∘ [lefttree, righttree])

This is certainly one valid concrete form for f, by construction, but we have not shown that it is unique. Indeed, in general, this is not the case.

Finally, in our example, the parallel form of count, defined on trees, is tree_count (≡ count′) which may be defined as

```
tree_count leaf(x)      = count x
tree_count node(x, y) = (tree_count x)
                                  + (tree_count y)
```

This is a classic divide-and-conquer function of the type we consider in the next section; in fact it could now be transformed, if desired, into a pipeline for evaluation on a static MIMD architecture. The base case of the recursion still has a call to count which would be evaluated sequentially in single processors. The linear recursion could be optimised by applying sequential transformations such as the conversion of linear recursion into a loop; see, for example, ref. 12.

4. TRANSFORMATIONS FOR STATIC MIMD

In static MIMD architectures, processes cannot be created or destroyed dynamically and the communication links between the fixed set of processes never change. Such an architecture can be described abstractly as a process network in a functional language,[8,15] by defining a set of mutually recursive functions. The functions correspond to the processes, their parameters to the communication links and the data transmitted to the components of the (list-valued) arguments; such lists may be infinite and are often called streams. A process network that includes a cycle will generate an infinite stream of data and must be described by a functional program with lazy semantics. We will consider only process networks with no cycles, i.e. directed acyclic graphs and it is easy to see that any such network can be drawn as a pipeline. The order of the processes in the pipeline is any one that is consistent with the precedence relation of the graph. Any stages in the pipeline that are skipped can be combined with other stages which will acquire a larger number of input and output channels. Of course, there are still implementation problems, such as a mismatch of data rates at different stages where, for example, one stage might require data items two at a time from its input stream whilst the others consume them singly. However, we will not be concerned with such low-level problems here (which is not to deny their difficulty).

We consider divide-and-conquer algorithms f specified as functional programs of the form

f x = **if** p x **then** q x **else** $E_{f,x}$

where q is a fixed function, i.e. not dependent on f, p is a boolean-valued fixed function and $E_{f,x}$ is an expression in the variables f and x. Now, $E_{f,x}$ must contain at least two occurrences of f or else there could be no 'divide' aspect to the specification. The resulting evaluation graph is then a tree which is balanced down to instances of the base case q x. Clearly applications of such a function are ideally suited to implementation on a dynamic MIMD architecture without transformation other than annotation to facilitate throttling, as discussed in the previous section. We now show how to transform such functions for implementation on a static, pipeline architecture.

In the FP-like notation referred to in the Introduction, the definition of the functions under consideration becomes:

f = p → q; Hf

where (Hf) x ≡ $E_{f,x}$. As observed above, H is non-linear in that Hf contains more than one occurrence of f and we consider a simply defined, yet substantial, extension of the linear class of functions considered in ref. 12. To illustrate our approach, we first consider a simple example, namely mergesort, in section 4.1. This is then followed by the more formal, general analysis in section 4.2.

4.1. Transformation into a pipeline: an illustrative example

The 'mergesort' function sorts a list of objects with an order relation, e.g. the integers with 'less than'. It works by first splitting the list into two parts, recursively sorting the parts and finally merging the two sorted lists that result. The idea of our transformation is first to convert the recursive program into a non-recursive form (using higher-order functions) and then map this into a pipeline

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

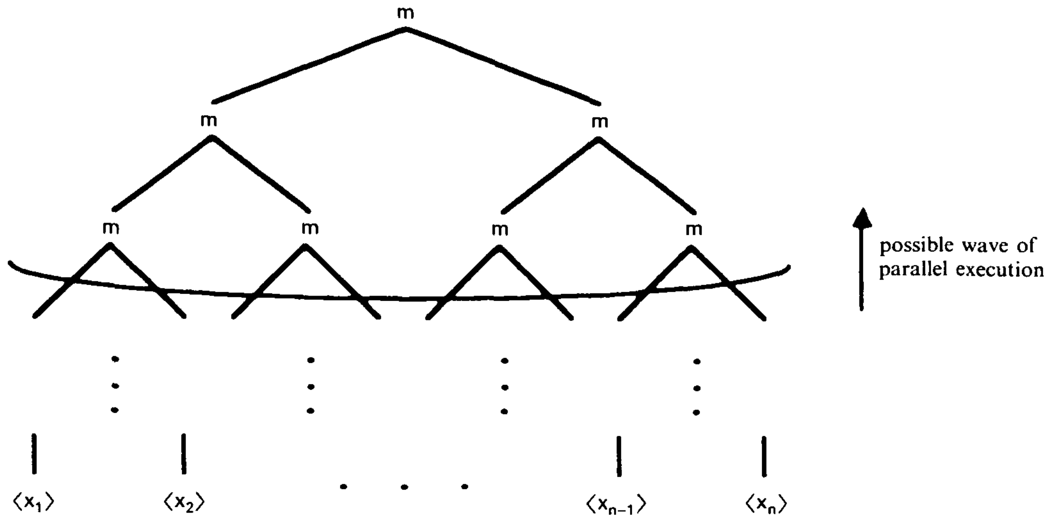

Figure 4.1. Dynamic MIMD computation of 'mergesort'.

by a simple data type transformation. The definition of the mergesort function, ms, is the following:

mx⟨x⟩ = ⟨x⟩
ms xs = merge (ms u, ms v)
  where (u, v) = split xs

where split is the function that divides a list into two parts 'as equally as possible', defined by:

split xs = (u, v) **such that**
  xs = u ∥ v **and** $0 \leqslant$ length u − length v $\leqslant 1$

and merge is the function that merges two sorted lists into a single sorted list, defined by:

merge (nil, ys) = ys
merge (xs, nil) = xs
merge (x::xs, y::ys)
  = **if** x < y **then** x::merge(xs, y::ys) **else** y::merge(x::xs, ys)

For simplicity, we assume ms is applied to a list with length n a power of 2 so that its evaluation tree is perfectly balanced; see Figure 4.1. The computation consists of two phases. In the first, the tree unfolds until the leaves contain singleton lists. In the second phase, the tree folds as the merge functions, abbreviated to m in the figure, are applied to their arguments across each level in the tree, progressively working upwards. The computation terminates when the fold phase applies the merge at the root of the tree. We wish to construct a pipeline to perform the fold phase, given the singleton lists at the bottom of the fully unfolded tree as input.

Each stage in our pipeline will correspond to one level in the tree and so we begin by defining a function that maps the collection of lists entering the merge functions at any level into the corresponding collection entering the next level up. Each collection may be viewed as a list of pairs (of lists), each pair constituting the arguments to one instance of merge. The required function, d-stage say, is therefore defined by:

d-stage m ⟨p⟩ = m p
d-stage m ps = pairs (m* ps) = (pairs ∘ m*) ps if length ps > 1

where the function pairs is defined by

pairs nil = nil
pairs x::y::t = (x, y)::pairs t

In fact, we can just define d-stage m = pairs ∘ m* if we extend the definition of pairs to singleton lists by the equation

pairs ⟨p⟩ = p

To define mergesort in a non-recursive way, all we have to do is to 'fold' d-stage m over the $1+\log_2(n)$ levels of the tree. That is, we require a $\log_2(n)$-fold composition of d-stage m, i.e. of pairs ∘ m*, which is given formally by the function comp as follows:

comp f⟨ x⟩ = f ⟨x⟩
comp f xs = comp f (f xs) = ((comp f) ∘ f) xs if length xs > 1

The mergesort function is now defined non-recursively by

ms = (comp (pairs ∘ m*)) ∘ pairs

This is all very well, but we have not performed any transformation yet. The computation is exactly the same as for the original recursive definition and proceeds 'dynamically' in that the number of parallel processes is determined by the function pairs. The basis for it is the successive mapping of a list of pairs. Since the list is of unknown size, the degree of parallelism is also unknown before run-time and will vary from stage to stage in the computation. This suggests considering instead a mapping on a pair of lists. This would enable the compilation of functions of two arguments (corresponding to the pairs) for each stage, which would process a stream of unknown length of pairs at run time.

In other words, we need a data type transformation of the kind described in section 3 to map (abstract) lists of pairs to (concrete) pairs of lists. This is a common transformation, for example used in VLSI design, for which the abstraction function, abs, is transpose (Γ), provided we regard pairs as lists of two items. It is also analytically very convenient since transpose has a number of useful properties, not least of which is that it is its own inverse.

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

Now, ms = (comp (pairs ∘ m*)) ∘ pairs, and it is easy to verify that the concrete form of a composition of functions is the composition of their concrete forms. Thus the concrete form of mergesort, ms′, is defined by:

ms′ = (comp (pairs′ ∘ m*′)) ∘ pairs′

Thus we need to find expressions for pairs′ and m*′. Now, pairs′ is given by a 'triangular' transformation since the argument type is the same for both pairs and pairs′, namely a list. Thus,

pairs′ = Γ ∘ pairs

(The 'triangle' is a different one from that in Figure 3.3). Thus, pairs′ takes a list as argument and outputs the transpose of the list of pairs returned by pairs: assuming the input list contains an even number of items. This value is the pair of lists formed by adding successive items of the input list at the end of alternate lists in the pair. If the argument is a singleton list, no data type transformation is involved so pairs and pairs′ both produce the value of the list-item.

Similarly, m* and m*′ share the same result type and so, as in Figure 3.3, we have

m*′ = m* ∘ Γ = m*2

(see Lemma 4.5). Our final form for mergesort is therefore

ms′ = (comp (pairs′ ∘ m*2)) ∘ pairs′

which is a multiple composition realisable by a pipeline. Each stage, other than the first, in the pipeline is the same and has two input streams of sorted lists. It merges corresponding lists in the input streams (according to m*2) and outputs the results on alternating output streams (according to Γ). These stages are depicted in Figure 4.2. The first stage just applies pairs′ to the list of singleton lists formed from the input to ms by applying [id]*. It will also be apparent that the last stage could be simplified to m.

Before giving the formal analysis in the next section, let us summarise the main steps we have taken to synthesise a pipeline for a divide-and-conquer type of function:

- determine the evaluation tree of the function, including the values in the leaves;
- transform the function into non-recursive form by determining the mapping between levels in the tree and folding over all levels;
- apply a data type transformation from lists of pairs (in general tuples for functions of arity greater than 2 at the nodes of the tree) to pairs (or tuples) of lists.

The functions on tuples then define the stages in the pipeline, the length of which is the depth of the evaluation tree which, of course, will not be known until run-time. However, the code for each stage can be generated at compile time, and a fixed number of stages loaded, some or all of which could be multiplexed to provide several logical stages. For example, the pipeline could be implemented by one physical stage, the outputs of which would be fed back into its inputs for a fixed number of cycles. Of course, a one-stage pipeline would not provide any speed-up since there would be no parallelism.

4.2. The general result

The mergesort example was particularly simple because the only references to the argument xs in the recursive equation also appear in the argument expression for the recursive call to the mergesort function itself. To generalise and make rigorous the preceding analysis, we first define the class of functions we are considering. This is an extension of the linear class defined in ref. 12.

Definition 4.1

A divide-and-conquer functional H (DCF), together with its predicate transformer functional H_t and its body-function E_H, is defined inductively to be any of the following cases:

(a) $H = ID$	$H_t = ID$	$E_H = 1$
(b) $Hf = (Gf) \circ a$	$H_t f = (G_t f) \circ a$	$E_H = E_G \circ [1, a \circ 2]$
(c) $Hf = a \circ (Gf)$	$H_t = G_t$	$E_H = a \circ E_G$
(d) $Hf = [g_1, \ldots, g_n]$ where $g_i = a_i$ or $H_i f$	$H_t = H_{it}$ for any i such that $g_i = H_i f$	$E_H = [F_1, \ldots, F_n]$ where $F_i = a_i \circ 2, E_{H_i}$ for $g_i = a_i, H_i f$ respectively
(e) $Hf = p \rightarrow Af; Bf$	$H_t f = p \rightarrow A_t f; B_t f$	$E_H = p \circ 2 \rightarrow E_A; E_B$
(f) $Hf = Pf \rightarrow Af; Bf$ where $P_t = A_t = B_t$	$H_t = P_t$	$E_H = E_P \rightarrow E_A; E_B$
(g) $Hf = [H_1 f, \ldots, H_n f]$	$H_t f = \{H_{1t} f, \ldots, H_{nt} f\}$	$E_H = [E_{H_1} \circ [1' \circ 1, 2], \ldots, E_{H_n} \circ [n' \circ 1, 2]]$

The functional {...} is defined by

$$\{f_1, \ldots, f_n\}\, x = \langle\!\langle f_1 x, \ldots, f_n x \rangle\!\rangle \quad (n \geqslant 1)$$

Objects of the form ⟪...⟫ are called ⟪ ⟫-trees which are isomorphic to untyped lists of the form ⟨...⟩. The corresponding selector functions, denoted by i', are defined by:

$$i' \langle\!\langle x_1, \ldots, x_n \rangle\!\rangle = x_i \quad (1 \leqslant i \leqslant n)$$
$$= \perp \quad \text{otherwise}$$

The reason we introduce {...} rather than just using [...] is to allow for the possibility of having ⟨ ⟩-lists at the tips of a nested ⟪ ⟫-tree structure. In particular, applications of $H_t\, id$ must be distinguishable from normal lists. It is case (g) of Definition 4.1 that allows non-linear functions to be defined, but we are not restricted to functionals explicitly in this form. For example, the functional H defined by:

$$Hf = [[Af, a], [Bf, Cf], b] \circ c$$

for DCFs A, B, C and fixed functions a, b, c, can be rewritten as:

$$Hf = [[1 \circ 1, a \circ 2], [2 \circ 1, 3 \circ 1], b \circ 2] \circ [[Af, Bf, Cf], id] \circ c$$

We define a DC-function to be one defined by an equation of the form $f = p \rightarrow q; Hf$ where H is a DCF and p, q are fixed functions. The DCFs constitute an extensive subclass of the degenerate multi-linear functionals studied in refs 9, 14, 16. We define the degree of a DCF

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

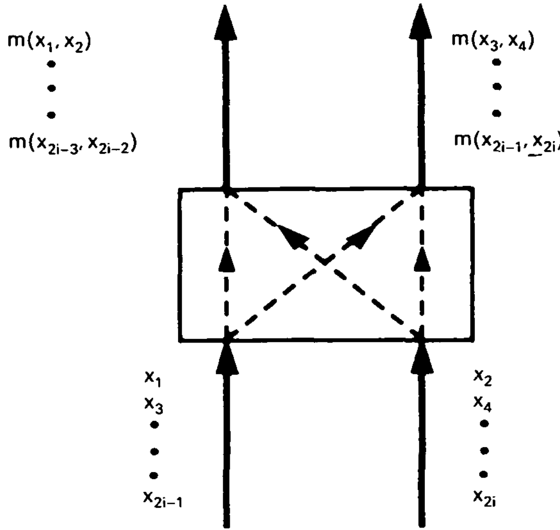

Figure 4.2. The pipeline implementation.

H to be m if Hf can be written in the form $\{f_1, \ldots, f_m\}$ for some positive integer m and functions $f_i (1 \leqslant i \leqslant m)$.

The transformation into non-recursive form requires some higher-order functions, some of which correspond to those used in the mergesort example of the previous section. They are defined as follows:

Definition 4.2

(*a*) The 'deep map function' deep is defined by the recursion equations:

$$\text{deep}\, f \ll x_1, \ldots, x_n \gg \; = \; \ll \text{deep}\, f\, x_1, \ldots, \text{deep}\, f x_n \gg$$

deep $fz = fz$ if z is not a $\ll\gg$-tree, i.e. $z \neq \ll x_1, \ldots, x_n \gg$ for any $x_1, \ldots, x_n$

(*b*) The postfix function $\wedge : (\alpha \times \beta \to \gamma) \to ((\varepsilon \to \alpha) \times (\varepsilon \to \beta) \to (\varepsilon \to \gamma))$ lifts a function of two arguments. The lifted form of a function b is denoted by $b^\wedge$ and defined by

$$f\, b^\wedge\, g = b \circ [f, g]$$

(*c*) repeat: $(\alpha \to \alpha) \to \alpha \to \text{list}_\infty\, \alpha$ is defined by

$$\text{repeat}\, f = [id] \,\|^\wedge\, (\text{repeat}\, f) \circ f) \equiv [id, f, f^2, \ldots]$$

(*d*) trunc: $(\alpha \to \text{bool}) \to \text{list}\ \alpha \to \text{list}\ \alpha$ truncates a list at the first element that satisfies the predicate passed as first parameter and is defined by

$$\text{trunc}\ p = \text{null} \to \overline{\text{nil}};\ p \circ hd \to [hd];\ [hd] \,\|^\wedge\, (\text{trunc}\ p) \circ tl$$

(*e*) upto: $(\alpha \to \text{bool}) \to (\alpha \to \alpha) \to \alpha \to \text{list}\ \alpha$ first generates a list by repeated application of its second argument, then truncates it using the predicate of its first argument and finally returns the reverse of this list (for future mathematical convenience). Its definition is

$$\text{upto}\, pf = \text{reverse} \circ (\text{trunc}\, p) \circ (\text{repeat} f)$$

where reverse is the usual list-reversal function. Alternatively, we can define upto without appealing to any laziness in the semantics of our language (repeat produces infinite lists) by

$$\text{upto}\, pf = p \to [id];\ ((\text{upto}\, pf) \circ f) \,\|^\wedge\, [id]$$

Thus, in particular, if eq0 denotes the predicate that tests an integer value for zero and sub1 subtracts 1 from an integer, upto eq0 sub1 is the reversed 'iota' function of APL which delivers the first $n+1$ non-negative integers in reverse order when applied to argument n.

(*f*) $\backslash : (\beta \times \alpha \to \beta) \to \beta \times \text{list}\ \alpha \to \beta$ is a left-fold function, defined on finite lists by

$$\backslash f (b, \text{nil}) = b$$
$$\backslash f (b, \langle x_1, \ldots, x_n \rangle) = f(\backslash f (b, \langle x_1, \ldots, x_{n-1} \rangle), x_n)$$

$(n \geqslant 1)$

This function is dual to right-fold, /, defined in section 2, with some uncurrying introduced for future convenience of notation.

(*g*) The m-adic map function $*^m$ is defined by

$$f^{*m} = \text{null} \circ 1 \to \overline{\text{nil}};\ [f \circ hd^*] \,\|^\wedge\, (f^{*m} \circ tl^*)$$

where f is an m-argument function with type $\alpha \times \ldots \times \alpha \to \beta$. Note that $*^1 \equiv *$.

(*h*) We also define the first-order function match that 'pairs' two $\ll\gg$-trees of compatible shapes by:

$$\begin{aligned} \text{match}\ \langle u, v \rangle &= \langle u, v \rangle \text{ if } u \text{ is not a } \ll\gg\text{-tree or } v \text{ is not a } \ll\gg\text{-tree} \\ &= \ll \text{match}\ \langle x_1, y_1 \rangle, \ldots, \text{match}\ \langle x_n, y_n \rangle \gg \text{ if } u = \ll x_1, \ldots, x_n \gg \text{ and } v = \ll y_1, \ldots, y_n \gg \\ &= \bot \quad \text{otherwise} \end{aligned}$$

□

Notice that we are working with untyped lists so that we make no distinction between finite lists and tuples. In fact we can, and will, overload ‖, * and $*^m$ to operate on $\ll\gg$-trees. We also overload the transpose function Γ so that it can operate on either $\ll\gg$-trees or standard lists. The functions deep and match have some important properties given by the following lemma. The proofs are easy inductions, based on extensionality.

Lemma 4.3

(*a*) deep $id = id$
(*b*) deep∘deep = deep
(*c*) deep $(f \circ g)$ = deep f∘deep g
(*d*) deep $[f, id]$ = match∘[deep f, id]

Under appropriate conditions, a non-recursive form can be obtained for a DC function. This is given by the following theorem which is proved in ref. 7. (It can also be proved by induction fairly easily.)

Theorem 4.4

Let f be defined by $f = p \to q;\ Hf$ where H is a strict DCF of degree m which satisfies:

- $\forall x. [(H_a^i p) x \Rightarrow (H_a^{i+1} p) x]$ for $i \geqslant 0$
- $\forall x. [p\, x \Rightarrow q\, x = Hq\, x]$

where $H_a f = H_1 f \ \& \ldots \& H_m f$ and & denotes 'lifted logical-and', i.e. $\& \equiv \wedge^\wedge$. Then we have

$$f = \backslash((\text{deep}\ E_H) \circ \text{match}) \circ [(\text{deep}\ q) \circ hd, tl] \circ (\text{upto}\ (\psi p)\ h)$$

where $h = \text{deep}\ (H_t\, id)$ and ψ is defined by $\psi\ p = ((\text{deep}\, p) =^\wedge (\text{deep}\ \overline{T}))$.

The class of functions H defined in the theorem was called overrun tolerant by Williams.[16] Now, the

36-2

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

applications of h in the list generated by upto will produce $\ll\gg$-trees whereas in a pipeline we require streams. Thus we flatten the $\ll\gg$-trees in a data type transformation which defines new versions of the functions h, deep, match, and E_H. The other functions q, hd, tl, upto, \ do not apply to $\ll\gg$-trees and so are not involved in the transformation. Let the type $\ll\gg$-tree be the abstract type and its flattened form the concrete type. In this transformation we will use properties of the functions * and $*^m$ given in the following lemma.

Lemma 4.5
(*a*) For all functions f, g, $(f \circ g)^* = f^* \circ g^*$
(*b*) For all functions g and m-argument functions $f(m \geqslant 2)$, $g^* \circ f^{*m} = (g \circ f)^{*m}$
(*c*) For all functions g and m-argument functions $f(m \geqslant 2)$, $f^{*m} \circ g^{**} = (f \circ g^*)^{*m}$
(*d*) For all functions f, $f^{*m} = f^* \circ \Gamma$.

The proofs simply apply both sides of the identities to an arbitrary list (tuple of lists in cases (*b*–*d*)), and use extensionality. For other argument types both sides return the undefined object. ∎

Denoting the abstract type $\ll\gg$-tree with leaves (i.e. objects which are not $\ll\gg$-trees) of type σ by $\tau(\sigma)$ and its flattened form, a 'flat' $\ll\gg$-tree, by $\tau'(\sigma)$, we have (dropping the subscript $\tau(\alpha)$ from abs):

- abs is some function such that flatten $\circ$ abs $= id$, where flatten is defined by

  flatten x = flatten $\ll x \gg = \ll x \gg$ if x is not a $\ll\gg$-tree
  flatten $\ll x_1, \ldots, x_n \gg$ = (flatten x_1) $\|\ldots\|$ (flatten x_n) otherwise

  We call a flat $\ll\gg$-tree a $\ll\gg$-list.
- Although the inverse of flatten is not unique, we can define abs uniquely by

  abs = is_sing → hd′; abs* $\circ$ split

  where is_sing is the predicate that tests for a singleton $\ll\gg$-tree and split is determined by $H_t id$, corresponding to the way in which the $\ll\gg$-tree is built up; see ref. 7 for a rigorous description.
- $h' : \sigma \rightarrow \tau(\sigma)$ is defined by h' = flatten $\circ (H_t\, id)^*$. This function will require simplification in specific cases where the definition of H_t is known. For example, for the mergesort function it reduces to id.
- deep′: $(\alpha \rightarrow \beta) \rightarrow \tau'(\alpha) \rightarrow \tau'(\beta)$ is defined by deep′ = map. However, note that the argument of deep′ (here E_H) may itself require transformation if α or β involves τ.
- match′ requires a more subtle derivation and the details are given in Appendix 1. The result is

  $$\text{match}' = \text{is_sing} \circ 2 \rightarrow \|^\wedge ; \{(\uparrow_m \circ 1) \|^\wedge (\uparrow_1 \circ 2)\} \|^\wedge \text{match}' \circ [\downarrow_m \circ 1, \downarrow_1 \circ 2]$$

  where $\uparrow_m$ denotes the function 'take' that returns the first m components of a list or $\ll\gg$-tree and $\downarrow_m$ the function 'drop' that returns the remainder; compare section 2.
- E'_H is now defined by $E'_H = E_H \circ [hd', \text{abs} \circ tl']$. This transformation merely involves changing the way E_H accesses its second argument, in particular, how it uses selector functions to access a flat $\ll\gg$-tree rather than a deep one. An inductive definition is given in Appendix 2 for the general case, but the above argument is sufficient if a fully automatic program derivation is not necessary.

In fact we now have a form of pipeline as an implementation of f. Each stage is a composition of two sub-stages: match′ which produces a $\ll\gg$-list of $\ll\gg$-lists of known length $m+1$ and E'^*_H. The $m+1$ items of each component list must enter a sub-stage serially and so we transform the list of $(m+1)$-tuples into a $(m+1)$-tuple of lists of length dependent on the argument supplied to f. In other words we transpose the result of match′. This corresponds exactly to the data type transformation we performed in the mergesort example of the previous section. We therefore obtain the following function definitions:

- match″ = $\Gamma \circ$ match′
- $(E^*_H)'' = E'^*_H \circ \Gamma = E'^{*(m+1)}_H$ by lemma 4.5.

Thus $(E^*_H)'' \circ$ match″ $= E'^*_H \circ$ match′, but no further transformation of E_H is necessary since it is mapped over its list-argument: the transposition is handled by the $(m+1)$-adic mapping function according to lemma 4.5. We have therefore achieved the following pipeline:

$$f = \backslash(E'^{*(m+1)}_H \circ \text{match}'') \circ [q^* \circ hd, tl] \circ (\text{upto } (\psi p)\, h')$$

This pipeline is illustrated in Figure 4.3. In this pipeline there are two preliminary stages which form the initial input to the main pipeline from the input x supplied to the original function. The main pipeline consists of stages $i = 1, \ldots, n$ from the right, and the final result emerges as r_n. For example, in the case of mergesort, q = id, E'_H = merge $\circ [1, 2]$ and x_0 is the result of repeatedly applying h′ = flatten $\circ$ split until a $\ll\gg$-list of singleton lists is obtained. match″ batches together the elements of the $\ll\gg$-lists r_{i-1} (together with x_i which is actually redundant), into tuples (in this case, pairs) which are distributed over the inputs of the function E'_H by Γ. The inputs x_i for $i \geqslant 1$ are not used, and it is the handling of these that introduces a significant amount of the extra complexity into the general transformation. In particular, the function match′ is almost trivial without it – 'pairs' in the mergesort example. The much more complicated quicksort program, which is also easily expressed in divide-and-conquer form, does use the inputs x_i in the pipeline. However, the general transformation presented above also handles this example straightforwardly, the details being in ref. 7.

5. CONCLUSION

We have described methodologies and techniques to exploit a range of parallel architectures using functional programming and program transformation. First, many problems can be mapped onto parallel architectures if their solutions are written in the 'parallel functional style' using a small suite of higher-order functions tailored to the data structures used. This style has many other advantages, in particular allowing generic programs to be written, and is not unique to functional programming: it is also typical of the APL philosophy for example. Parallelism can be extracted from such a program directly from certain of the higher-order functions used, for example 'map'.

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

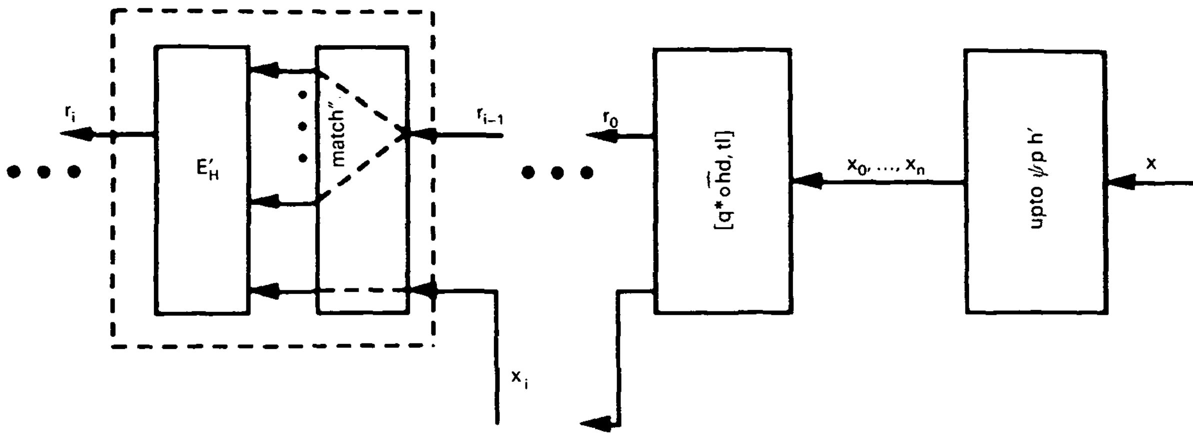

Figure 4.3. The divide-and-conquer pipeline.

However, in general, a program must be manipulated to run efficiently on a given type of parallel machine. Different transformations will be required for different machine types and the ability to transform a single program onto each would provide a uniform treatment of parallel computation. We considered two common types of parallel architecture – static and dynamic MIMD – and derived transformations tailored to each for well defined classes of functions, namely linear and divide-and-conquer. Of course, divide-and-conquer functions are well suited to dynamic MIMD architectures without transformation; the only problem is how to throttle the parallelism. This involves some form of complexity analysis and performance modelling (to account for current run-time loading) which was not addressed here.

Clearly, the results given here only begin to address the problem of parallel computation. At the higher level, we propose to identify types of function that correspond to particular modes of parallel execution, for example processor farm, message-passing and master-slave, to complement pipeline and divide-and-conquer discussed here. These would be higher-order functions (they would be parameterised by the functions in the individual processors, e.g. stages in a pipeline) called 'skeletons', and would enable source-to-source transformations to be written from applications programs.[4] There are then the problems of implementing the skeletons efficiently on specific hardware. These would be considerable, but at least the effort would not have to be repeated for every new application, as is currently the tendency.

As well as formal transformation of the type considered in this paper, other means of mapping programs efficiently onto the target machine will be necessary. The transformation system cannot hope to have as much information about program and architecture as an intelligent user. In particular, the user should be able to 'annotate' a transformed program to indicate particular properties of the architecture and execution requirements. For example, information about process placement might be important; two processes that communicate frequently should be placed on either the same processor or on two connected by a high-speed link. Similarly, a process with a lot of numerical computation should be placed on a processor with a floating point unit if available. As well as annotation, the user should be provided with the ability to apply 'new', application-specific transformation steps which should ideally be verifiable. Both the formal and informal types of program transformation need to be supported by a transformational programming environment which would have the potential to unify parallel applications, programming and architectures.

REFERENCES

1. J. Backus, Can functional programming be liberated from the Von Neumann style? *Communications of the ACM* **21** (8), 613–641 (1978).
2. R. S. Bird, *Lectures on Constructive Functional Programming*. Lecture Notes, International Summer School on Constructive Methods in Computing Science (1988).
3. R. M. Burstall and J. Darlington, A transformation system for developing recursive programs. *JACM* **24** (1) (1977).
4. M. I. Cole, A 'skeletal' approach to the exploitation of parallelism. In *Proc. CONPAR 88*, pp. 667–675. British Computer Society Workshop Series (1989).
5. J. Darlington *et al.*, A functional programming environment supporting execution, partial execution and transformation. In *Proc. PARLE 89*, pp. 365–366. Parallel Architectures and Languages Europe, Eindhoven, The Netherlands. LNCS, Springer-Verlag (June 1989).
6. A. J. Field and P. G. Harrison, *Functional Programming*. Addison-Wesley (1988).
7. I. P. Guzman, P. G. Harrison and E. Medina, submitted for publication (1992).
8. P. Henderson, *Functional Programming: Application and Implementation*. Prentice-Hall (1980).
9. P. G. Harrison, On the expansion of non-linear functions. *Acta Informatica* (1991).
10. P. G. Harrison, Towards the synthesis of static parallel algorithms: a categorical approach. In *Proc. IFIP TC2 Working Conference on Constructing Programs from Specifications*. Pacific Grove, CA, USA, (May 1991).
11. P. G. Harrison and H. Khoshnevisan, The transformation of data types. *Computer Journal* (1992).
12. P. G. Harrison and H. Khoshnevisan, A new approach to recursion removal. *Theoretical Computer Science* (1992).
13. P. G. Harrison and M. J. Reeve, The parallel graph reduction machine, ALICE. In *Proc. Workshop on Graph Reduction*. Santa Fe (September 1986), LNCS 279, Springer-Verlag.

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022

14. H. Khoshnevisan, Automatic transformation systems based on function-level reasoning. PhD Thesis, Imperial College, University of London (1987).
15. G. Kahn, The semantics of a simple language for parallel programming. In *Information Processing 74*. North Holland (1974).
16. J. H. Williams, On the development of the algebra of functional programs. *ACM Transactions on Programming Languages and Systems* 4, 733–757 (1982).
17. J. H. Williams, E. Wimmers and A. Aitken, In *Proc. ACM Symposium on Principles of Programming Languages* (1988).

APPENDIX 1. DERIVATION OF THE CONCRETE FORM OF MATCH

Referring to section 4.2, we know that the two arguments supplied to match are ≪≫-trees with matching structure, the second of which is smaller than the first in the sense that its leaves match with subtrees of the second. This is because the first has one more application of $H_t id$ at its leaves. We can therefore write the definition of match as

$$\text{match} = \text{not_}\ll\gg\text{-tree} \circ 2 \rightarrow id;\ \text{match}^{*2}$$

where $\text{not_}\ll\gg\text{-tree} = \sim \circ \text{is_tree}$, ~ denoting the usual logical negation connective. The result of match has type $\tau(\tau \times \beta)$, i.e. is a ≪≫-tree with pairs at its leaves, the first component of which is a ≪≫-tree. Thus, the data type transformation for match has, in the notation of section 3,

$$\text{abs}_\alpha = \text{abs}^* \quad \text{and} \quad \text{abs}_\beta = \text{abs}^* \circ \text{abs}$$

Hence,

$$\text{match}' = \text{not_}\ll\gg\text{-tree} \circ 2 \circ \text{abs}^* \rightarrow \text{flatten} \circ \text{flatten}^* \circ \text{abs}^*;\ \text{flatten} \circ \text{flatten}^* \circ \text{match}^{*2} \circ \text{abs}^*$$

Now, not_≪≫-tree(abs x) is true if and only if x is a singleton ≪≫-tree. Hence, the else branch of match′ is always applied to a pair of non-singleton ≪≫-trees, so we can replace abs* by (abs* ∘ split)*. (It should be noted that this type of conditional analysis makes automation through term-rewriting difficult and here we have a typical situation in which user interaction through a transformational environment is desirable). By lemma 4.5 and basic FP laws we now have

$$\begin{aligned}\text{match}' &= \text{not_}\ll\gg\text{-tree} \circ \text{abs} \circ 2 \rightarrow \text{flatten} \circ \text{id}^*;\\ &\quad \text{flatten} \circ (\text{flatten} \circ \text{match} \circ \text{abs}^*)^{*2} \circ \text{split}^*\\ &= \text{is_sing} \circ 2 \rightarrow \{\,\|\,\};\ \text{flatten} \circ \text{match}'^{*2} \circ \text{split}^*\end{aligned}$$

Now match′ itself has been expressed as a DC function which, in the light of the above discussion, we can write as

$$\backslash((\text{deep flatten}) \circ 1) \circ [\{hd\}, tl] \circ (\text{upto}\ \psi p\ \text{split}^*)$$

after some simplification – we again have a 'merge-sort' class of function. Suppose now that the ≪≫-tree generated by application of $H_t id$ has arity $m > 1$. Then it follows that split divides a ≪≫-list into another ≪≫-list of ≪≫-lists of length m. (The number of components in every such ≪≫-list must be a power of m). Thus match′ may be defined by

$$\text{match}' = \text{is_sing} \circ 2 \rightarrow \{id\}; \{[\uparrow_m \circ 1, hd' \circ 2]\} \,\|^\wedge \text{match}' \circ [\downarrow_m \circ 1, \downarrow_1 \circ 2]$$

(We assume that m is not greater than the length of a list). In fact we transform match′ a little further by flattening the pairs it outputs into ≪≫-lists of length $m+1$. This gives:

$$\text{match}' = \text{is_sing} \circ 2 \rightarrow \|^\wedge;\ \{(\uparrow_m \circ 1) \,\|^\wedge (\uparrow_1 \circ 2)\} \,\|^\wedge \text{match}' \circ [\downarrow_m \circ 1, \downarrow_1 \circ 2]$$

APPENDIX 2. AN INDUCTIVE DEFINITION FOR E'_H

The following inductive definition is due to Eduardo Medina of Fujitsu-España and the University of Malaga. Let S' be defined by $S'(i,j)\,\{x_1, \ldots, x_n\} = \{x_i, \ldots, x_{i+j-1}\}$ $(1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant n-i+1)$. We then have the properties that, wherever S' is defined,

$$hd' \circ S'(i,j) = i'$$

$$S'(i_1,j_1) \circ \|^\wedge \circ [S'(i_2,j_2), k] = S'(i_1+i_2-1, j_1)$$
for all functions h and $i_1+j_1-1 \leqslant j_2$

Now let m, m_t be the ≪≫-tree arity of H_t, H_{tt} respectively (as defined in Appendix 1). Then, for each of the cases of Definition 4.1 we have respectively that $E'_H =$

(*a*) $1'$
(*b*) $E'_G \circ \|^\wedge \circ [S'(1,m), \{a \circ (m+1)'\}]$
(*c*) $a \circ E'_G$
(*d*) $[b_1, \ldots, b_n]$ where $b_i = a_i^* \circ S'(m+1, 1)$ if $g_i = a_i$ and $b_i = E'_{H_i}$ if $g_i = H_i f$
(*e*) $p \circ (m+1)' \rightarrow E'_A;\ E'_B$
(*f*)† $E'_P \rightarrow E'_A;\ E'_B$
(*g*) $[E'_{H_1} \circ [S'(1,m_1), S'(m+1,1)], \ldots, E'_{H_n} \circ [S'(1+\Sigma_{t=1}^{n-1} m_t, m_n), S'(m+1,1)]]$

† This case requires the additional assumption that A_t and B_t have the same ≪≫-tree arity. This is not unreasonable since A and B would normally be of the same type, being the branches of a conditional.

Downloaded from https://academic.oup.com/comjnl/article/35/6/555/352657 by U.S. Department of Justice user on 17 August 2022