Derivation of Efficient Data Parallel Programs*

Paul Roe
School of Computing Science,
Queensland University of Technology,
GPO Box 2434, Brisbane, QLD 4001, Australia
E-mail: proe@fit.qut.edu.au

Abstract

This paper considers the expression and derivation of efficient data parallel programs for SIMD and MIMD machines. It is shown that efficient parallel programs must utilise both sequential and parallel computation; these are termed *hybrid programs*.

The Bird–Meertens formalism, a calculus of higher order functions, is used to derive and express programs. Our goal is to derive efficient parallel programs for a variety of machines by: starting with an abstract specification, deriving an abstract algorithm and successively refining this to more efficient and machine dependent algorithms incorporating greater implementation detail.

Nested data structures are used to express hybrid algorithms. Using this technique efficient accumulate (scan/parallel prefix) algorithms are derived for SIMD and MIMD machines.

1 Introduction

The main reason for parallel programming is to achieve high performance. Unfortunately designing and writing efficient parallel programs, especially for MIMD machines, is difficult. Data parallel programming is widely recognised as the simplest paradigm for parallel programming. Nevertheless careful design of data parallel programs is necessary to ensure efficient use of machine resources. In particular naive use of data parallelism may not produce optimal programs.

There has been a wide variety of proposals for parallel programming languages and methodologies. Several researchers have proposed the use of higher order functions for simplifying parallel programming, and calculi of these functions for deriving parallel programs [4, 7, 13]. One such calculus is the Bird–Meertens formalism (BMF); this paper investigates its use for deriving efficient parallel programs, and in particular data parallel programs.

Our goal is to derive efficient parallel programs for a variety of machines by: starting with an abstract specification, deriving an abstract algorithm and successively refining this to more efficient and machine dependent algorithms incorporating greater implementation detail. This is similar in spirit to the Skeletons project at Imperial College [5].

In this paper it is shown that efficient parallel programs comprise both parallel and sequential computation. Such programs are termed hybrid programs. The main result of this paper is to show how hybrid programs may be expressed and derived using BMF.

The remainder of this paper is organised as follows. The next section presents an overview of BMF and discusses data parallelism in functional programs. Section 3 puts forward the idea of hybrid algorithms: efficient algorithms for SIMD and MIMD machines which comprise sequential and parallel computation. Sections 4 and 5 give example derivations of efficient hybrid algorithms for SIMD and MIMD machines. In both cases the example chosen is parallel accumulate (scan/parallel prefix). The last sections discuss future work and conclusions.

2 Bird–Meertens formalism

2.1 Basics

The Bird–Meertens formalism is a concise notation for expressing and deriving functional programs. It is, in some respects, similar to Backus's FP [1]; they are both algebraic approaches to program transformation. However, unlike FP, BMF is typed and it is in general more flexible than FP. Bird and

*to appear in: Proc. ACSC–17, Jan '93, Christchurch NZ

Meertens jointly developed this formalism and the following references are highly recommended: [3, 9]. Essentially we follow Bird's notation (except that we curry operators). For more background information on programming using functions see [2].

In BMF function application is expressed by juxtaposition for example $f\ a\ b$; function application associates to the left $f\ a\ b = (f\ a)\ b$. Functions may be composed thus: $f \cdot g$. It is assumed that function application is more binding than any other operator. All functions and operators are curried.

A useful shorthand, called sectioning, will be used for partially applying operators.

$$\begin{array}{ll} (\oplus)\ a\ b & = a \oplus b \\ (a \oplus)\ b & = a \oplus b \\ (\oplus\, b)\ a & = a \oplus b \end{array}$$

Operators such as $\oplus, \otimes, \odot, \ominus$ represent arbitrary binary operators. For example $(+1)$ is the increment function.

For the purposes of this paper only two data structures will be needed: lists (sequences) and pairs. First we describe lists and some list operations. Lists are homogeneous and may be appended using $+\!\!+$, for example: $[1,2] +\!\!+ [3,4] = [1,2,3,4]$. Two useful higher order operators on lists are: map $(*)$ and reduce $(/)$. Informally:

$$\begin{array}{ll} f*\,[a_1, a_2, \ldots, a_n] & = [f\ a_1, f\ a_2, \ldots, f\ a_n] \\ \oplus/[a_1, a_2, \ldots, a_n] & = a_1 \oplus a_2 \oplus \ldots \oplus a_n \end{array}$$

For example: $(+1) * [1,2,3,4,5] = [2,3,4,5,6]$ and $+/[1,2,3,4,5] = 15$. The operator used in a reduction must be associative. In the case that a reduction is applied to the empty list the result is the identity element of the operator. The identity element of an operator $\oplus$ will be denoted $1_\oplus$, for further information see [3].

As a further example consider the function defined below, which calculates the length of a list:

$$\begin{array}{ll} \mathit{length} & = +/ \cdot (K\ 1)* \\ K\ a\ b & = a \end{array}$$

If *length* is applied to an empty list the result will be the identity element of $+$ which is zero.

An important reduction which will be used frequently is concatenate: $+\!\!+/$. Concatenate flattens a list of lists into a single list.

A number of laws will be required for manipulating programs. The most important ones are promotion lemmas. Promotion lemmas for map and reduce are shown below:

$$\begin{array}{ll} f* \cdot +\!\!+/ & = +\!\!+/ \cdot f** \\ \otimes/ \cdot +\!\!+/ & = \otimes/ \cdot (\otimes/)* \end{array}$$

These promotion lemmas are so called because they allow an operation to be promoted into a data structure's components. We will use these lemmas for expressing and manipulating parallel programs in subsequent sections.

Another useful law is map distributivity:

$$f* \cdot g* = (f \cdot g)*$$

Reductions on lists may be directed: for example left reduce $\not\rightarrow$ (`foldl` in functional languages):

$$\ominus\not\rightarrow_e [a_1, a_2, \ldots, a_n] \quad = ((e \ominus a_1) \ominus a_2) \ominus \ldots \ominus a_n$$

Notice how the directed reduction takes an initial seed element (e).

A specialisation lemma allows any homomorphism, that is a function of the form $\otimes/ \cdot f*$, to be expressed as a left reduction:

$$\otimes/ \cdot f* \ = \ \ominus\not\rightarrow_e$$

where:

$$\begin{array}{ll} a \ominus b & = a \otimes (f\ b) \\ e & = 1_\otimes \end{array}$$

The last list operation we consider is accumulate. This is used as an example parallel algorithm throughout the rest of the paper. It may be informally defined thus:

$$\oplus\not\!\!\rightarrow [a_1, a_2, \ldots, a_n] \ = \\ [a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \cdots \oplus a_n]$$

For example: $+\not\!\!\rightarrow [1,2,3,4,5] \ = \ [1,3,6,10,15]$.

A right-wise accumulate may be defined in an analogous way, and similar results apply to it; however, this is pursued no further in this paper.

The second data structure which we will make use

of are pairs. The following operations are required:

$pair\ x\ y$	$=$	(x, y)	make pair
$dup\ x$	$=$	(x, x)	duplicate
$(f \parallel g)\ (x, y)$	$=$	$(f\ x, g\ y)$	pair map
$\overline{\ominus}\ (x, y)$	$=$	$x \ominus y$	uncurry/ reduce

In addition we informally define a partial operator zip-with ($\bigtriangledown$) thus:

$$\bigtriangledown_{\oplus}([x_1 \ldots x_n], [y_1 \ldots y_n]) \ = \ [x_1 \oplus y_1, \ldots, x_n \oplus y_n]$$

Zip may be defined as: $zip \ = \ \bigtriangledown_{pair}$

Other functions, laws and lemmas will be introduced as they are required.

2.2 Parallelism

Functional programs have a natural data parallel interpretation. In particular: maps, reduces and zip-withs all correspond to data parallel operations. However not all functions are suitable for parallel evaluation, for example directed reductions are inherently sequential.

Skillicorn has shown how several common functions used in BMF are universal over four classes of parallel architecture and hence that these support architecture independent parallel computation [13]. In this paper we show how efficient data parallel programs, targeted at particular architectures, may be expressed and manipulated using BMF. Our goal is to develop programs using BMF and then refine these as necessary for implementation on a particular architecture. In particular this involves refining programs beyond the abstract level which Skillicorn considered.

We shall use accumulate as an example parallel program; this has a variety of possible implementations. It may be defined as a composition of map and reduce (a homomorphism), providing its operator is associative:

$$\oplus /\!\!/\!\!\triangleright \ = \ \otimes / \cdot [\cdot]*$$

where:

$$a \otimes b \ = \ a +\!\!+ ((last\ a) \oplus) * b$$
$$last\ [a_1, a_2, \ldots, a_n] \ = \ a_n$$

The function $[\cdot]$ constructs a singleton list. Accumulate as defined above may be evaluated in parallel (CREW PRAM model) in time $O(ln\ n)$ using n processors. A total of $n\ ln\ n$ operations are performed. There are many other possible definitions of accumulate. We derive another below:

$$\otimes / \cdot [\cdot]*$$

$= \{$left reduction specialisation, assuming $1_{\otimes} = []\}$

$$\ominus /\!\!\triangleright_{[]}$$

where:

$$xs \ominus x \ = \ xs \otimes [x]$$

$= \{$using $\otimes$ def.$\}$

$$xs +\!\!+ ((last\ xs) \oplus) * [x]$$

$= \{$using map def.$\}$

$$xs +\!\!+ [last\ xs \oplus x]$$

Thus, accumulate may be expressed as a left reduction:

$$\begin{array}{lll} \oplus /\!\!/\!\!\triangleright & = & \ominus /\!\!\triangleright_{[]} \\ & & \text{where} \\ & & xs \ominus x = xs +\!\!+ [last\ xs \oplus x] \end{array}$$

This is an optimal sequential algorithm which performs n operations in time n. In addition it does not require $\oplus$ to be associative. Notice that the parallel algorithm performs more operations than the sequential algorithm. This is discussed in the next section. Throughout the rest of this paper only accumulates suitable for parallel evaluation, that is those with associative operations, will be considered.

3 Hybrid algorithms

The previous section gave an example of a parallel algorithm which is not an efficient sequential algorithm (parallel accumulate). There are many such algorithms. This means that in general the most efficient way to organise a parallel algorithm, for a particular machine, is to utilise both sequential and parallel computation for the sequential and parallel parts of the machine. We call such programs, *hybrid* programs.

In order to derive programs for efficient implementation on a particular architecture, it is necessary to express and manipulate hybrid programs using BMF. Two general hardware architectures are considered: SIMD and MIMD.

3.1 SIMD hybrid algorithms

We consider SIMD machines whose PEs (processing elements) have limited memory and processing power, such as array processors and systolic machines. SIMD machines which have more powerful PEs are better treated as MIMD machines.

SIMD machines provide data parallel operations as primitives. For example a vector processor may provide vector (sequence) addition (∇_+); the exact operations are machine dependent. These machines compute by performing a sequence of data parallel operations. Thus SIMD machines may be efficiently programmed in sequential languages augmented with parallel bulk data operations (eg. Fortran 90). Effectively, SIMD machines consist of two computational layers:

Sequential layer
Parallel data operation layer

There are two issues associated with programming SIMD machines: the sequencing of parallel data operations and the splitting up of data to fit onto the machine. In general data will not exactly fit onto a SIMD machine; in which case it is necessary to divide the data and process it in parts. We wish to incorporate both of these issues into our programs.

There are virtual processor proposals which seek to treat a processor array in an analogous way to virtual memory. However this precludes some important efficient algorithms. This also applies to MIMD architectures.

It is assumed that a repertoire of predefined data parallel operations is available. For example, this might include a parallel map operation restricted to a particular set of functions. In addition a function is required for dividing data, in this cases lists, into machine-sized chunks: div_p splits a list into sub-lists of length p. The only property required of div is:

$$++/ \cdot div_p \;=\; id$$

where id is the identity function on lists.

A SIMD map algorithm may be derived from an abstract map algorithm in two simple steps:

$$f*$$

$$= \{div \text{ property}\}$$

$$f* \cdot ++/ \cdot div_p$$

$$= \{\text{map promotion}\}$$

$$++/ \cdot (f*)* \cdot div_p$$

It may seem as though all that has have achieved is to redefine map using map. However the two maps in the algorithm are meant to express different operations. The algorithm expresses that the list is divided into machine-sized sub-lists. The outer map expresses that sub-lists are processed in sequence. The inner map (a predefined parallel map) expresses that each sub-list is mapped over in parallel. It would be possible to differentiate by name or type the different maps/lists, but this is pursued no further.

This is a somewhat artificial example; in particular no optimisation of the hybrid algorithm is possible. In Section 4 a more complex example will be considered where the resulting hybrid algorithm can be optimised. However this example does demonstrate the importance of promotion in the derivation of hybrid algorithms.

3.2 MIMD hybrid algorithms

An analogous situation to expressing efficient parallel programs for SIMD machines arises with MIMD machines. We assume that a MIMD machine consists of a network of processing elements. Such a machine consists of two computational layers:

Parallel network (inter-processor) layer
Sequential (intra-processor) layer

Notice that compared with SIMD machines, the computational layers of MIMD machines are inverted.

Efficient MIMD algorithms should utilise efficient sequential and parallel computation for these two layers. This will be made explicit in BMF by expressing data parallel computation over nested data structures: representing the distribution of data

across a machine. For example, for a computation on a list, the list will be split into equal length sub-lists and distributed across a machine's PEs. All the sub-lists will be processed in parallel, with each individual sub-list being processed sequentially; subsequently sub-lists will be combined.

To express such algorithms we require a function, similar to *div*, which splits a list into sub-lists: $split_p$ splits a list into p sub-lists. Like *div* the only property required of *split* is:

$$+\!\!+/ \cdot split_p \;=\; id$$

For example if we have an abstract parallel map algorithm, it can be refined to a MIMD algorithm thus:

$$f*$$

$= \{split \text{ property}\}$

$$f* \cdot +\!\!+/ \cdot split_p$$

$= \{\text{map promotion}\}$

$$+\!\!+/ \cdot (f*)* \cdot split_p$$

This algorithm expresses that a list is split into sub-lists and distributed across a machine. Each sub-list resides on a separate PE. Sequential maps (inner map) are applied in parallel (outer map) across the sub-lists. Thus the operation of the algorithm is the converse of the SIMD one.

As with the SIMD map, this is an artificial example because no real computation is necessary at the parallel layer: apart from possible data distribution. Also, no optimisation of the final program is possible. However, in Section 5 a more complex example is considered where the resulting hybrid algorithm can be optimised, and where different algorithms are used for the two computational layers.

4 Hybrid SIMD accumulate

In this section we consider a larger example of deriving a hybrid algorithm for a SIMD machine: accumulate.

$$\oplus /\!\!\!/$$

$= \{div \text{ property}\}$

$$\oplus /\!\!\!/ \cdot +\!\!+/ \cdot div_p$$

(It is assumed that $\oplus$ is associative.) To proceed further a promotion lemma for accumulate is required. This will allow the promotion of accumulate into the sub-lists generated by *div*.

Lemma: accumulate promotion (I)

Accumulate may be promoted thus:

$$\oplus /\!\!\!/ \cdot +\!\!+/$$

$= \{\text{previous accumulate def.}\}$

$$\begin{array}{l} \odot/ \cdot [\cdot]* \cdot +\!\!+/ \\ \text{where } a \odot b = a +\!\!+ ((last\ a)\oplus) * b \end{array}$$

$= \{\text{map and reduce promotion}\}$

$$\odot/ \cdot (\odot/ \cdot [\cdot])*$$

$= \{\text{accumulate def.}\}$

$$\odot/ \cdot (\oplus /\!\!\!/)*$$

□

Returning to the derivation of the hybrid accumulate algorithm:

$$\oplus /\!\!\!/ \cdot +\!\!+/ \cdot div_p$$

$= \{\text{accumulate promotion}\}$

$$\begin{array}{l} \odot/ \cdot (\oplus /\!\!\!/)* \cdot div_p \\ \text{where } a \odot b = a +\!\!+ ((last\ a)\oplus) * b \end{array}$$

$= \{\text{using specialisation}\}$

$$\begin{array}{lll} \ominus /\!\!\!/_e \cdot div_p & & \\ \text{where} & a \ominus b & = a \odot (\oplus /\!\!\!/\ b) \\ \text{and} & e & = 1_\odot = [\,] \end{array}$$

$= \{\odot \text{ def.}\}$

$$a \ominus b = a +\!\!+ ((last\ a)\oplus) * (\oplus /\!\!\!/\ b)$$

It is possible to optimise the $\ominus$ operation further yielding:

$$as \ominus ([b] +\!\!+ bs) \;=\; as +\!\!+ \oplus /\!\!\!/ ([(last\ as) \oplus b] +\!\!+ bs)$$

However this probably is unnecessary since the parallel cost of performing a single $\ominus$ operation is likely

to be the same as mapping $\ominus$ over a machine-sized chunk of data.

Thus, $\oplus /\!\!/\!\!\rangle$ may be refined to the following SIMD algorithm:

$$\ominus /\!\!\!\rangle_{[\,]} \cdot div_p$$

where:

$$a \ominus b = a +\!\!+ ((last\ a)\oplus) * (\oplus /\!\!/\!\!\rangle\ b)$$

The left reduction expresses the sequencing of data parallel operations over the input, which is divided into machine-sized sub-lists. The exact implementation of $/\!\!/\!\!\rangle$ has been left unspecified; this may be a primitive or it may be programmed. In the latter case the definition previously given for accumulate may be used or the more efficient algorithm of Ladner and Fischer may be employed [8]. (Gibbons has investigated the derivation of algorithms similar to the latter in [6].)

Assuming that parallel accumulate takes logarithmic time, this algorithm runs in time $\lceil n/p \rceil \times (ln\ p + 1)$ where the input size is n and the size of the SIMD machine is p. This is a reasonably efficient algorithm. Theoretically, there are quicker algorithms, for example [11], but the practicality of these complex algorithms is unknown.

5 Hybrid MIMD accumulate

This section describes the derivation of an efficient MIMD accumulate. The derivation starts in a similar way to the SIMD one:

$$\oplus /\!\!/\!\!\rangle$$

$= \{split \text{ property}\}$

$$\oplus /\!\!/\!\!\rangle \cdot +\!\!+/ \cdot split_p$$

A modified form of accumulate promotion is required.

Lemma: accumulate promotion (II)

It has been proven that:

$$\oplus /\!\!/\!\!\rangle \cdot +\!\!+/ \ = \ \odot/ \cdot \ (\oplus /\!\!/\!\!\rangle)*$$
$$\text{where } a \odot b = a +\!\!+ ((last\ a)\oplus) * b$$

A simple proof by induction reveals that:

$$\odot/ = +\!\!+/ \cdot \ominus /\!\!/\!\!\rangle$$
$$\text{where } a \odot b \ = \ a +\!\!+ (a \ominus b)$$
$$a \ominus b \ = \ ((last\ a)\oplus) * b$$

Therefore accumulate promotion may be expressed thus:

$$\oplus /\!\!/\!\!\rangle \cdot +\!\!+/ \ = \ +\!\!+/ \cdot \ominus /\!\!/\!\!\rangle \cdot (\oplus /\!\!/\!\!\rangle)*$$
$$\text{where } a \ominus b = ((last\ a)\oplus) * b$$

□

Returning to the derivation, accumulate can be promoted, giving:

$$+\!\!+/ \cdot \ominus /\!\!/\!\!\rangle \cdot (\oplus /\!\!/\!\!\rangle)* \cdot split_p$$
$$\text{where } a \ominus b = ((last\ a)\oplus) * b$$

We would like to make use of the following property to optimise $\ominus /\!\!/\!\!\rangle$:

$$a \ominus b \ominus c \ = \ (last\ a \oplus last\ b) \otimes c$$

where:

$$x \otimes y \ = \ (x\oplus) * y$$

This may be expressed more generally as:

$$\ominus/ \ = \ \overline{\otimes} \cdot (\oplus/ \cdot last * \cdot init \parallel last) \cdot dup$$

The function *init* is defined below:

$$init\ [a_1, a_2, \ldots, a_n] \ = \ [a_1, a_2, \ldots, a_{n-1}]$$

Note that $(\oplus/[\,]) \otimes x = x$, since the identity element of $\oplus$ will also be the left identity element of $\otimes$ (see [3] for further details on the role of identity elements).

In order to utilise this optimisation the following lemma, due to Bird [3], is required:

Accumulation lemma

We informally define $inits^+$ to be:

$$inits^+\ [a_1, a_2, \ldots, a_n] \ = \ [[a_1], [a_1, a_2], \ldots, [a_1, \ldots, a_n]]$$

and *inits* to be:

$$inits \ = \ ([[\,]] +\!\!+) \cdot inits^+$$

Then accumulate, with an associative operator $\oplus$, may be expressed thus:

$$\otimes \not\!\!\!\!\!\!\rightarrow \;=\; (\otimes/)* \cdot inits^{+}$$

$\square$

Returning to the derivation we may optimise $\ominus \not\!\!\!\!\!\!\rightarrow$ thus:

$$\ominus \not\!\!\!\!\!\!\rightarrow$$

$= \{$accumulation lemma$\}$

$$(\ominus/) * \cdot inits^{+}$$

$= \{\ominus/$ property$\}$

$$(\overline{\otimes} \cdot (\oplus/ \cdot last * \cdot init \parallel last) \cdot dup) * \cdot inits^{+}$$

$= \{dup * \;=\; zip \cdot dup\}$

$$(\overline{\otimes} \cdot (\oplus/ \cdot last * \cdot init \parallel last)) * \cdot zip \cdot dup \cdot inits^{+}$$

$= \{\overline{\ominus} * \cdot (f \parallel g) * \cdot zip \;=\; \nabla_{\ominus} \cdot (f* \parallel g*)\}$

$$\nabla_{\otimes} \cdot ((\oplus/ \cdot last * \cdot init)* \parallel last*) \cdot dup \cdot inits^{+}$$

$= \{dup \cdot f \;=\; (f \parallel f) \cdot dup\}$

$$\nabla_{\otimes} \cdot ((\oplus/ \cdot last * \cdot init) * \cdot inits^{+} \parallel last * \cdot inits^{+}) \cdot dup$$

$= \{last* \cdot inits^{+} \;=\; id\}$

$$\nabla_{\otimes} \cdot ((\oplus/ \cdot last * \cdot init) * \cdot inits^{+} \parallel id) \cdot dup$$

$= \{$map distributivity$\}$

$$\nabla_{\otimes} \cdot ((\oplus/) * \cdot (last * \cdot init) * \cdot inits^{+} \parallel id) \cdot dup$$

$= \{f* \cdot init \;=\; init \cdot f*\}$

$$\nabla_{\otimes} \cdot ((\oplus/) * \cdot init * \cdot last ** \cdot inits^{+} \parallel id) \cdot dup$$

$= \{f** \cdot inits^{+} \;=\; inits^{+} \cdot f*\}$

$$\nabla_{\otimes} \cdot ((\oplus/) * \cdot init * \cdot inits^{+} \cdot last* \parallel id) \cdot dup$$

$= \{init* \cdot inits^{+} \;=\; inits \cdot init\}$

$$\nabla_{\otimes} \cdot ((\oplus/) * \cdot inits \cdot init \cdot last* \parallel id) \cdot dup$$

$= \{inits$ def$\}$

$$\nabla_{\otimes} \cdot ((\oplus/)* \cdot ([[\,]] +\!\!+) \cdot inits^{+} \cdot init \cdot last* \parallel id) \cdot dup$$

$= \{$map promotion and def$\}$

$$\nabla_{\otimes} \cdot (([\oplus/[\,]] +\!\!+) \cdot (\oplus/) * \cdot inits^{+} \cdot init \cdot last* \parallel id) \cdot dup$$

$= \{$accumulation lemma$\}$

$$\nabla_{\otimes} \cdot (([\oplus/[\,]] +\!\!+) \cdot \oplus \not\!\!\!\!\!\!\rightarrow \cdot init \cdot last* \parallel id) \cdot dup$$

$= \{$since $(\oplus/[\,]) \otimes y = y$ and ∇ def. plus new fun$\}$

$$g \cdot (\oplus \not\!\!\!\!\!\!\rightarrow \cdot init \cdot last* \parallel id) \cdot dup$$

where:

$$g\ (xs, [y] +\!\!+ ys) = [y] +\!\!+ \nabla_{\otimes}(xs, ys)$$

Thus accumulate $(\oplus \not\!\!\!\!\!\!\rightarrow)$ may be refined to the efficient parallel MIMD accumulate algorithm given below:

$$+\!\!+/ \cdot g \cdot (\oplus \not\!\!\!\!\!\!\rightarrow \cdot init \cdot last* \parallel id) \cdot dup \cdot (\oplus \not\!\!\!\!\!\!\rightarrow)* \cdot split_p$$

where:

$$\begin{array}{ll} g\ (xs, [y] +\!\!+ ys) & = [y] +\!\!+ \nabla_{\otimes}(xs, ys) \\ x \otimes y & = (x\oplus) * y \end{array}$$

In effect this hybrid accumulate algorithm is the fusion of a parallel and a sequential accumulate algorithm.

This algorithm takes time: $2\lceil n/p \rceil + ln\ p$, on a p processor machine with an input of size n. This may be compared with the time: $\lceil n/p \rceil \times (1 + ln\ p)$ for the unoptimised algorithm shown below:

$$+\!\!+/ \cdot \ominus \not\!\!\!\!\!\!\rightarrow \cdot (\oplus \not\!\!\!\!\!\!\rightarrow)* \cdot split_p$$

where:

$$a \ominus b = ((last\ a)\oplus) * b$$

The time difference between the optimised and unoptimised algorithms may seem negligible. However on a real machine a constant factor of say $(ln\ p)/2$ is very significant.

6 Future work

In general there are more difficulties involved in programming MIMD machines than SIMD ones; hence future investigation will concentrate on these architectures.

Data distribution is an unsolved problem in programming MIMD machines. One approach is to leave this to a machine's run-time system to perform dynamically, using heuristics. An alternative approach is to make this explicit in programs. The method for expressing hybrid MIMD programs uses the *split* function. One, of the several, operational interpretations of *split* is: to dynamically distribute data at run-time. Under this interpretation, reductions at the parallel level (layer) correspond to data aggregations.

Furthermore functions such as list-rotate, at the parallel level, correspond to communication operations. Mou has investigated this within the context of divide and conquer algorithms [10]. Mou's work is related to this work in that he takes a similar

algebraic approach to expressing parallel programs. However his work is concentrated exclusively on a divide and conquer paradigm of programming and is concerned with expressing rather than deriving parallel programs. Data distribution and communication deserve further investigation.

In this paper lists have been used to model computer architecture: to support the expression and derivation of hybrid algorithms. However different data structures may be more appropriate depending on the target architecture. For example a two dimensional mesh architecture may be more accurately modelled by an array data structure. Refining the machine model (data structure) will also allow communications operations to be exposed and refined.

Another area for future research is the performance analysis of parallel functional programs. Using a restricted set of higher order functions should simplify reasoning about program performance.

Lastly some network style parallel algorithms are restricted to data whose size is a power of two. Incorporating this kind of restriction (type restriction) into programs is an area of research. Sheeran's excellent work on Ruby (a relational language similar to BMF) looks promising in this area [12].

7 Conclusions

It has been shown that writing efficient algorithms for SIMD and MIMD machines entails writing hybrid algorithms which utilise parallel and sequential computation. Hybrid algorithms may be represented in BMF using nested data structures. BMF allows these hybrid algorithms to be manipulated in a straightforward manner. This represents a big step towards our ultimate goal of deriving efficient parallel programs by successively refining programs to incorporate more architecture specific details.

Acknowledgements

I would like to thank the anonymous referees for their helpful comments.

References

[1] J Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, 1978.

[2] R Bird and P Wadler. *An Introduction to Functional Programming*. Prentice Hall International, 1988.

[3] R S Bird. Lecture notes on constructive functional programming. Technical monograph 69, PRG, Oxford University, September 1988.

[4] M Cole. Higher order functions for parallel evaluation. In C Hall, J Hughes, and J O'Donnell, editors, *Proceedings of the 1988 Glasgow Workshop on Functional Programming, August 2-5, 1988, Rothesay, Isle of Bute, Scotland*. Research report 89/R4, Computing Science Dept. University of Glasgow, February 1989.

[5] J Darlington, A J Field, P G Harrison, P H J Kelly, D W N Sharp, Q Wu, and R L While. Parallel programming using skeleton functions. In *PARLE 93*. LNCS, Springer Verlag, 1993.

[6] J Gibbons. Computing downwards accumulations on trees quickly. In G Gupta, G Mohay, and R Topor, editors, *Proceedings of 16th Australian Computer Science Conference*, pages 685–691. Australian Computer Science Communications, volume 15, number 1, February 1993.

[7] P G Harrison. A higher-order approach to parallel algorithms. *The Computer Journal*, 35(6):555–566, 1992.

[8] R E Ladner and M J Fischer. Parallel prefix computation. *Journal of ACM*, 27(4):831–838, October 1980.

[9] L Meertens. Algorithmics – towards programming as a mathematical activity. In J W deBakker, M Hazewinkel, and L K Lenstra, editors, *CWI Symposium on Mathematics and Computer Science, Vol.1*, pages 289–334. CWI monographs, North-Holland, 1986.

[10] Z G Mou. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Proceedings of 3rd Symposium on Frontiers of Massively Parallel Computation*, pages 451–461. IEEE, October 1990.

[11] A Nicolau and H Wang. Optimal schedules for parallel prefix computation with bounded resources. In *3rd ACM Symposium on Parallel Programming, Williamsburg, Virginia*. ACM Press, April 1991.

[12] M Sheeran. Describing hardware algorithms in Ruby. In K Davis and J Hughes, editors, *Functional Programming: Proceedings of the 1989 Glasgow Workshop, 21-23 August 1989, Fraserburgh, Scotland*, Springer Workshops in Computing. Springer Verlag, 1990.

[13] D B Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.