List Homomorphic Parallel Algorithms for Bracket Matching

Murray Cole *

Abstract

We present a family of parallel algorithms for simple language recognition problems involving bracket matching. The algorithms are expressed in the Bird-Meertens Formalism, exploiting only list operations which are inherently massively parallel. Our intention is to illustrate the practical efficacy with which such algorithms can be derived and expressed given the support of a well understood theoretical foundation. One of the variants produced is of particular interest in that it exploits the same theoretical result twice to produce nested parallelism.

1 Introduction

In [8], we investigated an informal methodology for the generation of parallel algorithms based upon exploitation of a fundamental result from the Bird-Meertens "theory of lists". Our main example was an algorithm for the maximum segment sum problem. In this report we provide further examples of the approach. For completeness, the remainder of this section and sections 2 and 3 repeat the introductory material from [8]. Readers who are already familiar with its content may safely proceed to section 4.

The use of bulk operations on aggregate data sets as a means of generating programs with a high degree of implicit parallelism has a long history (e.g. see [5] for a recent presentation). Although traditionally associated with an imperative programming style and SIMD machines, the approach lends itself equally well to a purely functional presentation and seems amenable to implementation on coarser grained message-passing MIMD architectures [7].

The introduction of a pure functional style brings with it well documented opportunities for program design by semantics preserving transformation (perhaps most famously expounded in [1]). In this approach, one begins by writing a simple, "obviously" correct, but possibly inefficient solution. With a little ingenuity, and a toolkit of transformation rules, this can be transformed into a semantically equivalent, but more efficient solution to the same problem. The method is applicable

*E-mail address : mic@dcs.ed.ac.uk

whether the eventual target machine is sequential or parallel, given an appropriate cost model within which to work.

This approach has been used to derive parallel algorithms for a variety of problems and target architectures [6, 9, 11, 12, 13, 14, 15, 18]. In the remainder of this paper, we concentrate on a particular instantiation of the method, and the application of one of its simplest results.

2 The Bird-Meertens Formalism and its Parallel Interpretation

The Bird-Meertens Formalism (BMF or "theory of lists") is a small collection of (mainly) second-order functions on lists, a collection of algebraic identities and theorems relating these and a concise notation which facilitates the transformational approach to programming discussed above. Introductions to BMF can found in [3, 4]. In this paper we require only the two simplest and most familiar of these operations, *map* and *reduce*.

Map is the curried function which applies another function to every item in a list. In BMF it is written as an infix $*$. Thus, informally, we have

$$f* [x_1, x_2, ..., x_n] = [f x_1, f x_2, ..., f x_n]$$

Reduce is the curried function which collapses a list into a single value by repeated application of some binary operator. The full theory distinguishes between directed reductions, in which the list must be traversed from one end or the other, and undirected reductions, in which the result can be accumulated in either order (or in parallel) subject only to the requirement that the operator be associative. We will deal only with the undirected case, and will assume that the operator has an identity element. In BMF, reduce is written as an infix /, and so informally, for a suitable operator $\oplus$ with identity e, we have

$$\oplus/ [x_1, x_2, ..., x_n] = [e \oplus x_1 \oplus x_2 \oplus ... \oplus x_n]$$

Even with just these two simple operators, we begin to taste the flavour of the BMF approach. For example, using $+$ (with identity 0) and $+\!\!\!+$ (with identity $[\,]$) to represent addition and list concatenation respectively, we can express functions to sum a list and to flatten a list of lists into a single list as

$$\begin{array}{lcl} sum & = & +/ \\ flatten & = & +\!\!\!+/ \end{array}$$

and using $\cdot$ to represent functional composition and $\wedge$ to represent boolean conjunction, we can express the function which determines whether all members of a list satisfy some predicate p as

$$all\ p = (\wedge/) \cdot (p*)$$

It is not difficult to see that $*$ (trivially) and $/$ (most obviously as a binary tree) have simple massively parallel implementations on many architectures. In a series of papers, Skillicorn investigates the definition of a suitable parallel cost model for BMF and illustrates its efficacy with the transformational derivation of various parallel programs and the justification of the introduction of new operators to the theory [6, 16, 18, 17, 19].

3 The Homomorphism Lemma

As we have noted above, the bulk of work in BMF (in both sequential and parallel contexts) follows the simple specification then formal transformation approach. In what follows, we attempt to show that an understanding of the formalism and its theorems can be equally helpful in providing a clear conceptual framework within which to undertake a more directly intuitive approach to algorithm design. To exemplify this approach we consider a single, simple result, the so-called "Homomorphism Lemma" [3].

A function h from finite lists to some other type T is a list homomorphism if there exists an associative operator $\oplus$ (of type $T \times T \rightarrow T$) with identity element e, such that

$$\begin{array}{lcl} h\,[\,] & = & e, \\ h\,(x +\!\!\!+\, y) & = & h\,x \,\oplus\, h\,y \end{array}$$

for all lists x and y. For example, using $\uparrow$ to denote the operator which returns the larger of its two integer arguments, and $-\infty$ to denote an artificial identity for $\uparrow$, we can see that the function which takes an integer list and returns its largest member is such a homomorphism, with $\uparrow$ playing the role of $\oplus$ and $-\infty$ of e.

Similarly, and perhaps more interestingly, the function which sorts integer lists is also a homomorphism, with $\oplus$ corresponding to the operator which merges two already sorted lists with $[\,]$ as its identity (in other words, mergesort). In general, we may note the close correspondence with divide and conquer algorithms, with the extra condition that we are not free to specify the divide operation beyond the fact that it splits the list into two arbitrary segments.

The Homomorphism Lemma provides another characterization of list homomorphisms which relates them directly to a potentially highly parallel program schema.

Lemma 1 ***The Homomorphism Lemma*** *[3].*
A function h is a homomorphism with respect to $+\!\!\!+$ if and only if $h = (\oplus/) \cdot (f*)$ for some operator $\oplus$ and function f.

The implications for parallel programming are clear - if a problem is a list homomorphism, then it only remains to define $\oplus$ and f in order to produce a highly parallel solution. The performance of this program will be governed by the complexities of $\oplus$ and f, the efficiency with which the target architecture can support / ($*$ is easy) and, if message passing is involved, the way in which the size of the emerging result of the / grows as it is computed (e.g. compare the two homomorphisms discussed above).

4 Application to "Near" Homomorphisms

As in [8], we now investigate the proposition that an understanding of the homomorphism lemma can be a useful tool in the search for parallel algorithms for problems which are not homomorphisms. The essence of the approach is to consider the problem in the homomorphic style, then to find the extra computational "baggage" which is required to turn it into a genuine homomorphism. What we end up with is a hybrid of the original problem, whose solution contains the solution of the original as a component, and whose corresponding BMF expression includes a precise specification of the baggage. By definition, this gives us a highly parallel solution to the original problem. The extent to which the solution is efficient depends upon the usual factors discussed above, together with the cost of computing and communicating the baggage, which may depend upon the parallel machine model in use. Thus, we solve problems which are in some very vague sense "near" homomorphisms, by embedding them within true homomorphisms and then applying the standard lemma.

The algorithms discussed in this paper provide further evidence of the practicality and simplicity of such an approach. Additionally, one of the variants is of interest in that it exhibits two levels of homomorphism based parallelism. The presentation in the main body of the paper introduces several auxiliary functions informally, in an attempt to avoid clutter and distraction from the main flow. Full Standard ML definitions of all functions used are provided in the appendix. Of course, it should be remembered that these include some sequential features which would be superseded by parallel implementations of the BMF style primitives they model.

5 Bracket Matching Problems

We consider the problem of determining whether the brackets in a given string are correctly matched. We consider both the simple case in which there is only one type of bracket (sometimes called "bracket languages") and the more complicated case which allows an arbitrary number of types (sometimes called "input-driven languages"), which must match both independently and collectively. These are language recognition problems. For example, for the language generated by the grammar

$$\begin{array}{lcl} S & \rightarrow & S\;S \\ S & \rightarrow & (\;S\;) \\ S & \rightarrow & [\;S\;] \\ S & \rightarrow & \{\;S\;\} \\ S & \rightarrow & \textit{any sequence of other symbols} \end{array}$$

the string "[abc{}de(x{}yz)]" is accepted, whereas 'ab{[c[d]}]" is not.

It is known [10] that both bracket and input-driven languages can be recognized in $O(\log n)$ time on $O\left(\frac{n}{\log n}\right)$ processors in the PRAM model. However, the algorithms involved are non-trivial. Our intention here is to illustrate that good, if sub-optimal parallel algorithms can be derived in a simple, intuitive fashion by exploiting the Homomorphism Lemma. In a similar vein, [2] presents a BMF parsing algorithm for operator precedence grammars, from which a solution to the bracket matching problems can be specialised. Their algorithm has linear time behaviour in the worst case.

5.1 The Single Bracket Type Problem

In this instance we allow only one type of bracket (for example '(' and ')'). The problem has a well known linear time sequential algorithm, in which the string is examined from left to right. A counter is initialised to 0, and incremented or decremented as opening and closing brackets are encountered. In a correctly matched string the counter will never fall below zero, and will finish back at zero. In BMF, the algorithm can be expressed as a simple directed reduction.

The move to a homomorphic super-problem is not difficult. If two components of a concatenation are independently correctly matched, then so is the concatenation. Otherwise, we need to decide whether information collected from the independent sub-checks implies that a complete check would have been successful, in other words, that its counter would never become negative, and would return to zero. The second requirement can be tested by summing the counters from the two component checks. The first requires that the counter from the left component never became negative, and that the counter from the right component would not have become negative *had it been initialised to the final value of the left counter* rather than zero. Thus, our baggage must include both the final counter value and the minimum value reached. The homomorphism we require is therefore from lists of characters (i.e. strings) to triples containing an indication of whether the string was correctly matched and final and minimum values which would be reached by the counter during a sequential examination. To generate the low counter value for a concatenation we simply take the minimum of the low counter for its left component and the sum of the left component's final counter and the right component's low counter. In terms of Lemma 1, the $\oplus$ operator is defined by the equation

$$
\begin{array}{ll}
& (ok, final, low) \oplus (ok', final', low') \\
= & ((ok \wedge ok') \vee ((final + final' = 0) \wedge (low >= 0) \wedge (final + low' >= 0), \\
& \quad final + final', \;\; low \downarrow (final + low'))
\end{array}
$$

(where $\downarrow$ returns the smaller of its arguments) and the function f which, when mapped, describes the operation of the homomorphism on singleton lists is

$$
\begin{array}{lll}
f \;\; '(' & = & (false,\ 1,\ 1) \\
f \;\; ')' & = & (false,\ -1,\ -1) \\
f \;\; c & = & (true,\ 0,\ 0)
\end{array}
$$

Lemma 1 now presents us with a highly parallel algorithm for the homomorphism *ematch* (for extended match) from which we can trivially pluck the result we require.

$$
\begin{array}{lll}
exmatch & = & (\oplus/) \cdot (f*) \\
match & = & fst \cdot exmatch \\
& & where\ fst\ (a,\ -,\ -) = a
\end{array}
$$

Since our baggage consists of two integers and a constant number of simple integer and logical operations, we can expect an $O(\log n)$ algorithm on $O\left(\frac{n}{\log n}\right)$ processors on many architectures.

5.2 The Many Bracket Type Problem

The introduction of multiple bracket types complicates the problem a little, but a straightforward linear time sequential algorithm still exists. Rather than a simple count, a stack is now maintained during the scan of the input. Opening brackets are pushed, and closing brackets are matched with the current stack top. Failure is indicated by a mismatch, by an empty stack when a match is required, or by a non-empty stack at the end of the scan. Obviously the use of a stack appears unpromising from a parallel point of view.

We now apply our homomorphism based approach to the problem. As with the single bracket case, successful independent matches in both left and right components imply a successful match for their concatenation. Otherwise, we need to investigate whether the reasons for failure in the two components can compensate to produce an overall match. The key observation is that in doing so, we can safely ignore those segments of the components which were independently satisfactorily matched. Thus, the homomorphism we seek is really from general strings to strings with all matched segments removed. If such a string is empty, then the original string was correctly bracketed. Otherwise, for example, given "ab{[c[d]}]" we will return "{[}]".

What is required of the $\oplus$ in our homomorphism? Given the "leftovers" from two strings, it must return the leftovers of their concatenation. But if matching segments have already been removed from the components individually, any matched segment remaining in the concatenation must come from the area around the join. It can be removed by pairing off characters from the *reverse* of the left leftovers and the right leftovers, while these form matching pairs. The characters in the concatenation of what remains are exactly the leftovers from the concatenation of the original pair of strings. Thus, with the obvious f for singletons, we have

$$
\begin{array}{lcl}
f\ \ c & = & if\ isbracket\ \ c\ \ then\ [c]\ else\ [\,] \\
\\
xs \oplus ys & = & combine \cdot dropmatches\ (rev\ xs,\ ys) \\
\\
dropmatches\ ([\,], ys) & = & ([\,], ys) \\
dropmatches\ (xs, [\,]) & = & (xs, [\,]) \\
dropmatches\ (x : xs, y : ys) & = & if\ (match\ x\ y)\ then\ dropmatches\ (xs,\ ys) \\
 & & else\ (x : xs,\ y : ys) \\
\\
combine\ (xs,\ ys) & = & (rev\ xs) + \!\! + \ ys \\
\\
exmatch & = & (\oplus/) \cdot (f*) \\
\\
match & = & isempty \cdot exmatch
\end{array}
$$

Unfortunately, we now have both communications and computational baggage (in *dropmatches*) which is linear in the size of the leftover strings. In the best case (for strings with only short bracketed segments with shallow nesting) we will have our usual $O(\log n)$ performance on $O\left(\frac{n}{\log\ n}\right)$ processors. However, in the worst case (with one deeply nested segment) we will do no better than the linear time of the sequential algorithm. In the remainder of the paper, we investigate two variants of the algorithm which uncover scope for parallelism within the execution of the $\oplus$ operation itself.

6 Algorithms with Nested Parallelism

In this section we present two variants of the multi bracket type algorithm described above. In the first, we exploit our standard approach to introduce a layer of homomorphism based parallelism within the execution of the $\oplus$ operation. This has the effect of reducing the computational baggage to $O(\log n)$ but still involves linear communications baggage at the inner level of parallelism. In the second, less straightforward algorithm we manage to remove this final bottleneck.

6.1 A Homomorphic Implementation of $\oplus$

The sequential bottleneck in the $\oplus$ in the program above occurs in *dropmatches* (the remaining list reversals and type coercions have simple parallel realisations).

Can we express this as a homomorphism? To do so, we will first need to massage the data provide into a list of some kind. The obvious choice is to zip together the reverse of xs and ys, since this brings together the pairs of characters which must match. However, we cannot be sure that these are of the same length. To cater for this, we introduce a new datatype, with constructors One and $None$, and a zip which produces a list of pairs of items each labelled with One and uses $None$ to fill out pairs when the shorter list has run out. We use this to build a function $mould$, which takes a pair of lists, reverses the first then zips them together in the manner just described. For example,

$$mould\,([1,2],\ [3]) \ = \ [(One\ 2,\ One\ 3),\ (One\ 1,\ None)]$$

This is clearly highly parallel, and when applied to our strings, sets up the data for $dropmatches$. Our goal is now to implement the homomorphism from lists as returned by $mould$ to similar lists, with leading matching pairs removed. We can then pass its results through a corresponding $unmould$ function which reconstructs the implied list of "leftovers" from the whole $\oplus$ operation.

The important observation here is that the unmatched pairs from a concatenation of two such lists are either just those from the right component, if the entire left component is dropped, or the concatenation of those remaining from the left and the *entire original list* from the right otherwise. To complete the homomorphism, we will have to add copies of both original components to our baggage, and concatenate them as part of the result at each stage. The function applicable in the case of singleton lists (which we now label g) simply tests its pair.

The implementation of our homomorphism for $dropmatches$ is then

$$\begin{array}{lcl}
g\ (None,\ One\ c) & = & ([(None,\ One\ c)], [(None,\ One\ c)]) \\
g\ (One\ c,\ None) & = & ([(One\ c,\ None)], [(One\ c,\ None)]) \\
g\ (One\ c,\ One\ c') & = & (if\ (match\ c\ c')\ then\ [\,]\ else\ [(One\ c,\ One\ c')], \\
 & & \quad [(One\ c,\ One\ c')]) \\
 & & \\
(kept, original) \otimes & & \\
(kept', original') & = & (kept +\!\!+ (if\ kept = [\,]\ then\ kept'\ else\ original'), \\
 & & \ original +\!\!+ original') \\
 & & \\
dropmatches & = & fst \cdot (\otimes/) \cdot (g*)
\end{array}$$

Once again, but now at the inner level, we have a computation which can be implemented in $O(\log n)$ time on $O\left(\frac{n}{\log n}\right)$ processors, with shared memory, but which incurs a linear communications penalty otherwise. Now each such computation tree for $\otimes$ is embedded at a node of the outer level tree of $\oplus$ computations. As the number of $\oplus$ nodes decreases up the levels of the outer tree, so the length of arguments to the embedded $\otimes$ trees increases proportionately. Thus, given enough processors for optimal parallelism in the outer tree, we always have just enough processors available for optimal parallelism within each $\otimes$ tree.

In a shared memory implementation our execution time for the whole problem is therefore $O\left(\sum_{i=0}^{logn-1} log\frac{n}{2^i}\right)$ which is $O\left(log^2 n\right)$, on $O\left(\frac{n}{\log n}\right)$ processors.

7 A Parallel Dropwhile

In this section we present a second parallel algorithm for *dropmatches*. We note in passing that both this implementation and the one above have obvious generalizations to the standard filter function *dropwhile* (see the appendix). The new implementation has the advantage of removing the linear time communications bottleneck which occurs at the root of the $\otimes$ trees. It includes another instance of homomorphism based parallelism, but also requires a *scan* [5] and a *filter* to complete the process. Since both have efficient parallel implementations, and since the homomorphism involved uses only constant baggage, we achieve $O\left(log^2 n\right)$ time on $O\left(\frac{n}{\log n}\right)$ processors without the requirement for shared memory.

The algorithm has three phases. First, we count the number of leading matches in the list. This is achieved by first mapping each pair to 1 or 0, according to whether its contents are a match. We then apply a simple homomorphism from such lists to pairs of integers and booleans, indicating the number of leading 1s in the list, and whether the list was all 1s.

$$
\begin{aligned}
h\ x &= if\ (x = 1)\ then\ (1,\ true)\ else\ (0,\ true) \\
(m, ok) \odot (n, ok') &= if\ ok\ then\ (m + n,\ ok')\ else\ (m,\ false) \\
ones &= fst \cdot (\odot/) \cdot (h*)
\end{aligned}
$$

Next, we augment the items in the original list (of zipped pairs) to include an indication of their position within the list. This can be achieved with a simple scan

$$rank\ xs \ = \ zip\ (scan\ (+)\ (map\ (\lambda x.1))\ xs)$$

and then filter out the first *ones* of these, with a simple test of rank. Finally, we remove the rank indexes from those items which remain.

$$
\begin{aligned}
unrank &= map\ snd \\
keep\ i\ (j, -) &= i < j \\
dropmatches\ xs &= (unrank \cdot filter\ (keep\ ones) \cdot rank)\ xs
\end{aligned}
$$

8 Conclusions

We have provided further examples of an approach to parallel programming which focuses upon intuitive exploitation of a well founded theoretical result in the Bird-Meertens theory of lists. Once again it is of interest to ask how easily the resulting algorithms might have been derived in a more strictly formal setting.

Appendix

Useful Library Functions

```
fun fst (x,y) = x
fun snd (x,y) = y

fun curry f x y = f (x,y)
fun uncurry f (x,y)  = f x y

fun mem x []      = false
|   mem x (y::ys) = (x=y) orelse mem x ys;

exception problem of string
fun error message = raise (problem message)

fun filter p [] = []
|   filter p (x::xs) = if p x  then x::filter p xs
                       else filter p xs

fun dropwhile p [] = []
|   dropwhile p (x::xs) = if p x then dropwhile p xs
                          else x::xs

fun foldr f u [] = u
|   foldr f u (x::xs) = f x (foldr f u xs);


fun fold1 f [x] = x
|   fold1 f (x::xs) = f x (fold1 f xs);

fun fromto f t = if f>t then [] else f::fromto (f+1) t;

infix 5 --
val op -- = uncurry fromto;

fun listgen f from to = map f (from--to);

fun posinits xs =
let fun takexs n = take n xs
in
    listgen takexs 1 (length xs)
end;
```

```
fun scan1 f [] = []
|   scan1 f xs = map (fold1 f) (posinits xs);

fun andd x y = x andalso y;

fun zip [] []           = []
|   zip (x::xs) (y::ys) = (x,y)::zip xs ys;

fun min []      = 0
|   min (x::xs) = if x < (min xs) then x else min xs;
```

The Single Bracket Program

```
fun f "(" = (false, 1, 1)
|   f ")" = (false, ~1, ~1)
|   f  c  = (true, 0, 0);

fun cross (ok, final, low) (ok', final', low')
  = let val newok = (ok andalso ok') orelse
                    ((final+final' = 0) andalso
                     (low >=0) andalso (final+low' >= 0))
    in
        (newok, final+final', min [low, final+low'])
    end;


val  echeck  = fold1  cross o map f;

fun fst3 (a,_,_) = a;

val  check   = fst3 o echeck o explode;
```

Multibrackets with Single Level Parallelism

```
fun isbracket c = mem c [ "[", "]", "{", "}", "(", ")" ];

fun match c c'  = (c = "[") andalso (c' = "]") orelse
                  (c = "{") andalso (c' = "}") orelse
                  (c = "(") andalso (c' = ")");

fun combine (xs, ys) = (rev xs) @ ys;

(************************ Sequential dropmatches ************************)

fun dropmatches ([], ys)           = ([], ys)
|   dropmatches (xs, [])           = (xs, [])
|   dropmatches ((x::xs), (y::ys)) = if (match x y) then dropmatches(xs,ys)
                                     else ((x::xs),(y::ys));

(*********************************************************************)

fun f c  =  if isbracket c then [c] else []  ;

fun cross xs ys  =  (combine o dropmatches) (rev xs,ys);

val parcheck  = fold1 cross o map f o explode  ;
```

Multibrackets with Nested Homomorphisms

```
datatype 'a Maybe = One of 'a | None;

fun mould (xs, ys) =
   let fun zipplus ([], [])     = []
       |   zipplus (x::xs, [])  = (One x, None)::(zipplus (xs, []))
       |   zipplus ([], y::ys)  = (None, One y)::(zipplus ([], ys))
       |   zipplus (x::xs, y::ys)  = (One x, One y)::(zipplus (xs, ys))
in
   zipplus (rev xs, ys)
end;

fun unmould zs =
   let fun simple (One x) = x
       fun isOne  (One x) = true
       |   isOne  (None)  = false
       val xs' = (map simple o filter isOne o map fst) zs
       val ys' = (map simple o filter isOne o map snd) zs
in
   (rev xs') @ ys'
end;

(************ Parallel dropmatches as a near homomorphism *************)

fun g (None, None)    = error "Found a pair of Nones in g"
|   g (None, One c)   = ( [(None, One c)], [(None, One c)])
|   g (One c, None)   = ( [(One c, None)], [(One c, None)])
|   g (One c, One c') = (if (match c c') then []
                         else [(One c, One c')], [(One c, One c')] );


fun cross2 (kept, original) (kept', original') =
    (kept @ if (kept=[]) then kept' else original', original@original');

val dropmatches  = fst o foldr cross2 ([],[]) o map g;

(*********************************************************************)

fun f c  =  if isbracket c then [c] else []  ;

fun cross xs ys  =  (unmould o dropmatches o mould) (xs,ys);

val parcheck  = fold1 cross o map f o explode  ;
```

The Constant Overhead Dropmatches

```
fun star (m:int, ok) (n, ok') = if ok then (m+n, ok') else (m, false);

fun f2 x =  if (x=1) then (1, true) else (0, false);

val ones = fst o foldr star (0,true) o map f2;

fun rank xs = zip (scan1 add (map (fn x => 1) xs)) xs;

val unrank  = map snd;

fun dropmatches xs =
let val n = (ones o map (fn (x,y) => if onematch' (x,y) then 1 else 0)) xs
    fun keep (i:int) (j,_) = i<j
in
    (unrank o filter (keep n) o rank) xs
end;
```

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] D.T. Barnard, J.P. Schmeiser, and D.B. Skillicorn. Deriving Associative Operators for Language Recognition. *Bulletin of the EATCS*, 43:131–139, 1991.

[3] R.S. Bird. Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987.

[4] R.S. Bird. Algebraic Identities for Program Calculation. *Computer Journal*, 32(2):122–126, 1989.

[5] G.E. Blelloch. *Vector Models for Data Parallel Computing*. MIT Press, 1990.

[6] W. Cai and D.B. Skillicorn. Calculating Recurrences Using the Bird-Meertens Formalism. Submitted to *Science of Computer Programming*, 1992.

[7] W. Cai and D.B. Skillicorn. Evaluation of a Set of Message-Passing Routines on Transputer Networks. In A.R. Allen, editor, *Transputer Systems - Ongoing Research. Proceedings of the 15th WoTUG meeting.*, pages 24–35. IOS Press, 1992.

[8] M.I. Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. Technical Report CSR–25–93, Dept. of Computer Science, University of Edinburgh, 1993.

[9] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. To appear in the proceedings of PARLE 93, 1993.

[10] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[11] G.K Jouret. Exploiting Data-Parallelism in Functional Languages. *Ph. D. Thesis* , Department of Computer Science, Imperial College, 1991.

[12] P. Kelly. *Functional Programming for Loosely Coupled Multiprocessors.* Pitman, 1989.

[13] J.T. O'Donnell. Calculating a Parallel Algorithm for the Maximum of a Set of Naturals. In Proceedings of the Fourth Glasgow Workshop on Functional Programming, 1991.

[14] D.W.N. Sharp, A.J. Field, and H. Khoshnevisan. An Exercise in the Synthesis of Parallel Functional Programs for Message Passing Architectures. In W. Joosen and E. Milgrom, editors, *Parallel Computing: From Theory to Sound Practice. Proceedings of EWPC92*, pages 452–463. IOS Press, 1992.

[15] D.W.N. Sharp, P.G. Harrison, and J. Darlington. A Synthesis of a Dynamic Message Passing Algorithm for Quicksort. *Technical Report 91-19*, Department of Computer Science, Imperial College, 1991.

[16] D.B. Skillicorn. Architecture-Independent Parallel Computation. *Computer*, 23(12):38–51, 1990.

[17] D.B. Skillicorn. Parallelism and the Bird-Meertens Formalism. Unpublished, 1992.

[18] D.B. Skillicorn. Deriving parallel programs from specifications using cost information. *Science of Computer Programming*, 20(3), June 1993.

[19] D.B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *TR 92-329*, Department of Computing and Information Science, Queen's University, Kingston, 1992.