Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem

Murray Cole *

Abstract

We review the use of the Bird-Meertens Formalism as a vehicle for the construction of programs with massive implicit parallelism. We show that a simple result from the theory, concerning the expression of list homomorphisms, can help us in our search for parallel algorithms and demonstrate its application to some simple problems including the maximum segment sum problem. Our main purpose is to show that an understanding of the homomorphism lemma can be helpful in producing programs for problems which are "not quite" list homomorphisms themselves. A more general goal is to illustrate the benefits which can arise from taking a little theory with a pinch of pragmatic salt.

1 Introduction

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 [4] 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 [6].

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 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 [5, 7, 8, 9, 10, 11, 12, 14]. In the remainder of this paper, we concentrate on a particular instantiation of the method, and the application of one of its simplest results.

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

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 [2, 3]. 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 [5, 13, 14, 15, 16].

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" [2].

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

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

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*** *[2].*
A function h is a homomorphism with respect to $\mathbin{+\!\!+}$ 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

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.

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.

4.1 Maximum Segment Sum

Our first example is the maximum segment sum problem, mss. Given a list of integers, our goal is to find the contiguous segment of the list whose members have the largest sum among all such segments. The empty segment is defined to have sum 0. For simplicity, we will only consider the problem of returning the value of this sum. It is simple to make the amendment which also returns an indicator of its position within the list. As an example, we have

$$mss\ [2, -4, 2, -1, 6, -3] = 7$$

where the result is contributed by the segment [2, -1, 6].

The problem is widely studied in the world of program specification and transformation. In [3], Bird presents the derivation of an $O(n)$ sequential time algorithm (using a directed reduce) from a simple $O\left(n^3\right)$ sequential time specification. Similarly, in [5], Cai and Skillicorn derive an $O(\log n)$ parallel time algorithm from the same specification. The latter derivation includes the introduction of a reasonably complex new operator *recur-prefix*, originally derived in an investigation into the solution of recurrence relations.

Taking the "near homomorphism" approach, we note immediately that mss is not a list homomorphism. If $l = x +\!\!+\ y$, then while it may well be the case that $mss\ l = (mss\ x) \uparrow (mss\ y)$ we must not overlook the possibility that the true segment of interest lies partly in x and partly in y. This is the consideration which will produce the extra baggage needed to create an associated homomorphism. A little thought (wherein lies the intuition) reveals that the third possibility requires us to consider the segment constructed from a "maximum concluding segment" (mcs) of x (which might simply be empty, with sum 0) and the "maximum initial segment" (mis) of y. There can be no other possibilities - we already have sufficient information about all segments which lie entirely in either x or y. Thus we must add to our baggage these two extra values, for each segment. The $\oplus$ of our homomorphism is beginning to take shape. We now have sufficient information to generate the mss for a concatenation from the information about its two components -

$$mss(x \mathbin{+\!\!+} y) = (mss\ x) \uparrow (mss\ y) \uparrow (mcs\ x + mis\ y)$$

To produce a valid homomorphism, we must also generate the mis and mcs values for the concatenated list $x \mathbin{+\!\!+} y$. Again, a little thought reveals that the mis of a concatenation is the larger of the mis of the left argument and the sum of the whole left argument and the mis of the right argument (remember that this can be 0). Calculation of the mcs of the concatenation follows symmetrically. Thus, we must add the total sum (ts) of the entire segment to our baggage. The process ends here, since the total sum of a concatenation is simply the sum of its components' sums.

We can now define our homomorphism, which we shall call $emss$ (for *extended* mss). It is a function from integer lists to integer quadruples (mss, mis, mcs, ts), and the $\oplus$ operator is defined by the equation

$$\begin{aligned}
& (mssx, misx, mcsx, tsx) \oplus (mssy, misy, mcsy, tsy) \\
= \ & ((mssx) \uparrow (mssy) \uparrow (mcsx + misy), misx \uparrow (tsx + misy), \\
& (mcsx + tsy) \uparrow (mcsy), tsx + tsy)
\end{aligned}$$

It only remains to define the function f which, when mapped, describes the operation of the homomorphism on singleton lists. This is quite straightforward. The first three values of the quadruple for a singleton list are the single value if this is non-negative, zero otherwise. The total sum is simply the value. Thus,

$$f\ x = (x \uparrow 0,\ x \uparrow 0,\ x \uparrow 0,\ x)$$

The homomorphism lemma now presents us with a highly parallel algorithm for $emss$ from which we can trivially pluck the mss result.

$$\begin{aligned}
emss &= (\oplus /) \cdot (f*) \\
mss &= fst \cdot emss \\
& \quad\ where\ fst\ (a,\ -,\ -,\ -) = a
\end{aligned}$$

In practical terms, the algorithm looks promising - the baggage has only introduced three extra integers per communication and a constant number of simple integer operations. On many architectures we can expect an $O(\log n)$ algorithm on $O\left(\frac{n}{\log n}\right)$ processors.

4.2 Longest Satisfying Segment Problems

We now consider a class of related problems which require us to find the longest segment of a list which holds some property. For example, we might be interested in the longest sequence of zeros, or the longest sequence of the same number, or the longest sorted sequence. Using an approach similar to that of the previous section, we will build a highly parallel program schema for problems of this type.

The most general formulation of such a class of problems would simply be parameterized by a predicate on lists. Unfortunately, this is turns out to be too flexible to produce an efficient parallel algorithm with our approach. A key characteristic of the *mss* problem was that we could determine important facts about a concatenation (e.g. its sum) from *summarized* information about its components (their sums). This is not the case with such a general notion as a list predicate. For example, consider the predicate which is satisfied by segments whose contents average to zero. Then the length of the longest satisfying segment of the concatenation of $[2, 1, 1]$ and $[-2, 3, 1]$ is 3, even though all the segment lengths contributed to the $\oplus$ by the standard summaries (longest initial segment and so on) would be zero. Any scheme which finds the correct solution needs access to the complete lists, and adding this data to our parallel baggage reduces efficiency to a useless level. We will return to this phenomenon in the next section.

Here, we restrict our attention to properties p which can be described in terms of the following schema

$$\begin{array}{lll} p & [] & = true \\ p & [x] & = ... \\ p & [x, y] & = ... \\ p & (x : y : zs) & = p[x, y] \wedge p(y : zs) \end{array}$$

For the three examples discussed above, we would have

$$\begin{array}{llcl} zeros & [x] & = & x = 0 \\ zeros & [x, y] & = & (zeros\ [x]) \ \wedge\ (zeros\ [y]) \\ \\ same & [x] & = & true \\ same & [x, y] & = & x = y \\ \\ sorted & [x] & = & true \\ sorted & [x, y] & = & x <= y \end{array}$$

As before, we note immediately that such problems are not homomorphic - we must consider the longest satisfying segment (lss) which straddles the concatenation as well as those which are offered independently by each component. Thus we will require baggage analogous to that in the *emss* problem, namely the lengths of the longest initial and concluding satisfying segments and the total length of the entire list, which we shall label lis, lcs and tl respectively.

The $\oplus$ operation will be similar to that required for *emss*, but with an additional complication. When considering the concatenation of a pair of concluding and initial satisfying segments, it is not guaranteed in general that the result also satisfies the predicate. For example, the concatenation of two sorted sequences may not be sorted. On the other hand, the concatenation of two sequences of zeros is a sequence of zeros. Additionally, in any computation involving the length of a complete list (tl), we have to be aware of whether or not that complete list actually satisfies the predicate. Thus, our additional baggage must include the last element of each longest concluding satisfying sequence ($last$), the first element of each of each longest initial satisfying sequence ($first$) and boolean indications of whether or not the whole list components of the concatenation each satisfy the predicate independently (xok and yok). With this information for each of the components of the concatenation we can construct our generic $\oplus$ for the problem class.

$$
\begin{aligned}
&(lssx,\ lisx,\ lcsx,\ tlx,\ firstx,\ lastx,\ xok) \oplus \\
&(lssy,\ lisy,\ lcsy,\ tly,\ firsty,\ lasty,\ yok) \\
&= (newlss,\ newlis,\ newlcs,\ tlx + tly,\ firstx,\ lasty,\ newok) \\
&where \\
&\qquad \begin{array}{lcl}
connect & = & p\,[lastx,\ firsty] \\
newlss & = & lssx \uparrow lssy \uparrow (if\ connect\ then\ lcsx + lisy\ else\ 0) \\
newlis & = & lisx + (if\ okx\ \wedge\ connect\ then\ lisy\ else\ 0) \\
newlcs & = & lcsy + (if\ oky\ \wedge\ connect\ then\ lcsx\ else\ 0) \\
newok & = & okx\ \wedge\ oky\ \wedge\ connect
\end{array}
\end{aligned}
$$

Finally, our generic f is

$$
\begin{array}{lcl}
f\ x & = & (xfits,\ xfits,\ xfits,\ 1,\ x,\ x,\ p\,[x]) \\
 & & where\ xfits\ =\ if\ p\,[x]\ then\ 1\ else\ 0
\end{array}
$$

As for the maximum segment sum, all that remains is to apply the homomorphism lemma and extract the first component of the resulting tuple. We then have a fast parallel algorithm for any problem in the class, with only a constant amount of computational and communications baggage.

4.3 On Taking It All Too Far - Finding the Mode

We conclude with a simple example which illustrates that the generality of the proposed approach can admit solutions which are of no practical utility from the perspective of parallel implementation, due to the excessive baggage involved. The problem is to find the mode of a list of integers. Clearly the problem is not a list homomorphism - knowing only the modes of [1,2,1,2,1,1,2] and [2,3,2,3,3,3,2] does not lead us to the mode of their concatenation. Much worse, it appears[1] that any homomorphism into which the problem can be embedded requires baggage proportional to the length of the list, and as much work in the $\oplus$ as it would take to solve the problem directly in a single pass ($\Omega\,(n \log n)$) in a sequential comparison model. A similar problem was observed in the previous section with respect to our initial, very general notion of “satisfying segment”.

Of course, this is just a reflection of the observation that we can embed any function on lists in a homomorphism which simply throws away partial results and

[1] Although evidence to the contrary is invited.

solves the problem directly at each concatenation. Not surprisingly, these cases do not result in useful parallel algorithms.

In passing, we note that finding the mode of a *sorted* sequence can be easily expressed in similar style to the longest satisfying sequence problems above.

5 Summary and Conclusions

We have presented an informal methodology for the derivation of parallel algorithms for a variety of problems on lists. The method relies upon a degree of intuition supported by an understanding of principles and results drawn from a more formal methodology. Thus the justification of the algorithms' correctness remains intuitive rather than being verifiable by a sequence of checkable transformations. It would be interesting to discover the facility with which the algorithms could be generated and verified in a more formal setting. An obvious complication is the fact that the final algorithms actually solve problems which subsume the original specifications.

6 Acknowledgements

Thanks are primarily due to David Skillicorn for his work on parallel BMF which has inspired the observations described here. Thanks also go to Mark Jerrum for the lower bound result on finding the mode sequentially, and to Richard Miller and David Skillicorn for their helpful comments on this paper.

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

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

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

[5] W. Cai and D.B. Skillicorn. Calculating Recurrences Using the Bird-Meertens Formalism. Unpublished, 1992.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] D.B. Skillicorn. Deriving Parallel Programs from Specifications Using Cost Information. *TR 91-316*, Department of Computing and Information Science, Queen's University, Kingston, 1991.

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

[16] 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.