Systematic Efficient Parallelization of Scan and Other List Homomorphisms

Sergei Gorlatch*

University of Passau, D–94030 Passau, Germany

Abstract. Homomorphisms are functions which can be parallelized by the divide-and-conquer paradigm. A class of *distributable homomorphisms* (DH) is introduced and an efficient parallel implementation schema for all functions of the class is derived by transformations in the Bird-Meertens formalism. The schema can be directly mapped on the hypercube with an unlimited or an arbitrary fixed number of processors, providing provable correctness and predictable performance. The popular *scan*-function (parallel prefix) illustrates the presentation: the systematically derived implementation for *scan* coincides with the practically used "folklore" algorithm for distributed-memory machines.

1 Introduction

This paper deals with formal design of parallel programs. We advocate that the issues of correctness and performance should and can be addressed during the design/derivation process, rather than as an afterthought.

As a derivational calculus we use the *Bird-Meertens Formalism* (BMF) [1]. Computations are specified using a set of higher-order functions over lists and other data structures; the specification is refined into an executable form by semantically sound transformation rules, which guarantees the *correctness* of the target program. We structure the derivation process, by exposing the points where the design decisions are made and estimating the target *performance*. Such a structuring naturally leads to extracting the typically used and efficiently implementable classes (templates, skeletons) of parallelism.

In this paper, we study functions, called *homomorphisms*, which are parallelizable using the important divide-and-conquer paradigm. We define a class, called *Distributable Homomorphisms* (DH), and derive an efficient and provably correct parallel implementation schema for all functions of the class.

As an illustrating example, we use the *scan* (parallel prefix) function, which encapsulates a computational pattern common for many parallel applications [2]. The specialization of our parallel implementation schema for the case of *scan* yields the known parallel algorithms for *scan* on a hypercube with either a linear or an arbitrary fixed number of processors. In contrast to the usual *ad hoc* presentation of these algorithms, we derive them in a systematic sequence of design steps, which are methodologically substantiated and formally correct.

* The author was partially supported by the DAAD cooperation programs ARC and PROCOPE, and by the Project INTAS-93-1702.

2 BMF, Homomorphisms and Scan

We use a variant of the Bird-Meertens Formalism (BMF) with non-empty lists of length 2^k, $k = 0, 1, \cdots$ (powerlists [3]). Function *length* yields the length of a list. The constructors are: (i) $[.]$ yielding the *singleton list* and (ii) *balanced concatenation* $+\!\!+$, where $x +\!\!+ y$ is defined iff $length\, x = length\, y = 2^k$.

We use the following functions and functionals, defined informally:

$\circ$	backward functional composition;
id	the identity function;
$map\, f$	*map* of an unary function f, i.e., $map\, f\, [x_1, \cdots, x_n] = [f\, x_1, \cdots, f\, x_n]$;
$red\,(\odot)$	*balanced reduce* with a binary associative operation $\odot$, where $red(\odot)\,[a] = a$, $red(\odot)\,(x +\!\!+ y) = (red(\odot)\, x) \odot (red(\odot)\, y)$;
$zip\,(\odot)$	component-wise application of $\odot$ to a pair of lists of equal length: $zip\,(\odot)\,(\,[x_1, \cdots, x_n]\,, [y_1, \cdots, y_n]\,) = [\,(x_1 \odot y_1), \cdots, (x_n \odot y_n)\,]$;
$< >$	"zipped tupling": for a tuple of functions $f_i : [\alpha] \to [\alpha], i = 1, \cdots n$, function $< f_1, \cdots, f_n >$ yields the list of result tuples.

Definition 1. A list function h is a *homomorphism* iff there exists a binary associative *combine operator* $\oslash$, such that for all lists x and y:

$$h\,(x +\!\!+ y) = h\,(x) \oslash h\,(y) \tag{1}$$

i.e., the value of h on a list depends in a particular way (using $\oslash$) upon the values of h on the pieces of the list.

The computations of $h(x)$ and $h(y)$ are independent and can be carried out in parallel, which expresses the well-known divide-and-conquer paradigm.

Theorem 2 (Bird [1]). *Function h is a homomorphism iff:*

$$h = red\,(\oslash) \circ map\,(f) \tag{2}$$

where $\oslash$ is from (1) and $f\,(a) = h\,([a])$.

Theorem 2 provides a common parallelization for all homomorphisms as a composition of two *stages* [4]: the first, *map*, is totally parallel, the second, *red*, can be parallelized on a tree-like structure, with $\oslash$ applied in the nodes.

There are two problems for a given function: first, how to find the combine operator $\oslash$ of (2) and, second, how to implement the *red* stage efficiently in parallel. In [5], we described a systematic approach to constructing the combine operator, starting from two sequential representations of the given function. The present paper deals with the second problem, the implementation.

For a homomorphism of type $[\alpha] \to \alpha$, there are a logarithmic number of steps in the tree computation, with communications of constant size, which yields an efficient algorithm. However, when a homomorphism yields a list, i.e., its combine operator contains $+\!\!+$, then the tree implementation requires linear execution time, independently of the number of processors [6].

Scan as a Homomorphism. Our illustrating example is the *scan*-function which, for associative $\odot$ and a list, computes "prefix sums", e.g.:

$$scan\,(\odot)\,[a, b, c, d] = [a,\ (a \odot b),\ (a \odot b \odot c),\ (a \odot b \odot c \odot d)]$$

Function *scan* is a homomorphism with combine operator $\oslash$:

$$scan\,(\odot)\,(x +\!\!\!+\, y) \;=\; S_1 \oslash S_2 \;=\; S_1 \;+\!\!\!+\; (map\,(last(S_1)\,\odot)\; S_2)\,, \qquad (3)$$
$$\text{where } S_1 \;=\; scan\,(\odot)\,x\,,\ S_2 \;=\; scan\,(\odot)\,y.$$

Here, so-called *sectioning* $(a\,\odot)$ is used: $(a\,\odot)\;b \;=\; a \odot b$.

Despite the fact that $\oslash$ contains $+\!\!\!+$, there exist efficient parallel algorithms for *scan* [7, 8], which, rather than producing a monolithic output list, distribute it between processors. Our goal is to derive such algorithms systematically.

3 Distributable Homomorphisms

We introduce a specific class of homomorphisms by restricting the permitted form of the combine operator.

Definition 3. For two binary associative operations $\oplus$ and $\otimes$ on elements, the combine operator $\oplus\!\!\bowtie\!\!\otimes$ on lists is defined as follows:

$$u \;\oplus\!\!\bowtie\!\!\otimes\; v \;=\; zip\,(\oplus)\,(u, v) \;+\!\!\!+\; zip\,(\otimes)\,(u, v) \qquad (4)$$

We write $\oplus\!\updownarrow\!\otimes$ for the following homomorphism with combine operator $\oplus\!\!\bowtie\!\!\otimes$:

$$\oplus\!\updownarrow\!\otimes\;[a] = [a] \qquad (5)$$
$$\oplus\!\updownarrow\!\otimes\;(x +\!\!\!+\, y) = ((\oplus\!\updownarrow\!\otimes)\;x)\;\;\oplus\!\!\bowtie\!\!\otimes\;\;((\oplus\!\updownarrow\!\otimes)\;y)$$

Definition 4. Function $h\;:\;[\alpha]\;\rightarrow\;[\alpha]$ is a *distributable homomorphism* (DH) iff $h \;=\; \oplus\!\updownarrow\!\otimes$, for some $\oplus$ and $\otimes$.

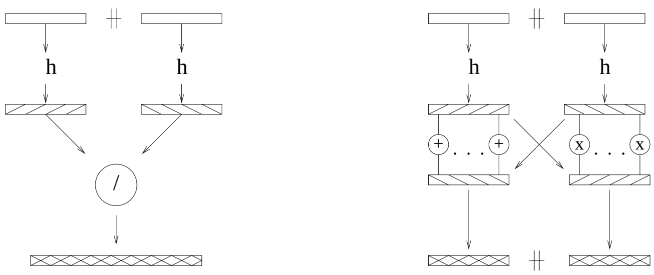

Fig. 1. A general homomorphism (left) and a distributable homomorphism (right), computed on a concatenation of two lists

The "distributed reduction": $redd\,(\odot)\,x \;=\; [red\,(\odot)\,x, red\,(\odot)\,x, \cdots, red\,(\odot)\,x]$ is obviously a homomorphism with the combine operator:

$$R_1 \;\ominus\; R_2 \;=\; zip\,(\odot)\,(R_1, R_2) \;+\!\!\!+\; zip\,(\odot)\,(R_1, R_2) \qquad (6)$$

This fits format (4), thus *redd* is a DH:

$$redd\,(\odot) \;=\; \odot \updownarrow \odot \tag{7}$$

Function *redd* is implemented as `ReduceAll` in the recent MPI standard [9].

Scan: Adjusting to DH. Let us try to express the right-hand side of (3) in format (4), i.e., with both arguments of $+\!\!\!+$ in *zip*ped form. This is easy for the left argument but requires an additional function for the right argument of $+\!\!\!+$:

$$S_1 \;=\; zip\,(\pi_1)\,(S_1, S_2) \tag{8}$$

$$map\;(last\,(S_1)\,\odot)\;S_2 \;=\; zip\,(\odot)\,(R_1, S_2) \tag{9}$$

where π_1 yields the first element of a pair and $R_1 \;=\; redd\,(\odot)\,x$. We can thus obtain the desired format if we "tuple" *scan* together with *redd* into new function $<scan, redd>$, which can be adjusted to (4) by transition to lists of pairs.

The target expression of *scan* is as follows:

$$scan\,(\odot) \;=\; (map\,\pi_1) \,\circ\, (\,\oplus\updownarrow\otimes\,) \,\circ\, map\,(pair) \tag{10}$$

$$\begin{aligned} \text{where}\quad & pair\,a \;=\; (a, a) \\ & (s_1, r_1) \oplus (s_2, r_2) \;=\; (s_1\,,\; r_1 \odot r_2) \\ & (s_1, r_1) \otimes (s_2, r_2) \;=\; (r_1 \odot s_2\,,\; r_1 \odot r_2) \end{aligned} \tag{11}$$

Operations $\oplus$ and $\otimes$ work on pairs and are read off from (6), (8) and (9).

The systematic adjustment of *scan* to the DH format yields exactly the pair structure, its initialization by function *pair* and the computations $\oplus$ and $\otimes$ in (11), which are used *ad hoc* in the known efficient algorithms for *scan*.

4 Towards a Hypercube Implementation

Our goal is to find a provably correct and efficient parallel implementation for all DH functions. As a particular parallel topology for lists of length $n = 2^k$, we take a k-dimensional hypercube with n nodes. We use the standard encoding: the position l, $0 \le l < n$, of the list is stored in the hypercube node, whose index is the k-bit representation of l. Processor l can communicate with its k neighbours; the neighbour in dimension d is $xor\,(l,\,2^{d-1})$, where xor is the bit-wise exclusive OR. So, the only difference between a list of length $n = 2^k$ and the k-dimensional hypercube is that in the latter we have random element access and communication primitives. To make this analogy more visible, we abuse the typing by using $[\alpha]$ for both types.

We introduce a pattern of the hypercube behaviour, skeleton *swap*, which describes a pair-wise communication in dimension d, followed by a computation with $\oplus$ and $\otimes$, such that for an index l of a list x:

$$swap\;d\;(\oplus,\otimes)\;x\;l \;=\; \begin{cases} x(l) \oplus x(xor(l,\,2^{d-1}))\,, & \text{if } l < xor(l,\,2^{d-1}) \\ x(xor(l,\,2^{d-1})) \otimes x(l)\,, & \text{otherwise} \end{cases}$$

$$\text{where}\;\; length\,(x) \;=\; 2^k\,,\; 1 \le d \le k\,,\; 0 \le l < 2^k.$$

Let us define $swap^k \;=\; (swap\,k) \,\circ\, \cdots (swap\,2) \,\circ\, (swap\,1)$.

Proposition 5. *Every DH function is implementable as follows:*

$$\oplus \updownarrow \otimes \;=\; swap^k\,(\oplus, \otimes) \tag{12}$$

Scan: the Hypercube Implementation. Implementation (12) is of a general nature; for a particular function, it suffices to substitute concrete operations for $\oplus$ and $\otimes$. From (10) and (12) we obtain the following program for *scan*:

$$scan\,(\odot) \;=\; map\,(\pi_1) \;\circ\; swap^k\,(\oplus, \otimes) \;\circ\; map\,(pair) \tag{13}$$

where *pair*, $\oplus$ and $\otimes$ are defined by (11).

This is the well-known "folklore" implementation [7]. In Figure 2, it is illustrated on the 2-dimensional hypercube which is computing $scan\,(+)\,[1,2,3,4]$.

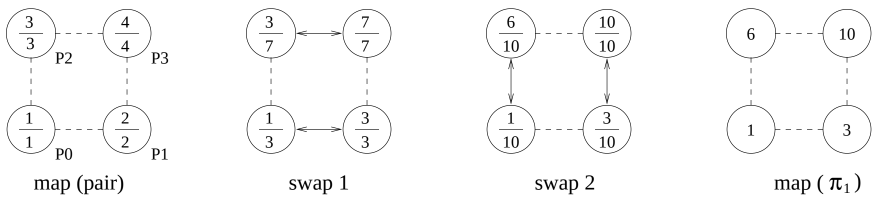

Fig. 2. Computing *scan* on a hypercube

The time complexity of computing *scan* for a list of length n, on n processors, is $O\,(\log n)$. The cost (time-processor product [7]) is $O\,(n \log n)$, whereas the cost of the sequential computation is $O\,(n)$. So the implementation is *time optimal* but *not cost optimal*.

5 Bounded Number of Processors

Let us now consider the more practical situation, where the processor number p is arbitrary but fixed: $p < n$, where p divides n.

We introduce the type $[\alpha]_p$ of lists of length p and use the notation map_p, etc. for functions defined on such lists. We take the approach from [6]: the input list is distributed over p sublists, which are called *blocks*. This is done by the *distribution function*, $dist\,(p) \,:\, [\alpha] \,\rightarrow\, [[\alpha]]_p$.

The following equality relates distribution with its reverse, flattening:

$$red\,(+\!\!+) \;\circ\; dist\,(p) \;=\; id \tag{14}$$

Homomorphisms have the following important property.

Theorem 6 (Promotion [1]). *For homomorphism h with combine operator $\oslash$:*

$$h \;\circ\; red\,(+\!\!+) \;=\; red\,(\oslash) \;\circ\; (map\; h) \tag{15}$$

Now, we can start to transform $\oplus \updownarrow \otimes$:

$\oplus \updownarrow \otimes$

$=$ { equality (14), two times }

$(red(+\!\!\!+) \circ dist(p)) \circ \oplus \updownarrow \otimes \circ (red(+\!\!\!+) \circ dist(p))$

$=$ { associativity of $\circ$, promotion law (15) }

$red(+\!\!\!+) \circ dist(p) \circ red(\oplus\!\!\bowtie\!\!\otimes) \circ map_p(\oplus \updownarrow \otimes) \circ dist(p)$

We separate the first, distributing and the last, collecting stage of the result expression, and assume that these stages are implemented by the environment. The remaining, middle part both accepts and yields distributed data of type $[[\alpha]]_p$. For an arbitrary function h, we call such expression the *p-distributed version* of h and use notation $(\widetilde{h})_p$ for it. For a DH, we have:

$$(\widetilde{\oplus \updownarrow \otimes})_p = dist(p) \circ red(\oplus\!\!\bowtie\!\!\otimes) \circ map_p(\oplus \updownarrow \otimes) \tag{16}$$

Program (16) is not optimal: it concatenates and then redistributes the list.

Proposition 7. Redistribution Elimination Rule:

$$dist(p) \circ red(\oplus\!\!\bowtie\!\!\otimes) = ((zip \oplus) \updownarrow (zip \otimes))_p \tag{17}$$

The redistribution elimination (17) applied to (16), together with (12) yields:

$$(\widetilde{\oplus \updownarrow \otimes})_p = swap_p^k(zip(\oplus), zip(\otimes)) \circ map_p(\oplus \updownarrow \otimes) \tag{18}$$

This is a common p-processor implementation of a DH function. It consists of two stages: a sequential computation of the function in all p processors simultaneously on their blocks, and then a sequence of swaps on the hypercube. This implementation has been derived formally and, thus, is provably correct. For an input list of length n, the *swap*-stage requires $\log p$ steps, with blocks of size n/p to be sent and received and sequential component-wise computations on them; this yields a total time $O((n/p) \cdot \log p)$.

Scan: Efficient Implementation. Since there are *scan* algorithms with better performance than the general case of DH, we should use some special properties of*scan*. The direct specialization of the general program (16) for *scan* is as follows:

$$(\widetilde{scan})_p(\odot) = dist(p) \circ red(\oslash) \circ map_p(scan(\odot)) \tag{19}$$

Here, $\oslash$ is from (3). Our goal is to specialize the Redistribution Elimination Rule (17) for the case of such combine operator.

The idea of the transformation is that, instead of performing computations in the blocks at each step of the reduction, we accumulate step-by-step one value for each block and then perform (in one go) the necessary computation across the blocks. For element a and block u, the latter computation can be defined as an operation $\odot\!\!\lhd$, such that $a \odot\!\!\lhd\, u = map(a \odot)\, u$.

For $\oslash$ of the form (3), the Redistribution Elimination Rule becomes then:

$$(dist(p) \circ red(\oslash))\, x = zip_p(\odot\!\!\lhd)(y, x), \tag{20}$$
$$\text{where } y = (prescan_p(\odot) \circ map_p(last))\, x$$

Here, zip_p is applied to lists of length p and is directly parallelizable like map_p.

The list y of accumulating values is computed by the *prescan* function, which yields the result of *scan*, "shifted to the right":

$$prescan\,(\odot)\,[x_1, x_2, \cdots, x_n] \;=\; [0_\odot,\; x_1,\; x_1 \odot x_2,\; \cdots,\; x_1 \odot x_2 \odot \cdots \odot x_{n-1}]$$

where $0_\odot$ is the neutral element of $\odot$.

Despite its simplicity and analogy to *scan*, function *prescan* is not a homomorphism, but it can be adjusted to the DH format exactly like *scan*, with the only difference in the pairing function, which now is of the form $prepair\, a \;=\; (0_\odot, a)$. A parallel implementation of the *prescan* function is then of the same three-stage form as the *scan* implementation (13). After substituting it into (20) and fusing two *map*s, we obtain from (19) the following target algorithm for $(\widetilde{scan})_p$:

$$\begin{aligned} (\widetilde{scan})_p\,(\odot)\;x \;&=\; zip_p\,(\odot\!\lhd)\,(y\;,\;z)\,, \\ \text{where}\quad z \;&=\; map_p\,(scan\,(\odot))\;x \\ y \;&=\; (map_p\,(\pi_1)\;\circ\;swap_p^k\,(\oplus,\otimes)\;\circ\;map_p\,(prepair\;\circ\;last))\;z \end{aligned} \tag{21}$$

with $\oplus, \otimes$ from (11) and $k = \log p$.

Implementation (21) has three stages:

- Compute z: each processor applies the *scan* function to its block of x.
- Compute y: after picking the last elements of their blocks of z and initializing pairs by *prepair*, the processors work together in $\log p$ swaps; the computations $\oplus$ and $\otimes$ are defined by (11).
- Compute the result: each processor adds (in the sense of $\odot$) its element of y to each element of its block of z.

This is exactly the known algorithm for *scan* on a hypercube with an arbitrary fixed number of processors [7]. Its complexity is $O(n/p + \log p)$, which is a clear improvement over the implementation (18) in the general case of DH.

6 Conclusion

This paper makes a contribution to parallel programming methodology by giving a definition of the DH class of functions on lists, and a formal derivation of an efficient parallel implementation schema for all functions of the class on a hypercube with either a linear or an arbitrary fixed number of processors. The derivation is based on the semantically sound transformation rules of the BMF, which guarantees its correctness. The performance of the target implementations is easily predictable and conforms with the known estimates.

An argument for the practicality of our approach is that the DH class includes the important *scan* function and that the specialization of the common implementation yields the parallel algorithm for *scan*, which is nowadays considered to be the best in practice [8]. The structure of the parallel implementation for DH is similar to the ascending algorithms of [10], which provides confidence that other important application algorithms can be derived in a similar way.

Because of the lack of of space, we only mention the related work on divide-and-conquer [11], formal derivation of *scan* algorithms [12, 13, 14], parallelizing transformations in BMF [6] and transition from functional to parallel imperative representations [15]. An extended version of the paper with the full comparison to the related work and other technical details is available in WWW from `http://www.brahms.fmi.uni-passau.de/cl/index-gorlatch`.

References

1. R. S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, NATO ASO Series F: Computer and Systems Sciences. Vol. 55, pages 151–216. Springer Verlag, 1988.
2. G. Blelloch. Scans as primitive parallel operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.
3. J. Misra. Powerlist: a structure for parallel recursion. *ACM TOPLAS*, 16(6):1737–1767, 1994.
4. S. Gorlatch. Stages and transformations in parallel programming. In M. Kara et al., editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 147–162. IOS Press, 1996.
5. S. Gorlatch. Constructing list homomorphisms. Technical Report MIP-9512, Universität Passau, 1995.
6. D. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing*, 28:65–83, 1995.
7. M. J. Quinn. *Parallel Computing*. McGraw-Hill, Inc., 1994.
8. M. Reid-Miller. List ranking and list scan on the Cray C-90. In *Proceedings SPAA'94*, pages 104–113, 1994.
9. D. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.
10. F. Preparata and J. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, 1981.
11. Z. G. Mou. Divacon: A parallel language for scientific computing based on divide and conquer. In *Proc. 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 451–461, October 1990.
12. J. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 4(3):329–338, 1994.
13. J. Gibbons. Upwards and downwards accumulations on trees. In R. Bird, C. Morgan, and J. Woodcock, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science 669, pages 122–138, 1992.
14. J. Kornerup. Mapping a functional notation for parallel programs onto hypercubes. *Information Processing Letters*, 53:153–158, 1995.
15. K. Achatz and W. Schulte. Architecture independent massive parallelization of divide-and-conquer algorithms. In B. Moeller, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science 947, pages 97–127, 1995.

This article was processed using the LaTeX macro package with LLNCS style