Segmented Diffusion Theorem

Zhenjiang Hu†*, Tomonari Takahashi†, Hideya Iwasaki‡, Masato Takeichi†

†The University of Tokyo
‡The University of Electro-Communications
*PRESTO 21, Japan Science and Technology Corporation
{hu,tomonari,takeichi}@ipl.t.u-tokyo.ac.jp
iwasaki@cs.uec.ac.jp

Abstract— **Skeletal parallel programming ease parallel programming by providing efficient ready-made skeletons. However, nested use of skeletons are difficult to be implemented in an efficiently and load unbalanced way. In this paper, we propose a novel transformation theorem called the segmented diffusion theorem, an extension of the diffusion theorem, which can flatten many nested uses of skeletons into more efficient skeletal parallel programs. This theorem is not only useful for guiding construction of efficient skeletal parallel programs, but also suitable for optimizing skeletal parallel programs in compilers.**

Keywords—**Parallel Programming, Program Transformation, Nested Parallelism, Parallelization**

I. Introduction

Parallel skeletons are a set of functions that make both programming and parallelization easier. By using parallel skeletons with efficient implementations, programmers can write efficient parallel programs without the knowledge of details of parallel algorithms or hardwares. Programming using parallel skeletons is called *skeletal parallel programming* [Col89], [DFH+93], [Ski94], or simply *skeletal programming*. Examples of such parallel skeletons are higher order functions such as *map*, *reduce* and *scan* in BMF [Ski92].

These skeletons are certainly efficient enough for *flat* data structures, e.g. list of integers, by distributing almost the same number of elements to each processor. However, for the case of *nested* data structures, e.g. list of lists of integers, simply distributing the same number of elements to each processor cannot lead to good results. Consider the following program:

$$maxs\ ass \ = \ maximum * ass.$$

which maps the function *maximum* to a list of lists to obtain maximum for each element list. Suppose $ass = [as_1, as_2, \ldots, as_n]$, then this expression is inefficiency when the lengths of as_1, as_2, …, as_n are of big difference. To see this clearly, consider the extreme case of

$$maximum * [[1], [2], [1, 2, 3, 4, \ldots, 100]],$$

and assume that we have three processors. If we naively use one processor to compute *maximum* on each element list according to the parallel semantics of *map*, computation time will be dominated by the processor which computes *maximum* $[1, 2, \ldots, 100]$, and the load imbalance cancels the effect of parallelism in the *map* skeleton. In fact, nested use of skeletons usually suffer such problem.

In this paper, we are investigating that under what condition of f, the computation of

$$f * [xs_1, xs_2, \ldots, xs_n]$$

can be implemented efficiently no matter how different the lengths of the element lists are. We shall propose a powerful theorem called *segmented diffusion theorem*, based on which an algorithm is given for flattening nested skeletons systematically.

The organization of this paper is as follows. After briefly reviewing the notational conventions and some basic concepts in the BMF parallel model in Section II, we give the segmented diffusion theorem in Section III and show the strategy of applying the theorem for flattening nested skeletons in Section IV. Section V applies the theorem to obtain an efficient skeletal parallel programs for the XML tag-matching problem, and gives the experimental results. Section VI concludes the paper.

II. BMF and Skeletal Parallel Programming

We will address our method on the Bird Meertens Formalisms (BMF for short) [Bir87], a data parallel programming model [Ski90], though the method itself is not limited to the BMF model. We choose BMF because it can provide a concise way to describe both programs and transformation of programs. Those who are familiar with the functional language Haskell [JH99] should have no problem in understanding the programs in this paper. From the notational viewpoint, the main difference is that we use more symbols or special parentheses to shorten the expressions so that manipulation of expressions can be described in a more concise way.

A. Functions

Function application is denoted by a space and the argument which may be written without brackets. Thus $f\,a$ means $f\,(a)$. Functions are curried, and application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. Function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, not $f\,(a \oplus b)$. *Function composition* is denoted by a centralized circle $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function composition is an associative operator, and the identity function is denoted by *id*. Infix binary operators will often be denoted by $\oplus, \otimes$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary or binary functions by $a \oplus b = (a \oplus)\,b = (\oplus\,b)\,a = (\oplus)\ a\ b$.

B. Parallel Data Structure: Join Lists

Join lists (or *append lists*) are finite sequences of values of the same type. A list is either the empty, a singleton, or the concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a, and $x \mathbin{+\!\!+} y$ for the concatenation (join) of two lists x and y. Concatenation is associative, and $[\,]$ is its unit. For example, $[1] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} [3]$ denotes a list with three elements, often abbreviated to $[1, 2, 3]$. We also write $a : x$ for $[a] \mathbin{+\!\!+} x$.

C. Parallel Skeletons: map, reduce, scan, zip

It has been shown [Ski90] that BMF is a nice architecture-independent parallel computation model, consisting of a small fixed set of specific higher order functions which can be regarded as the primitive parallel skeletons suitable for parallel implementation. Four important higher order functions are *map*, *reduce*, *scan* and *zip*.

Map is the skeleton which applies a function to every element in a list. It is written as an infix $*$. Informally, we have

$$k * [x_1, x_2, \ldots, x_n] = [k\,x_1, k\,x_2, \ldots, k\,x_n].$$

Reduce is the skeleton which collapses a list into a single value by repeated application of some associative binary operator. It is written as an infix $/$. Informally, for an associative binary operator $\oplus$, we have

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

Scan is the skeleton that accumulates all intermediate results for computation of reduce. Informally, for an associative binary operator $\oplus$ and an initial value e, we have

$$\begin{aligned} &\oplus\!\!\not\!\!\not\!\!{}_{e}\ [x_1, x_2, \ldots, x_n] \\ &\quad = \ [e, e \oplus x_1, e \oplus x_1 \oplus x_2, \ldots, \\ &\qquad\qquad e \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n]. \end{aligned}$$

Note that this definition is a little different from that in [Bir87] where e is required to be the unit of $\oplus$. In fact efficient implementation of the scan skeleton does not need this restriction.

Zip is the skeleton that merges two lists into a single one by paring the corresponding elements. The resulting list has the same length as that of shorter one. Informally, we have

$$\begin{aligned} &[x_1, x_2, \ldots, x_n] \curlyvee [y_1, y_2, \ldots, y_n] \\ &\quad = [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]. \end{aligned}$$

It has been shown that these four operators have nice massively parallel implementations on many architectures [Ski90]. If k and $\oplus$ need $O(1)$ parallel time, then $k*$ can be implemented using $O(1)$ parallel time, and both $\oplus/$ and $\oplus\!\!\not\!\!\not_{e}$ can be implemented using $O(\log N)$ parallel time, where N denotes the size of the list. For example, $\oplus/$ can be computed in parallel on a tree-like structure with the combining operator $\oplus$ applied in the nodes, while $k*$ is computed in parallel with k applied to each of the leaves. The study on efficient parallel implementation of $\oplus\!\!\not\!\!\not_{e}$ can be found in [Ble89], though the implementation may be difficult to understand.

III. Segmented Diffusion Theorem

We aims at a general framework for efficient implementation of so-called *segmented computation* [Ble89], i.e., a computation defined in the form of $f*$ where f is described in terms of skeletons. It contains a nested use of skeletons; skeletons are used inside the map skeleton. This kind of form appears quite often in skeletal parallel programming.

Consider, as a simple example, to compute sums of all the rows in a sparse matrix. Note that to reduce the space, a sparse matrix like

$$\begin{pmatrix} 0 & 0 & 3 & 0 & 0 \\ 1 & 4 & 1 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 \end{pmatrix}$$

is often represented by a list of list like

$$\begin{aligned} &[[(2,3)], \\ &\ [(0,1),(1,4),(2,1),(3,5)], \\ &\ [\,], \\ &\ [(0,9)]]. \end{aligned}$$

With this representation, the summation of a matrix can be solved easily using the skeletons by

$$sm \quad = \quad ((+/) \circ (snd*))*.$$

If we implement sm by just following the parallel semantics of the skeletons of *map* and *reduce*, we cannot obtain much parallelism. To see this clear, suppose we are given 4 processors. According to the semantics of *map*, we assign four element lists to each processor. Since the lengths of the lists assigned to each processor are 1,4,0,1, which are of much difference, the unbalanced loads cancel the parallelism.

Blelloch [Ble89][Ble92] gave a *case study*, showing that if f is a scan $\oplus\!\!\#_e$, then $f*$ is called *segmented scan* and can be implemented efficiently. However, it is too restrictive if f must be a scan. We hope to address this problem more generally.

A. Flattening Lists

To remove nested use of skeletons, we should have a flat representation of nested lists. One simple way is to represent nested lists by a flat list with tags indicating the starting element of each inner lists [Ble89]. For instance, the nested list

$$[[1,2],[3,4,5]]$$

may be represented by

$$[(T,1),(F,2),(T,3),(F,4),(F,5)]$$

where the tag T is associated to the first element of each inner list, while the tag F is associated to others. Simple as it is, it cannot represent a nested list with an empty inner list, because the empty list does not have any element, let alone to say the starting one. To resolve this problem, we extend the T flag with an additional number to indicate the number of empty lists appearing before the element. For example, the nested list

$$[[1],[],[],[2,3,4],[],[5]]$$

is flattened into

$$[(T\ 0,1),(T\ 2,2),(F,3),(F,4),(T\ 1,5)],$$

that is,

$$\begin{array}{l} \mathit{flatten}\ [[1],[],[],[2,3,4],[],[5]] \\ \quad = [(T\ 0,1),(T\ 2,2),(F,3),(F,4),(T\ 1,5)]. \end{array}$$

With the flattened representation of nested lists, each processor can be assigned almost the same number of data elements, and therefore, reasonable load balancing between processors can be achieved. For the above example of computation on matrix, the four processors may be assigned to $[(T\ 0,(2,3)),(T\ 0,(0,1))]$, $[(F,(1,4),(F,(2,1)]$, $[(F,(3,5))]$, and $[(T\ 1,(0,9))]$, respectively. Note that the elements of the same inner list may be divided and assigned to more than one processor.

For the sake of simple presentation, we will assume no empty list appears at the end in a nested list.

B. Segmented Diffusion Transformation

To flatten nested skeletons, we shall propose a transformation rule, called *segmented diffusion theorem*. This theorem could be considered as the segmented version of the *diffusion theorem* [HTI99].

B.1 Diffusion Theorem

The diffusion theorem was motivated by the need of a more friendly interface for skeletal parallel programming, making it easy to find a good combination of skeletons to solve complicated problems.

Definition 1 (accumulate) Let g, p, q be functions, and $\oplus$ and $\otimes$ be associative operators. Function accumulate is defined by

$$\begin{array}{l} \mathsf{accumulate}\ [\,]\ e = g\ e \\ \mathsf{accumulate}\ (a:x)\ e = \\ \qquad p(a,e)\ \oplus\ \mathsf{accumulate}\ x\ (e \otimes q\ a). \end{array}$$

We write $[\![g,(p,\oplus),(q,\otimes)]\!]$ for the function accumulate. ■

The *diffusion theorem* [HTI99] shows that the recursive function accumulate can be decomposed into an efficient parallel program in terms of the primitive skeletons *map*, *reduce*, *zip*, and *scan*.

Theorem 1 (Diffusion) The skeleton accumulate can be diffused into the following composition of skeletal functions.

$$\begin{array}{lcl} \mathsf{accumulate}\ x\ e & = & \mathbf{let}\ y \mathbin{+\!\!+} [e'] = \otimes\!\!\#_e(q*x)) \\ & & \qquad z = x \Upsilon y \\ & & \mathbf{in}\ (\oplus/\,(p*z)) \oplus (g\ e') \end{array}$$

■

Therefore, recursive function accumulate abstracts a 'good' combination of parallel primitives. On one hand, recursive functions defined in the form of accumulate can be decomposed into the form using skeletons. On the other hand, as discussed in [HIT02], compositions of primitive skeletons can be fused into accumulate.

B.2 Segmented Diffusion Theorem

Our theorem concerning nested skeletons states that mapping the function accumulate to every sublist can be turned into a form applying another accumulate to the flattened representation of the given nested list.

Theorem 2 (Segmented Diffusion) Let xss be a nested list. Then

$$\begin{array}{l}(\lambda xs.\ [\![g,(p,\oplus),(q,\otimes)]\!]\ xs\ c) * xss \\ \quad = [\![g',(p',\oplus'),(q',\otimes')]\!](\mathit{flatten}\ xs)(F,c,\mathit{True},0)\end{array}$$

where[1]

$$\begin{array}{l}g'(\_,\_,\mathit{True},i) = \mathit{copy}\ i\ (g\ c) \\ g'(\_,a,\mathit{False},i) = g\ a : \mathit{copy}\ i\ (g\ c)\end{array}$$

$$\begin{array}{l}p'((T\ m,x),(\_,\_,True,\_)) \\ \qquad = \mathit{copy}\ m\ (g\ c) +\!\!+ [p\ (x,c)] \\ p'((T\ m,x),(\_,a,False,\_)) \\ \qquad = [g\ a] +\!\!+ \mathit{copy}\ m\ (g\ c) +\!\!+ [p\ (x,c)] \\ p'((F,x),(\_,a,\_,\_)) \\ \qquad = [p\ (x,a)]\end{array}$$

$$\begin{array}{l}us \oplus' vs \\ \qquad = \mathit{init}\ us +\!\!+ [\mathit{last}\ us \oplus \mathit{head}\ vs] +\!\!+ \mathit{tail}\ vs\end{array}$$

$$\begin{array}{l}q'(T\ m,x) = (T\ m, c\otimes q\ x, \mathit{False}, 0) \\ q'(F,x) = (F, q\ x, \mathit{False}, 0)\end{array}$$

$$\begin{array}{l}(\_,\_,\_,i) \otimes' (T\ m,b,v,\_) = (T\ m,b,v,i) \\ (t,a,\_,i) \otimes' (F,b,v,\_) = (t,a\otimes b,v,i)\end{array}$$

■

Since space is limited, we do not give detailed proof here; one can prove it by induction on the length of xss. Two remarks are worth making on the segmented diffusion theorem. First, it follows from the efficient implementation of accumulate that the map of accumulate can be efficiently implemented. Second, since skeletons $f*$, $\oplus/$ and $\oplus \#_e$ are special cases of accumulate, the map of these skeletons can be efficiently implemented.

Corollary 3: Skeletal programs $(f*)*$, $(\oplus/)*$, and $(\oplus\#_e)*$ can be efficiently flattened into an accumulate. ■

IV. Flattening Algorithm

Based on the segmented diffusion theorem, we give the following algorithm, called $\mathcal{F}$, for flattening nested skeletons. The input to the algorithm is nested skeletal programs in the form of $f*$, where f may be a complicated skeletal program. The output is an equivalent but more efficient skeletal parallel programs eliminating this nesting.

$\mathcal{F}(f*)$ =
  let $f' = \mathit{fusion}(f)$
  in
    if f' is not an accumulate
    then $f*$
    else
      let
        $acc' = \mathit{segmened\text{-}diffusion}(f'*)$
        $g = \mathit{diffusion}(acc')$
      in g

The algorithm can be read as follows. Given the program $f*$, $\mathcal{F}$ first tries to apply the fusion transformation [HIT02][GWL99] to merge compositions of skeletons into a single one. If this fusion fails, then the result f' will not be in the form of accumulate, and $\mathcal{F}$ will not change the program and returns the initial one as result. Otherwise, it applies the segmented diffusion theorem to flatten $f'*$ into another accumulate function, which can be transformed back to a skeletal parallel program according to the diffusion theorem.

To be concrete, recall the example in Section III, where we have the following skeletal parallel program:

$$sm \quad = \quad ((+/) \circ (snd*))*.$$

First, we apply fusion transformation to $(+/) \circ (snd*)$ and get

$$f'\ xs = [\![\lambda c.[\,],(\lambda(x,c).snd\ x,+),(id,\lambda c.\lambda a.c)]\!]\ xs\ 0$$

which is in the accumulate form. Then we apply the segmented diffusion theorem to $f'*$ and obtain a new function

$$\begin{array}{l}\lambda xss.\ ([\![g',(p',\oplus'),(q',\otimes')]\!] \\ \qquad (\mathit{flatten}\ xss)\ (F,0,\mathit{True},0))\end{array}$$

where the definitions for g', p', $\oplus'$, q' and $\otimes'$ are omitted. Finally, we apply the diffusion theorem to transform accumulate back into a program in terms of the primitive skeletons, which is omitted here.

As a more interesting example, consider to compute a list of polynomials: Given is a list of lists ass

$$\begin{array}{rcl}ass & = & [[a_{11},a_{12},\ldots,a_{1n_1}], \\ & & \ [a_{21},a_{22},\ldots,a_{1n_2}], \\ & & \qquad\vdots \\ & & \ [a_{m1},a_{22},\ldots,a_{mn_m}]]\end{array}$$

[1] We omit the definition for the functions *copy*, *head*, *last*, *init*, which are standard in Haskell [JH99].

and a constant c, we want to design an efficient skeletal parallel program *polys ass c* to compute

$$
\begin{aligned}
&[a_{11} \times c + a_{12} \times c^2 + \cdots + a_{1n_1} \times c^{n_1}, \\
&\ a_{21} \times c + a_{22} \times c^2 + \cdots + a_{2n_2} \times c^{n_2}, \\
&\qquad\qquad\qquad \vdots \\
&\ a_{m1} \times c + a_{m2} \times c^2 + \cdots + a_{mn_m} \times c^{n_m}]
\end{aligned}
$$

where n_i and n_j may be much different when $i \neq j$.

Clearly, computation of each polynomial is independent, and *polys* can be defined using the map skeleton as

$$polys\ ass\ c = (\lambda as.\, poly\ as\ c) * ass.$$

Here *poly* is to compute a single polynomial, which can be defined in terms of the primitive skeletons as follows [HIT02].

$$
\begin{aligned}
&poly\ as\ c\ = \\
&\quad +/((\lambda(x,y).x \times y) * tail(\oplus\#_{(\_,1)}\ as) \\
&\ \text{where } (\_, e) \oplus\ x = (x, e * c)
\end{aligned}
$$

One may stop here to appreciate the difficulty in flattening *polys*; the function *poly* used inside the map skeleton looks rather complicated. However, we can apply the cheap fusion transformation [HIT02] to obtain the following definition

$$
\begin{aligned}
&poly\ as\ c = \\
&\quad [\![\lambda e.0, (\lambda(a,e).\, a \times e, +), (\lambda x.c, \times)]\!]\ as\ 1.
\end{aligned}
$$

Now one can apply the segmented diffusion theorem followed by the diffusion theorem to obtain a new efficient skeletal parallel program for computing *polys*. We leave the detailed derivation to the readers.

V. Experiment: The Tag Matching Problem

As a more practical example, consider that we are given a document composed of many paragraphs of various lengths. It contains many open and close tags like XML document, and we are interested in checking whether each paragraph is consistent, i.e. open and close tags must perfectly match within each paragraph.

Using a stack, one can easily do checking for one paragraph. Opening tags are pushed, and a closing tags are matched with the current stack top. Failure is indicated by a mismatch, or by a nonempty stack when a match is required or at the end of the scan

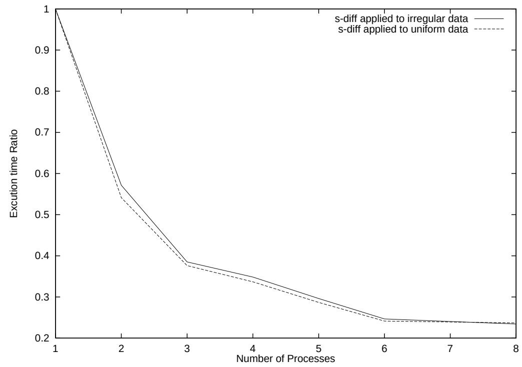

Fig. 1. S-Diff applied to uniform and irregular data

of the input.

$$
\begin{aligned}
check\ [\,]\ s &= isEmpty\ s \\
check\ (a:x)\ s &= \text{if } isOpen\ a \\
&\quad\ \ \text{then } check\ x\ (push\ a\ s) \\
&\quad\ \ \text{elseif } isClose\ a \\
&\quad\ \ \text{then } noEmpty\ s\ \wedge \\
&\qquad\qquad match\ a\ (top\ s) \wedge \\
&\qquad\qquad check\ x\ (pop\ s) \\
&\quad\ \ \text{else } check\ x\ s
\end{aligned}
$$

Here we use several boolean functions; *isOpen* and *isClose* are to determine if an input tag is an opening or an closing one, *match* to determine if two tags are bracket-matched, and *isEmpty* and *noEmpty* to determine if a stack is empty or not.

With *check*, checking whether all paragraphs are matched or not can be defined by

$$tagMatch = (\lambda xs.\, check\ xs\ Empty) * .$$

It has been shown in [HTI99][AIH00] that *check* can be defined in terms of accumulate, so we can use the segmented diffusion theorem.

To take a look at the effectiveness of the segmented diffusion theorem, we conducted some preliminary experiments on the following two programs for *tagMatch*.

- Map-Diff: the skeletal program without applying the flattening algorithm.
- S-Diff: the skeletal program after applying the flattening algorithm.

We use the skeletal library implemented by C language and MPI library [AIH00]. Our machine environment consists of two multiprocessor PCs: one has 4 processors (PentiumIII Xeon 550MHz) and the other has also 4 processors (PentiumIII Xeon 450MHz).

Test data size of the entire document is 5,000,000 characters, and total number of paragraphs is set to 100. Fig. 1 shows some results. The solid line

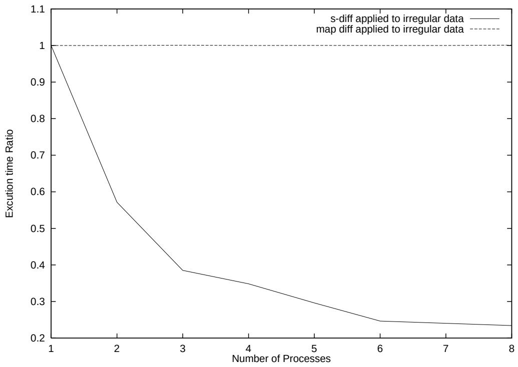

Fig. 2. S-Diff and Map-Diff applied to irregular data

represents the case where the sizes of paragraphs are quite different, and the broken line represents the case where the sizes of paragraphs are almost uniform. We can see that two lines are very similar, showing that S-Diff is sufficiently efficient even though the input (nested) data is irregular.

To compare S-Diff with Map-Diff, we have conducted another experiment. The input data is the same as that used in the experiment of the solid line in Fig. 1, except that the original nested representation is used in this experiment. Equal number of inner lists (paragraphs) in the nested list are distributed to each processor without taking their sizes into account, and they are supplied to *check*. The result is shown by a broken line in Fig. 2. The total computation time does not decrease even though total number of processors increases. This fact suggests that the computation time depends on the paragraph of the largest size. The solid line is the same as that of Figure 1, but we plot it in Fig. 2 for comparison.

These results clearly show the power of the segmented diffusion theorem and effectiveness of the proposed flattening algorithm.

VI. Conclusion

In this paper, we propose a novel theorem called the segmented diffusion theorem, an extension of the existing diffusion theorem, which can efficiently flatten nested skeletons. Based the theorem, we present a flattening theorem, and shows how it can be used to optimized a wide class of nested skeletons. Several examples and experimental results show its power and effectiveness.

Different from the approach in [TIH01], we do not need to introduce a set of nested skeletons. Compared with the flatten transformation in [HIT02], our new flattening transformation based on the segmented diffusion theorem can deal with nest lists that may contains empty inner lists, and furthermore the new theorem is more concise in the sense the result program is a single accumulate (instead of a composition of a project function and accumulate) which is more suitable for further transformation.

References

[AIH00] S. Adachi, H. Iwasaki, and Z. Hu. Diff: A powerful parallel skeleton. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Application*, pages 525–527 (Vol.4), Las Vegas, June 2000. CSREA Press.

[Bir87] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[Ble89] Guy E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[Ble92] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.

[Col89] M. Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[DFH+93] 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 *Parallel Architectures & Languages Europe.* Springer-Verlag, June 93.

[GWL99] Sergei Gorlatch, Christoph Wedler, and Christian Lengauer. Optimization rules for programming with collective operations. In Mikhail Atallah, editor, *IPPS/SPDP'99. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing*, pages 492–499, 1999.

[HIT02] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *11th European Symposium on Programming (ESOP 2002)*, pages 83–97, Grenoble, France, April 2002. Springer Verlag, LNCS 2305.

[HTI99] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[JH99] S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* Available online: `http://www.haskell.org`, February 1999.

[Ski90] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

[Ski94] D.B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[Ski92] D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *NATO ARW "Software for Parallel Computation"*, June 92.

[TIH01] T. Takahashi, H. Iwasaki, and Z. Hu. Efficient parallel skeletons for nested data structures. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Application*, Las Vegas, June 2001. CSREA Press.