Powerlist: A Structure for Parallel Recursion

JAYADEV MISRA
The University of Texas

Many data-parallel algorithms—Fast Fourier Transform, Batcher's sorting schemes, and the prefix-sum—exhibit recursive structure. We propose a data structure called *powerlist* that permits succinct descriptions of such algorithms, highlighting the roles of both parallelism and recursion. Simple algebraic properties of this data structure can be exploited to derive properties of these algorithms and to establish equivalence of different algorithms that solve the same problem.

Categories and Subject Descriptors: D3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; recursion; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms: Algorithms

Additional Key Words and Phrases: Algebra of parallel programs, Batcher sort, Fast Fourier Transform, hypercube, recursion, parallel programs, prefix sum

1. PARALLELISM AND RECURSION

Many important synchronous parallel algorithms—Fast Fourier Transform, routing and permutation, Batcher sorting schemes, solving tridiagonal linear systems by odd-even reduction, and prefix-sum algorithms—are conveniently formulated in a recursive fashion. Network structures on which parallel algorithms are typically implemented—butterfly, sorting networks, hypercube, and the complete binary tree—are also recursive in nature. However, parallel recursive algorithms are typically described iteratively, one parallel step at a time.[1] Similarly, the connection structures are often explained pictorially by displaying the connections between one "level" and the next. The mathematical properties of the algorithms and connection structures are rarely evident from these descriptions.

[1]A notable exception is the recursive description of a prefix-sum algorithm in Karp and Ramachandran [1990].

This material is based in part on work supported by the Texas Advanced Research Program under grant no. 003658-219 and by the National Science Foundation Award CCR-9111912. A preliminary version of this article appeared in Misra [1994].
Author's address: Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712; email: misra@cs.utexas.edu.

We propose in this article a data structure called *powerlist* that highlights the role of both parallelism and recursion. Many of the known parallel algorithms such as FFT, the Batcher Merge, prefix-sum, and embedding arrays in hypercubes, etc. have surprisingly concise descriptions using powerlists. Simple algebraic properties of powerlists permit us to deduce properties of these algorithms that employ structural induction.

2. POWERLIST

The basic data structure on which recursion is employed in LISP [McCarthy et al. 1962] or ML [Milner et al. 1990] is a *list*. A list is either empty or is constructed by concatenating an element to a list. In this article, we restrict ourselves to finite lists. We call such a list *linear* because the list length grows by 1 as a result of applying the basic constructor. Such a list structure seems unsuitable for expressing parallel algorithms succinctly; an algorithm that processes the linear-list elements has to describe how successive elements of the list are processed.

We propose *powerlist* as a data structure that is more suitable for describing parallel algorithms. The base, which corresponds to the empty list for the linear case, is a list of one element. A longer powerlist is constructed from the elements of two powerlists of the same length (described later). Thus, a powerlist is multiplicative in nature as its length doubles by applying the basic constructor.

There are two different ways in which powerlists are joined to create a longer powerlist. If p, q are powerlists of the same length, then

$p \mid q$ is the powerlist formed by concatenating p and q, and

$p \bowtie q$ is the powerlist formed by successively taking alternate items from p and q, starting with p.

Furthermore, we restrict p, q to contain similar elements as defined in Section 2.1. In our examples, the sequence of elements of a powerlist is enclosed within angular brackets, while for notational convenience linear lists will be enclosed in square brackets.

$$\langle 0\rangle \mid \langle 1\rangle = \langle 0 \quad 1\rangle$$
$$\langle 0\rangle \bowtie \langle 1\rangle = \langle 0 \quad 1\rangle$$
$$\langle 0 \quad 1\rangle \mid \langle 2 \quad 3\rangle = \langle 0 \quad 1 \quad 2 \quad 3\rangle$$
$$\langle 0 \quad 1\rangle \bowtie \langle 2 \quad 3\rangle = \langle 0 \quad 2 \quad 1 \quad 3\rangle$$

The operation $\mid$ is called *tie*, and $\bowtie$ is *zip*.

2.1 Definitions

A data item from the linear list theory will be called a *scalar*. Typical scalars are the items of base types—integer, boolean, etc.—tuples of scalars, functions from scalars to scalars, and linear lists of scalars. However, scalars are uninterpreted in our theory, as we assume merely that scalars can be checked for type compatibility. We will use several standard operations on scalars for illustration purposes.

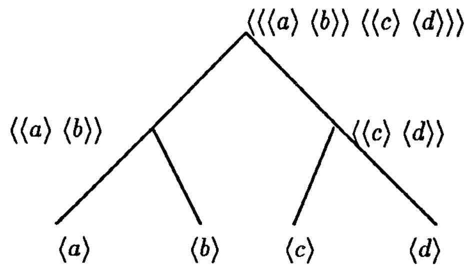


Fig. 1. Representation of a complete binary tree where the data are at the leaves. For leaf nodes, the powerlist has one element. For nonleaf nodes, the powerlist has two elements, namely, the powerlists for the left and right subtrees.

A *powerlist* is a list of length 2^n, for some n, $n \geq 0$, all of whose elements are similar.

Two scalars are *similar* if they are of the same type; two powerlists are *similar* if they are equal in length and if any element of one is similar to any element of the other. Note that *similar* is an equivalence relation.

Let S denote an arbitrary scalar, P a powerlist, and u, v similar powerlists. A recursive definition of a powerlist is the following:

$$\langle S \rangle \text{ or } \langle P \rangle \text{ or } u \mid v \text{ or } u \bowtie v.$$

2.1.1 *Examples*

$\langle 2 \rangle$ is a powerlist of length 1 containing a scalar.

$\langle\langle 2 \rangle\rangle$ is a powerlist of length 1 containing a powerlist of length 1 of scalar.

$\langle\ \rangle$ is not a powerlist.

$\langle[\ ]\rangle$ is a powerlist of length 1 containing the empty linear list.

$\langle\langle[2]\ [3\ 4\ 7]\rangle\langle[4]\ [\ ]\rangle\rangle$ is a powerlist of length 2, each element of which is a powerlist of length 2, whose elements are linear lists of numbers.

$\langle\langle 0\ 4\rangle\langle 1\ 5\rangle\langle 2\ 6\rangle\langle 3\ 7\rangle\rangle$ is a representation of the matrix

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$$

where each column is an element of the outer powerlist.

$\langle\langle 0\ 1\ 2\ 3\rangle\langle 4\ 5\ 6\ 7\rangle\rangle$ is another representation of the above matrix where each row is an element of the outer powerlist.

$\langle\langle\langle a\rangle\ \langle b\rangle\rangle\ \langle\langle c\rangle\ \langle d\rangle\rangle\rangle$ is a representation of the tree in Figure 1. The powerlist contains two elements, one each for the left and right subtrees.

2.2 Functions over Powerlists

We write function application without parentheses where no confusion is possible. Thus, we write "fx" instead of "$f(x)$" and "gxy" instead of "$g(x, y)$." The constructors $\mid$ and $\bowtie$ have the same binding power, and their binding power is lower than that of function application. Throughout this article, S denotes a scalar, P a powerlist, and x, y either scalar or powerlist. Typical names for powerlist variables are p, q, r, s, t, u, and v.

Functions over linear lists are typically defined by case analysis; a function is defined over the empty list and, recursively, over nonempty lists. Functions over powerlists are defined analogously. For instance, the following function *rev* reverses the order of the elements of the argument powerlist:

$$rev\langle x \rangle = \langle x \rangle$$
$$rev(p \mid q) = (rev\, q) \mid (rev\, p).$$

The case analysis, as for linear lists, is based on the length of the argument powerlist. We adopt the pattern-matching scheme of ML [Milner et al. 1990] and Miranda [Turner 1986][2] to *deconstruct* the argument list into its components, p and q, in the recursive case. In general, deconstruction uses the operators $\mid$ and $\bowtie$; see Section 3. In the definition of *rev*, we have used $\mid$ for deconstruction, but we could have used $\bowtie$ instead, defining *rev* in the recursive case by

$$rev(p \bowtie q) = (rev\, q) \bowtie (rev\, p).$$

It can be shown by using the laws in Section 3 that the two proposed definitions of *rev* are equivalent and that we have

$$rev(rev\, P) = P$$

for any powerlist P.

2.2.1 *Scalar Functions.* Operations on scalars are outside our theory. Some of the examples in this article, however, use scalar functions, particularly, addition and multiplication over complex numbers and *cons* over linear lists. A scalar function f has zero or more scalars as arguments, and its value is a scalar. We coerce the application of f to a powerlist by applying f "pointwise" to the elements of the powerlist. A scalar function f of one argument is defined as follows:

$$f\langle x \rangle = \langle f x \rangle$$
$$f(p \mid q) = (fp) \mid (fq).$$

Also, it can be shown that

$$f(p \bowtie q) = (fp) \bowtie (fq).$$

A scalar function that operates on two arguments will often be written as an infix operator. For any such function $\oplus$ and similar powerlists p, q, u, v, we have

$$\langle x \rangle \oplus \langle y \rangle = \langle x \oplus y \rangle$$
$$(p \mid q) \oplus (u \mid v) = (p \oplus u) \mid (q \oplus v)$$
$$(p \bowtie q) \oplus (u \bowtie v) = (p \oplus u) \bowtie (q \oplus v).$$

Thus, scalar functions commute with both $\mid$ and $\bowtie$.

It is important to note that because a scalar function is applied recursively to each element of a powerlist, its effect propagates through all "levels." Thus, $+$ applied to matrices forms their elementwise sum.

[2] Miranda is a trademark of Research Software Ltd.

2.3 Discussion

The base case of a powerlist is a singleton list, not an empty list. Empty lists, or equivalent data structures, do not arise in the applications we have considered. For instance, in matrix algorithms, the base case is a 1×1 matrix rather than an empty matrix. Similarly, the Fourier transform is defined for a singleton list rather than an empty list, and the smallest hypercube has one node.

The recursive definition of a powerlist says that a powerlist is either of the form $u \bowtie v$ or $u \mid v$. In fact, every nonsingleton powerlist can be written in either form in a unique manner (see Section 3). A simple way to view $p \mid q = L$ is that if the elements of L are indexed by n-bit strings in increasing numerical order, where the length of L is 2^n, then p is the sublist of elements whose highest bit of the index is 0, and q is the sublist with 1 in the highest bit of the index. Similarly, if $u \bowtie v = L$, then u is the sublist of elements whose lowest bit of the index is 0, and v's elements have 1 as the lowest bit of the index.

At first, it may seem strange to allow two different ways for constructing the same list by using *tie* or *zip*. But as we see in this article, this causes no difficulty, and furthermore, this flexibility is essential because many parallel algorithms—the Fast Fourier Transform being most prominent—exploit both forms of construction

We have restricted u, v in $u \mid v$ and $u \bowtie v$ to be similar. This restriction allows us to process a powerlist by recursive divide and conquer, where each division yields two halves that can be processed in parallel by employing the same algorithm. (Square matrices, for instance, are often processed by quartering them. We will show how quartering, or quadrupling, can also be expressed in our theory.) The similarity restriction allows us to define complete binary trees, hypercubes, and square matrices that are not "free" structures. And, though the length of a powerlist is a power of 2, which somewhat restricts our theory, it is possible to design a more general theory eliminating this constraint. We sketch an outline of it in Section 6.

3. LAWS

L0. For singleton powerlists, $\langle x \rangle, \langle y \rangle$ we have

$$\langle x \rangle \mid \langle y \rangle = \langle x \rangle \bowtie \langle y \rangle.$$

L1. Dual Deconstruction. For any nonsingleton powerlist P there exist similar powerlists r, s, u, v such that

$$P = r \mid s \quad \text{and} \quad P = u \bowtie v.$$

L2. Unique Deconstruction

$$(\langle x \rangle = \langle y \rangle) \equiv (x = y)$$
$$(p \mid q = u \mid v) \equiv (p = u \wedge q = v)$$
$$(p \bowtie q = u \bowtie v) \equiv (p = u \wedge q = v).$$

L3. Commutativity of $\mid$ and $\bowtie$. $(p \mid q) \bowtie (u \mid v) = (p \bowtie u) \mid (q \bowtie v)$.

These laws can be derived by defining *tie* and *zip* suitably, using the standard functions from the linear list theory. One possible strategy is to define tie as the concatenation of two equal length lists, then to use Laws L0 and L3 as the definition of *zip*; Laws L1 and L2 can be derived next. Alternatively, these laws may be regarded as axioms relating *tie* and *zip*.

Law L0 is often used in proving base cases of algebraic identities. Laws L1 and L2 allow us to deconstruct uniquely a nonsingleton powerlist using either $|$ or $\bowtie$. Law L3 is crucial. It is the only law relating the two construction operators $|$ and $\bowtie$ in the general case. Hence, it is invariably applied in proofs by structural induction where both constructors play a role.

3.1 Inductive Proofs

Most proofs on powerlists are by induction on the length, depth, or shape of the list. The length *len* of a powerlist is the number of elements in it. Since the length of a powerlist is a power of 2, the logarithmic length *lgl* is a more useful measure. Formally,

$$lgl\langle x \rangle = 0$$
$$lgl(u \mid v) = 1 + (lgl\, u).$$

The *depth* of a powerlist is the number of "levels" in it:

$$depth\langle S \rangle = 0$$
$$depth\langle P \rangle = 1 + (depth\, P)$$
$$depth(u \mid v) = depth\, u.$$

In the last case, since u and v are similar powerlists, they have the same depth. Most inductive proofs on powerlists order them lexicographically on the pair (depth, logarithmic length). For instance, to prove that a property Π holds for all powerlists, it is sufficient to prove that

$\Pi\langle S \rangle$,
$\Pi\, P \Rightarrow \Pi\langle P \rangle$, and
$(\Pi\, u) \wedge (\Pi\, v) \wedge (u, v)$ similar $\Rightarrow \Pi(u \mid v)$.

The last proof step could be replaced by

$$(\Pi\, u) \wedge (\Pi\, v) \wedge (u,v) \text{ similar} \Rightarrow \Pi(u \bowtie v).$$

The *shape* of a powerlist P is a sequence of natural numbers $n_0, n_1, \ldots, n_d$ where d is the depth of P and

n_0 is the logarithmic length of P,
n_1 is the logarithmic length of (any) element of P, say r, and
n_2 is the logarithmic length of any element of $r, \ldots$

A formal definition of shape is similar to that of depth. The shape is a linear sequence because all elements, at any level, are similar. The shape and the type of the scalar elements define the structure of a powerlist completely. For inductive proofs, the powerlists may be ordered lexicographically by the pair (depth, shape), where the shapes are compared lexicographically.

Example. The *len*, *lgl*, and *depth* of $\langle\langle 0\ 1\ 2\ 3\rangle\langle 4\ 5\ 6\ 7\rangle\rangle$ are 2, 1, and 1, respectively. The *shape* of this powerlist is the sequence 1 2, because there are 2 elements at the outer level and 4 elements at the inner level.

4. EXAMPLES

Here, we show a few small algorithms on powerlists. These include such well-known examples as the Fast Fourier Transform and Batcher sorting schemes. We restrict the discussion in this section to simple or unnested powerlists where the depth is 0. Higher-dimensional lists and algorithms for matrices and hypercubes are taken up in a later section. Since the powerlists are unnested, induction based on length is sufficient to prove properties of these algorithms.

4.1 Permutations

We define a few functions that permute the elements of powerlists. The function *rev* defined in Section 2.2 is a permutation function. These functions appear as components of many parallel algorithms.

4.1.1 *Rotate.* Function *rr* rotates a powerlist to the right by one; thus, $rr\langle a\ b\ c\ d\rangle = \langle d\ a\ b\ c\rangle$. Function *rl* rotates to the left: $rl\langle a\ b\ c\ d\rangle = \langle b\ c\ d\ a\rangle$.

$$rr\langle x\rangle = \langle x\rangle, \qquad rl\langle x\rangle = \langle x\rangle$$
$$rr(u \bowtie v) = (rr\ v) \bowtie u, \quad rl(u \bowtie v) = v \bowtie (rl\, u).$$

There does not seem to be any simple definition of *rr* or *rl* using $|$ as the deconstruction operator. It is easy to show through structural induction that *rr* and *rl* are inverses. An amusing identity is $rev(rr(rev(rr\,P))) = P$.

A powerlist may be rotated through an arbitrary amount k by applying k successive rotations. A better scheme for rotating $(u \bowtie v)$ by k is to rotate both u and v by about $k/2$. More precisely, the function *grr* given below rotates a powerlist to the right by k, where $k \geq 0$. It is straightforward to show that for all k, $k \geq 0$, and that for all p, $(grr\,k\,p) = (rr^{(k)}\,p)$, where $rr^{(k)}$ is the k-fold application of *rr*:

$$grr\,k\langle x\rangle = \langle x\rangle$$
$$grr(2\times k)(u \bowtie v) = (grr\,k\,u) \bowtie (grr\,k\,v)$$
$$grr(2\times k + 1)(u \bowtie v) = (grr\,(k+1)\,v) \bowtie (grr\,k\,u).$$

4.1.2 *Rotate Index.* A class of permutation functions can be defined by the transformations on the element indices. For a powerlist of 2^n elements, we associate an n-bit index with each element, where the indices are the binary representations of $0, 1, \ldots, 2^n - 1$ in sequence. For a powerlist $u\,|\,v$, indices for the elements in u have 0 as the highest bit, and elements in v have 1 as the highest bit. In $u \bowtie v$, similar remarks apply for the lowest bit. Any bijection h mapping indices to indices defines a permutation of the powerlist: the element with index i is moved to the position where it has index $(h\,i)$. Next, we consider two simple index mapping functions; the corresponding

P's indices	=	(000	001	010	011	100	101	110	111)
List P	=	$\langle a$	b	c	d	e	f	g	$h\rangle$
P's indices rotated right	=	(000	100	001	101	010	110	011	111)
$rs\ P$	=	$\langle a$	c	e	g	b	d	f	$h\rangle$
P's indices rotated left	=	(000	010	100	110	001	011	101	111)
$ls\ P$	=	$\langle a$	e	b	f	c	g	d	$h\rangle$

Fig 2. Permutation functions rs, rl defined in Section 4.2.2.

permutations of powerlists are useful in describing the shuffle-exchange network. Note that indices are not part of our theory.

A function that rotates an index to the right by one position has the permutation function for *r*ight *s*huffle, or *rs*, associated with it. The definition of *rs* may be understood as follows. The effect of rotating an index to the right is that the lowest bit of an index becomes the highest bit; therefore, if *rs* is applied to $u \bowtie v$, the elements of u—those having 0 as the lowest bit—will occupy the first half of the resulting powerlist because their indices have 0 as the highest bit after rotation; similarly, v will occupy the second half. Figure 2 shows the effects of index rotations on an 8-element list. Analogously, the function that rotates an index to the left by one position induces the permutation defined by *l*eft *s*huffle, or *ls*, as shown below:

$$\begin{aligned} rs\langle x \rangle &= \langle x \rangle, & ls\langle x \rangle &= \langle x \rangle \\ rs(u \bowtie v) &= u \mid v, & ls(u \mid v) &= u \bowtie v. \end{aligned}$$

It is trivial to see that *rs* and *ls* are inverses.

4.1.3 *Inversion.* The function *inv* is defined by the following function on indices: an element with index b in P has index b' in $(inv\, P)$, where b' is the reversal of the bit string b. Thus,

$$\begin{array}{rccccccccl} & 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 & \\ inv\langle & a & b & c & d & e & f & g & h & \rangle = \\ \langle & a & e & c & g & b & f & d & h & \rangle. \end{array}$$

The definition of *inv* is:

$$\begin{aligned} inv\langle x \rangle &= \langle x \rangle \\ inv(p \mid q) &= (inv\, p) \bowtie (inv\, q). \end{aligned}$$

This function arises in a variety of contexts. In particular, *inv* is used to permute the output of a Fast Fourier Transform network into the correct order.

The following proof shows a typical application of structural induction:

*INV*1. $inv(p \bowtie q) = (inv\, p) \mid (inv\, q)$.

Proof is by structural induction on p and q.

Base.

$inv(\langle x \rangle \bowtie \langle y \rangle)$
$= \{\text{From Law L0: } \langle x \rangle \bowtie \langle y \rangle = \langle x \rangle \mid \langle y \rangle\}$
$inv(\langle x \rangle \mid \langle y \rangle)$
$= \{\text{definition of } inv\}$
$inv\langle x \rangle \bowtie inv\langle y \rangle$
$= \{inv\langle x \rangle = \langle x \rangle, inv\langle y \rangle = \langle y \rangle.$ Thus, they are singletons. Applying L0$\}$
$inv\langle x \rangle \mid inv\langle y \rangle$

Induction.

$inv((r \mid s) \bowtie (u \mid v))$
$= \{\text{commutativity of } \mid, \bowtie\}$
$inv((r \bowtie u) \mid (s \bowtie v))$
$= \{\text{definition of } inv\}$
$inv(r \bowtie u) \bowtie inv(s \bowtie v)$
$= \{\text{induction}\}$
$(inv\, r \mid inv\, u) \bowtie (inv\, s \mid inv\, v)$
$= \{\mid, \bowtie \text{ commute}\}$
$(inv\, r \bowtie inv\, s) \mid (inv\, u \bowtie inv\, v)$
$= \{\text{apply definition of } inv \text{ to both sides of } \mid\}$
$inv(r \mid s) \mid inv(u \mid v).$

Using INV1 and structural induction, it is easy to establish that

$$inv(inv\, P) = P,$$
$$inv(rev\, P) = rev(inv\, P),$$

and for any scalar operator $\oplus$,

$$inv(P \oplus Q) = (inv\, P) \oplus (inv\, Q).$$

The last result holds for any permutation function in place of *inv*.

4.2 Reduction

In the linear list theory [Bird 1989], reduction is a higher-order function of two arguments, an associative binary operator, and a list. Reduction applied to $\oplus$ and $[a_0 a_1 \dots a_n]$ yields $(a_0 \oplus a_1 \oplus \cdots \oplus a_n)$. This function over powerlists is defined by the following:

$$red \oplus \langle x \rangle = x$$
$$red \oplus (p \mid q) = (red \oplus p) \oplus (red \oplus q).$$

4.3 Gray Code

Gray code sequence [Gray 1953] for n, when $n \geq 0$, is a sequence of 2^n n-bit strings, where the consecutive strings in the sequence differ in exactly one bit

$n = 0$	⟨[ ]⟩
$n = 1$	⟨[0] [1]⟩
$n = 2$	⟨[00] [01] [11] [10]⟩
$n = 3$	⟨[000] [001] [011] [010] [110] [111] [101] [100]⟩

Fig. 3. Standard Gray code sequence for n, $n = 0, 1, 2, 3$.

position. The last and the first strings in the sequence are considered consecutive. Standard Gray code sequences for $n = 0, 1, 2, 3$ are shown in Figure 3. We represent the n-bit strings by linear lists of length n and a Gray code sequence by a powerlist whose elements are the linear lists. The standard Gray code sequence may be computed by function G, for any n:

$$G\,0 = \langle[\,]\rangle$$
$$G(n+1) = (0:P) \mid (1:(rev\,P))$$
$$\text{where} \quad P = (G\,n).$$

Here, $(0:)$ is a scalar function that takes a linear list as an argument and appends 0 as its prefix. According to the coercion rule, $(0:P)$ is the powerlist obtained by prefixing every element of P by 0. Similarly, $(1:(rev\,P))$ is defined, where the function rev is from Section 2.2.

4.4 Polynomial

A polynomial with coefficients p_j, $0 \le j < 2^n$, where $n \ge 0$, may be represented by a powerlist p whose jth element is p_j. The polynomial value at some point ω is $\Sigma_{0 \le j < 2^n}\, p_j \times \omega^j$. For $n > 0$ this quantity is

$$\sum_{0 \le j < 2^{n-1}} p_{2j} \times \omega^{2j} + \sum_{0 \le j < 2^{n-1}} p_{2j+1} \times \omega^{2j+1}.$$

The following function ep evaluates a polynomial p using this strategy. In anticipation of the Fast Fourier Transform, we generalize ep to accept an arbitrary powerlist as its second argument. For powerlists p and w, which have possibly unequal lengths, let $(p\;ep\;w)$ be a powerlist of the same length as w, obtained by evaluating p at each element of w:

$$\langle x \rangle\; ep\; \langle w \rangle = \langle x \rangle$$
$$p\; ep\; (u \mid v) = (p\; ep\; u) \mid (p\; ep\; v)$$
$$(p \bowtie q)\; ep\; w = (p\; ep\; w^2) + (w \times (q\; ep\; w^2)).$$

Note that w^2 is the pointwise squaring of w. Also, note that ep is a pointwise function in its second argument.

4.5 Fast Fourier Transform

For a polynomial p with complex coefficients, its Fourier transform is obtained by evaluating p at a sequence (i.e., powerlist) of points (Wp). Here, (Wp) is the powerlist $\langle \omega^0, \omega^1, \ldots, \omega^{n-1} \rangle$, where n is the length of p and ω is the nth principal root of 1. Note that (Wp) depends only on the length of p

but not on its elements, so for similar powerlists, p, q, $(Wp) = (Wq)$. It is easy to define the function W in a manner similar to *ep*.

We need the following properties of W for the derivation of *FFT*. Equation (1) follows from the definition of W and the fact that $\omega^{2/N} = 1$, where N is the length of p (and q). The second equation says that the right half of $W(p \bowtie q)$ is the negative of its left half. This is because each element in the right half is the same as the corresponding element in the left half multiplied by ω^N; since ω is the $(2 \times N)$th root of 1, $\omega^N = -1$.

$$W^2(p \bowtie q) = (Wp)|(Wq) \qquad (1)$$

$$W(p \bowtie q) = u|(-u), \quad \text{for some } u \qquad (2)$$

The Fourier transform, *FT*, of a powerlist p is a powerlist of the same length as p, given by

$$FT\,p = p\,ep\,(Wp)$$

where *ep* is the function defined in Section 4.4.

The straightforward computation of $(p\,ep\,v)$ for any p, v consists of evaluating p at each element of v; this takes time $O(N^2)$ where p, v have length N. Since (Wp) is of a special form, the Fourier transform can be computed in $O(N \log N)$ steps, using the Fast Fourier Transform algorithm [Cooley and Tukey 1965]. This algorithm also admits an efficient parallel implementation, requiring $O(\log N)$ steps on $O(N)$ processors. We derive the FFT algorithm next.

$FT\langle x \rangle$
$=$ {definition of FT}
$x\,ep\,(W\langle x \rangle)$
$=$ {Since $W\langle x \rangle$ is a singleton, from the definition of *ep*}
$\langle x \rangle$

For the general case,

$FT(p \bowtie q)$
$=$ {From the definition of FT}
$(p \bowtie q)\,ep\,W(p \bowtie q)$
$=$ {from the definition of *ep*}
$p\,ep\,W^2(p \bowtie q) + W(p \bowtie q) \times (q\,ep\,W^2(p \bowtie q))$
$=$ {from the property of W; see Eq. (1)}
$p\,ep\,((Wp)|(Wq)) + W(p \bowtie q) \times (q\,ep\,((Wp)|(Wq)))$
$=$ {distribute each *ep* over its second argument}
$((p\,ep\,(Wp))|(p\,ep\,(Wq))) + W(p \bowtie q) \times ((q\,ep\,(Wp))|(q\,ep\,(Wq)))$
$=$ {$(Wp) = (Wq)$, $p\,ep\,(Wp) = FT\,p$, $q\,ep\,(Wq) = FT\,q$}
$((FT\,p)|(FT\,p)) + W(p \bowtie q) \times ((FT\,q)|(FT\,q))$
$=$ {using P, Q for $FT\,p$, $FT\,q$, and $u|(-u)$ for $W(p \bowtie q)$; see Eq. (2)}

$$
\begin{array}{ll}
& (P \mid P) + (u \mid -u) \times (Q \mid Q) \\
= & \{\mid \text{ and } \times \text{ in the second term commute}\} \\
& (P \mid P) + ((u \times Q) \mid (-u \times Q)) \\
= & \{\mid \text{ and } + \text{ commute}\} \\
& (P + u \times Q) \mid (P - u \times Q).
\end{array}
$$

We collect the two equations for FT to define FFT, the Fast Fourier Transform. In the following, $(powers\,p)$ is the powerlist $\langle \omega^0, \omega^1, \ldots, \omega^{N-1} \rangle$ where N is the length of p and ω the $(2 \times N)$th principal root of 1. This was the value of u in the previous paragraph. The function *powers* can be defined similarly to *ep*:

$$
\begin{array}{l}
FFT\langle x \rangle = \langle x \rangle \\
FFT(p \bowtie q) = (P + u \times Q) \mid (P - u \times Q) \\
\quad \text{where} \\
\qquad P = FFT\,p \\
\qquad Q = FFT\,q \\
\qquad u = powers\,p.
\end{array}
$$

It is clear that $FFT(p \bowtie q)$ can be computed from $(FFT\,p)$ and $(FFT\,q)$ in $O(N)$ sequential steps or $O(1)$ parallel steps, using $O(N)$ processors (u can be computed in parallel), where N is the length of p. Therefore, $FFT(p \bowtie q)$ can be computed in $O(N \log N)$ sequential steps or, $O(\log N)$ parallel steps, using $O(N)$ processors.

The compactness of this description of FFT is in striking contrast to the usual descriptions; for an example, see Chandy and Misra [1988, Section 6.13]. The compactness can be attributed to the use of recursion and the avoidance of explicit indexing of the elements by employing $\mid$ and $\bowtie$. FFT illustrates the need for including both $\mid$ and $\bowtie$ as constructors for powerlists. Another function that employs both $\mid$ and $\bowtie$ is *inv* of Section 4.1.

4.5.1 *Inverse Fourier Transform.* The inverse of the Fourier transform, IFT, can be defined similarly to the FFT. We derive the definition of IFT from that of the FFT by pattern matching.

For a singleton powerlist, $\langle x \rangle$, we compute the following:

$$
\begin{array}{ll}
& IFT\langle x \rangle \\
= & \{\langle x \rangle = FFT\langle x \rangle\} \\
& IFT(FFT\langle x \rangle) \\
= & \{IFT, FFT \text{ are inverses}\} \\
& \langle x \rangle.
\end{array}
$$

For the general case, we have to compute $IFT(r \mid s)$ given r and s. Let

$$IFT(r \mid s) = p \bowtie q$$

in the unknowns p and q. We chose this form of deconstruction so that we can easily solve the equations we next generate. Taking *FFT* of both sides,

$$FFT(IFT(r \mid s)) = FFT(p \bowtie q).$$

The left side is $(r \mid s)$ because *IFT* and *FFT* are inverses. Replacing the right side by the definition of $FFT(p \bowtie q)$ yields the following equations:

$$\begin{aligned} r \mid s &= (P + u \times Q) \mid (P - u \times Q), \\ P &= FFT\,p \\ Q &= FFT\,q \\ u &= powers\,p. \end{aligned}$$

These equations are easily solved for the unknowns P, Q, u, p, q. The law of unique deconstruction, L2, can be used to deduce from the first equation that $r = P + u \times Q$ and $s = P - u \times Q$. Also, since p and r are of the same length, we may define u using r instead of p. The solutions of these equations yield the following definition for *IFT*. Here, /2 divides each element of the given powerlist by 2:

$$\begin{aligned} IFT\langle x \rangle &= \langle x \rangle \\ IFT(r \mid s) &= p \bowtie q \end{aligned}$$

where

$$\begin{aligned} P &= (r + s)/2 \\ u &= powers\,r \\ Q &= ((r - s)/2)/u \\ p &= IFT\,P \\ q &= IFT\,Q. \end{aligned}$$

As in the *FFT*, the definition of *IFT* includes both constructors | and $\bowtie$. It can be implemented efficiently on a butterfly network. The complexity of *IFT* is the same as that of the *FFT*.

4.6 Batcher Sort

In this section, we develop some elementary results about sorting and discuss two remarkable sorting methods developed by Batcher [1968]. We find it interesting that $\bowtie$, not |, is the preferred operator in discussing the principles of parallel sorting. Henceforth, a list is *sorted* means that its elements are arranged in a nondecreasing order. A general method of sorting is given by

$$\begin{aligned} sort\langle x \rangle &= \langle x \rangle \\ sort(p \bowtie q) &= (sort\,p)\ merge\ (sort\,q) \end{aligned}$$

where *merge*, written as a binary infix operator, creates a single sorted powerlist out of the elements of its two argument powerlists, each of which is sorted. In this section, we show two different methods for implementing *merge*. One scheme is the Batcher merge, given by the operator *bm*. Another scheme is given by *bitonic* sort where the sorted lists u and v are merged by applying the function bi to $(u \mid (rev\, v))$.

A comparison operator $\updownarrow$ is used in these algorithms. The operator is applied to a pair of equal length powerlists p, q; it creates a single powerlist out of the elements of p, q by

$$p \updownarrow q = (p \min q) \bowtie (p \max q).$$

That is, the $2i$th and $(2i + 1)$th items of $p \updownarrow q$ are $(p_i \min q_i)$ and $(p_i \max q_i)$, respectively. The powerlist $p \updownarrow q$ can be computed in constant time using $O(len\, p)$ processors.

4.6.1 *Bitonic Sort.* A sequence of numbers, $x_0, x_1, \ldots, x_i, \ldots, x_N$, is *bitonic* if there is an index i, $0 \le i \le N$, such that $x_0, x_1, \ldots, x_i$ is monotonic (ascending or descending) and $x_i, \ldots, x_N$ is monotonic. The function bi given below applied to a bitonic powerlist returns a sorted powerlist of the original items:

$$bi\langle x \rangle = \langle x \rangle$$
$$bi(p \bowtie q) = (bi\, p) \updownarrow (bi\, q).$$

For sorted powerlists u and v, the powerlist $(u \mid (rev\, v))$ is bitonic; thus u and v can be merged by applying bi to $(u \mid (rev\, v))$. The form of the recursive definition suggests that bi can be implemented on $O(N)$ processors in $O(\log N)$ parallel steps, where N is the length of the argument powerlist.

4.6.2 *Batcher Merge.* Batcher also proposed a scheme for merging two sorted lists. We define this scheme, bm, as an infix operator below:

$$\langle x \rangle\ bm\ \langle y \rangle = \langle x \rangle \updownarrow \langle y \rangle$$
$$(r \bowtie s)\ bm\ (u \bowtie v) = (r\, bm\, v) \updownarrow (s\, bm\, u).$$

The function bm is well suited for parallel implementation. The recursive form suggests that $(r\, bm\, v)$ and $(s\, bm\, u)$ can be computed in parallel. Since $\updownarrow$ can be applied in $O(1)$ parallel steps using $O(N)$ processors, where N is the length of the argument powerlists, the function bm can be evaluated in $O(\log N)$ parallel steps. In the rest of this section, we develop certain elementary facts about sorting and prove the correctness of bi and bm.

4.6.3 *Elementary Facts about Sorting.* We consider only "compare-and-swap" sorting methods. It is known [Knuth 1973] that such a sorting scheme is correct if and only if it sorts lists containing 0s and 1s only. Therefore, we restrict our discussion to powerlists containing 0s and 1s only.

For a powerlist p, let $(z\, p)$ be the number of 0s in it. To simplify notation, we omit the space and write zp. Clearly,

A0. $z(p \bowtie q) = zp + zq$ and $z\langle x \rangle$ is either 0 or 1.

Powerlists containing only 0s and 1s have the following properties:

A1. $\langle x \rangle$ sorted and $\langle x \rangle$ bitonic.
A2. $(p \bowtie q)$ sorted $\equiv p$ sorted $\wedge$ q sorted $\wedge$ $0 \le zp - zq \le 1$.

A3. $(p \bowtie q)$ bitonic $\Rightarrow p$ bitonic $\wedge\ q$ bitonic $\wedge\ |zp - zq| \le 1$.

Note that the condition analogous to (A2) under which $p \mid q$ is sorted is:

A2′. $(p \mid q)$ sorted $\equiv p$ sorted $\wedge\ q$ sorted $\wedge\ (zp < (len\,p) \Rightarrow zq = 0)$.

The simplicity of A2 compared with A2′, may suggest why $\bowtie$ is the primary operator in parallel sorting.

The following results, B1 and B2, are easy to prove. We prove B3.

B1. p sorted, q sorted, $zp \ge zq \Rightarrow (p \min q) = p \wedge (p \max q) = q$
B2. $z(p \updownarrow q) = zp + zq$
B3. p sorted, q sorted, $|zp - zq| \le 1 \Rightarrow (p \updownarrow q)$ sorted

PROOF. Since the statement of B3 is symmetric in p, q, assume $zp \ge zq$.

p sorted, q sorted, $|zp - zq| \le 1$
$\Rightarrow$ {assumption: $zp \ge zq$}
p sorted, q sorted, $0 \le zp - zq \le 1$
$\Rightarrow$ {$A2$ and $B1$}
$p \bowtie q$ sorted, $(p \min q) = p$, $(p \max q) = q$
$\Rightarrow$ {replace p, q in $p \bowtie q$ by $(p \min q), (p \max q)$}
$(p \min q) \bowtie (p \max q)$ sorted
$\Rightarrow$ {definition of $p \updownarrow q$}
$p \updownarrow q$ sorted. □

4.6.4 *Correctness of Bitonic Sort.* We show that the function bi applied to a bitonic powerlist returns a sorted powerlist of the original elements: B4 states that bi preserves the number of zeroes of its argument list (i.e., it loses no data), and B5 states that the resulting list is sorted.

B4. $z(bi\,p) = zp$

PROOF. By structural induction using B2.

B5. L bitonic $\Rightarrow (bi\,L)$ sorted. □

PROOF. By structural induction.

Base. Straightforward.

Induction. Let $L = p \bowtie q$

$p \bowtie q$ bitonic
$\Rightarrow$ {A3}
p bitonic, q bitonic, $|zp - zq| \le 1$
$\Rightarrow$ {induction on p and q}
$(bi\,p)$ sorted, $(bi\ q)$ sorted, $|zp - zq| \le 1$
$\Rightarrow$ {from B4: $z(bi\,p) = zp$, $z(bi\,q) = zq$}
$(bi\,p)$ sorted, $(bi\,q)$ sorted, $|z(bi\,p) - z(bi\,q)| \le 1$

$\Rightarrow$ {apply B3 with $(bi\, p), (bi\, q)$ for p, q}

$(bi\, p) \updownarrow (bi\, q)$ sorted

$\Rightarrow$ {definition of bi}

$bi(p \bowtie q)$ sorted. □

4.6.5 *Correctness of Batcher Merge.* We can show that bm merges two sorted powerlists in a manner similar to the proof of bi. Instead, we establish a simple relationship between the functions bm and bi from which the correctness of the former is obvious. We show that:

B6. $p\, bm\, q = bi(p \mid (rev\, q))$, where rev reverses a powerlist (Section 2.2).

If p and q are sorted, then $p \mid (rev\, q)$ is bitonic, which is a fact that we do not prove here. Then, from the correctness of bi, it follows that $bi(p \mid (rev\, q))$ is sorted, so $p\, bm\, q$ is sorted. And it contains the elements of p and q.

PROOF OF B6. By structural induction.

Base. Let $p, q = \langle x \rangle, \langle y \rangle$

$bi(\langle x \rangle \mid rev \langle y \rangle)$

$=$ {definition of rev}

$bi(\langle x \rangle \mid \langle y \rangle)$

$=$ {$(\langle x \rangle \mid \langle y \rangle) = (\langle x \rangle \bowtie \langle y \rangle)$}

$bi(\langle x \rangle \bowtie \langle y \rangle)$

$=$ {definition of bi}

$\langle x \rangle \updownarrow \langle y \rangle$

$=$ {definition of bm}

$\langle x \rangle\, bm\, \langle y \rangle$

Induction. Let $p, q = r \bowtie s,\ u \bowtie v$

$bi(p \mid (rev\, q))$

$=$ {expanding p, q}

$bi((r \bowtie s) \mid rev(u \bowtie v))$

$=$ {definition of rev}

$bi((r \bowtie s) \mid (rev\, v \bowtie rev\, u))$

$=$ {$\mid, \bowtie$ commute}

$bi((r \mid rev\, v) \bowtie (s \mid rev\, u))$

$=$ {definition of bi}

$bi(r \mid rev\, v) \updownarrow bi(s \mid rev\, u)$

$=$ {induction}

$(r\, bm\, v) \updownarrow (s\, bm\, u)$

$=$ {definition of bm}

$(r \bowtie s)\, bm\, (u \bowtie v)$

$=$ {using the definitions of p, q}

$p\, bm\, q$. □

The compactness of the description of Batcher's sorting schemes and the simplicity of their correctness proofs demonstrate the importance of treating recursion and parallelism simultaneously.

4.7 Prefix-Sum

Let L be a powerlist of scalars and $\oplus$ be a binary, associative operator on that scalar type. The prefix sum of L with respect to $\oplus$, $(ps\, L)$, is a list of the same length as L given by

$$ps\, \langle x_0, x_1, \dots, x_i, \dots, x_N \rangle$$
$$= \langle x_0, x_0 \oplus x_1, \dots, x_0 \oplus x_1 \oplus \dots x_i, \dots, x_0 \oplus x_1 \oplus \cdots \oplus x_N \rangle;$$

that is, in $(ps L)$, the element with index i, $i > 0$, is obtained by applying $\oplus$ to the first $(i + 1)$ elements of L in order. We will give a formal definition of prefix-sum later in this section. Prefix sum is of fundamental importance in parallel computing. We show that two known algorithms for this problem can be concisely represented and proved in our theory. Again, *zip* turns out to be the primary operator for describing these algorithms.

A particularly simple scheme for prefix-sum of 8 elements is shown in Figure 4. In that figure, the numbered nodes represent processors, though the same 8 physical processors are used at all levels. Initially, processor i holds the list element L_i, for all i. The connections among the processors at different levels depict data transmissions. In level 0, each processor from 0 through 6 sends its data to its right neighbor. In the ith level, processor k sends its data to $(k + 2^i)$ if such a processor exists; this means that for $j < 2^i$, processor j receives no data in level i data transmission. Each processor updates its own data, d, to $r \oplus d$ where r is the data it receives; if the processor receives no data in some level then d is unchanged. It can be shown that after completion of the computation at level $(\log_2(len\, L))$, processor i holds the ith element of $(ps L)$.

Another scheme, due to Ladner and Fischer [1980], first applies $\oplus$ to adjacent elements x_{2i}, x_{2i+1} to compute the list $\langle x_0 \oplus x_1, \dots x_{2i} \oplus x_{2i+1}, \dots \rangle$. This list has half as many elements as the original list; its prefix-sum is then computed recursively. The resulting list is $\langle x_0 \oplus x_1, \dots, x_0 \oplus x_1 \oplus \cdots \oplus x_{2i} \oplus x_{2i+1}, \dots \rangle$. This list contains half of the elements of the final list; the missing elements are $x_0, x_0 \oplus x_1 \oplus x_2, \dots, x_0 \oplus x_1 \oplus \cdots \oplus x_{2i}, \dots$ These elements can be computed by "adding" $x_2, x_4, \dots,$ appropriately to the elements of the already computed list.

Both schemes for prefix computation are inherently recursive. Our formulations will highlight both parallelism and recursion.

4.7.1 *Specification.* As we did for the sorting schemes (Section 4.6), we introduce an operator in terms of which the prefix-sum problem can be defined. First, we postulate that 0 is the left identity element of $\oplus$, i.e., $0 \oplus x = x$. For a powerlist p, let p^* be the powerlist obtained by shifting p to

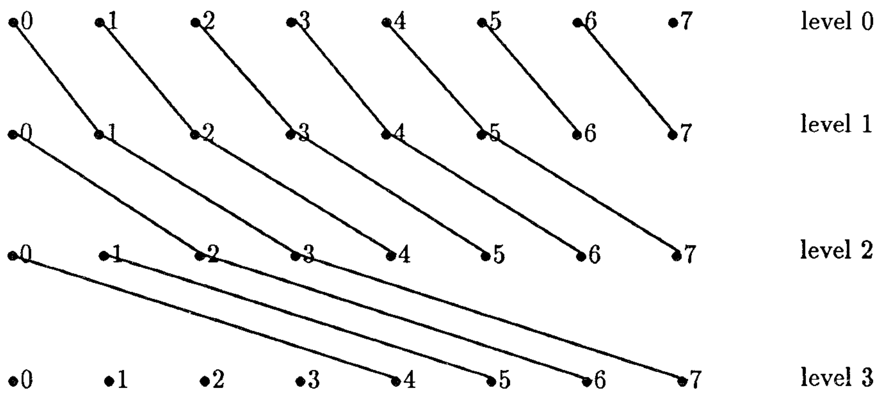


Fig. 4. A network to compute the prefix-sum of eight elements.

the right by one. The effect of shifting is to append a 0 to the left and discard the rightmost element of p; thus, $\langle a\ b\ c\ d\rangle^* = \langle 0\ a\ b\ c\rangle$. Formally,

$$\langle x\rangle^* = \langle 0\rangle$$
$$(p \bowtie q)^* = q^* \bowtie p.$$

Then, it is easy to show

S1. $(r \oplus s)^* = r^* \oplus s^*$
S2. $(p \bowtie q)^{**} = p^* \bowtie q^*$.

Consider the following equation in the powerlist variable z.

$$z = z^* \oplus L \qquad \text{(DE)}$$

where L is some given powerlist. This equation has a unique solution in z because

$$\begin{aligned} z_0 &= (z^*)_0 \oplus L_0 \\ &= 0 \oplus L_0 \\ &= L_0, \quad \text{and} \\ z_{i+1} &= z_i \oplus L_{i+1}, \quad 0 \le i < (len\,L) - 1. \end{aligned}$$

For $L = \langle a\ b\ c\ d\rangle$, $z = \langle a,\ a \oplus b,\ a \oplus b \oplus c,\ a \oplus b \oplus c \oplus d\rangle$, which is exactly $(ps\,L)$. *We define $(ps\,L)$ to be the unique solution of* (DE), *and we call* (DE) *the defining equation for $(ps\,L)$.*

Notes

(1) The operator $\oplus$ is not necessarily commutative. Therefore, the rhs of (DE) may not be the same as $L \oplus z^*$.
(2) The operator $\oplus$ is scalar, so it commutes with $\bowtie$.
(3) The uniqueness of the solution of (DE) can be proved entirely within the powerlist algebra, similar to the derivation of Ladner-Fischer scheme (given later).

(4) Adams [1994] has specified the prefix-sum problem without postulating an explicit 0 element. For any $\oplus$, he introduced a binary operator $\vec{\oplus}$ over two similar powerlists such that $p \mathbin{\vec{\oplus}} q = p^* \oplus q$. The operator $\vec{\oplus}$ can be defined without introducing a 0.

4.7.2 *Computation of the Prefix-Sum.* The function *sps* (*s*imple *p*refix-*s*um) defines the scheme of Figure 4:

$$\begin{aligned} sps\,\langle x \rangle &= \langle x \rangle \\ sps\,L &= (sps\,u) \bowtie (sps\,v), \\ \text{where} \quad u \bowtie v &= L^* \oplus L. \end{aligned}$$

In the first level in Figure 4, $L^* \oplus L$ is computed. If $L = \langle x_0, x_1, \ldots, x_i, \ldots \rangle$, then this is $\langle x_0, x_0 \oplus x_1, \ldots, x_i \oplus x_{i+1} \ldots \rangle$. This is the zip of the two sublists $\langle x_0, x_1 \oplus x_2, \ldots, x_{2i-1} \oplus x_{2i}, \ldots \rangle$ and $\langle x_0 \oplus x_1, \ldots, x_{2i} \oplus x_{2i+1}, \ldots \rangle$. Next, prefix-sums of these two lists are computed independently, then zipped. The Ladner-Fischer scheme is defined by the function *lf*:

$$\begin{aligned} lf\langle x \rangle &= \langle x \rangle \\ lf(p \bowtie q) &= (t^* \oplus p) \bowtie t \\ \text{where} \quad t &= lf(p \oplus q) \end{aligned}$$

4.7.3 *Correctness.* We can prove the correctness of *sps* and *lf* by showing that the function *ps* satisfies the equations defining each of these functions. It is more instructive to see that both *sps* and *lf* can be derived easily from the specification (DE). We carry out this derivation for the Ladner-Fischer scheme as an illustration of the power of algebraic manipulations. First, we note that $ps\langle x \rangle = \langle x \rangle$:

$ps\langle x \rangle$
$=$ {from the defining equation (DE) for $ps\langle x \rangle$}
$(ps\langle x \rangle)^* \oplus \langle x \rangle$
$=$ {definition of *}
$\langle 0 \rangle \oplus \langle x \rangle$
$=$ {$\oplus$ is a scalar operation}
$\langle 0 \oplus x \rangle$
$=$ {0 is the identity of $\oplus$}
$\langle x \rangle$

Derivation of Ladner-Fischer Scheme. Given a powerlist $p \bowtie q$, we derive an expression for $ps(p \bowtie q)$. Let $r \bowtie t$, in unknowns r, t, be $ps(p \bowtie q)$. We solve for r, t:

$r \bowtie t$
$=$ {$r \bowtie t = ps(p \bowtie q)$. Using (DE)}
$(r \bowtie t)^* \oplus (p \bowtie q)$

$= \quad \{(r \bowtie t)^* = t^* \bowtie r\}$
$\quad (t^* \bowtie r) \oplus (p \bowtie q)$
$= \quad \{\oplus, \bowtie \text{ commute}\}$
$\quad (t^* \oplus p) \bowtie (r \oplus q)$

Applying the unique deconstruction law, L2, to the equation $r \bowtie t = (t^* \oplus p) \bowtie (r \oplus q)$, we conclude that:

LF1. $r = t^* \oplus p$.
LF2. $t = r \oplus q$.

Now, we eliminate r from LF2 using LF1 to get $t = t^* \oplus p \oplus q$. Using (DE) and this equation we obtain

LF3. $t = ps(p \oplus q)$.

Below, we summarize the derivation of $ps(p \bowtie q)$:

$\quad ps(p \bowtie q)$
$= \quad \{\text{by definition}\}$
$\quad r \bowtie t$
$= \quad \{\text{Using (LF1) for } r\}$
$\quad (t^* \oplus p) \bowtie t$

where t is defined by LF3. This is exactly the definition of the function *lf* for a nonsingleton powerlist. We also note that

$\quad r$
$= \quad \{\text{by eliminating } t \text{ from LF1 using LF2}\}$
$\quad (r \oplus q)^* \oplus p$
$= \quad \{\text{definition of } ^*\}$
$\quad r^* \oplus q^* \oplus p.$

Using (DE) and this equation, we obtain LF4 that is used in proving the correctness of *sps*.

LF4. $r = ps(q^* \oplus p)$.

Correctness of sps. We show that for a nonsingleton powerlist L,

$$ps\, L = (ps\, u) \bowtie (ps\, v), \quad \text{where } u \bowtie v = L^* \oplus L.$$

PROOF. Let $L = p \bowtie q$. Then we have:

$\quad ps\, L$
$= \quad \{L = p \bowtie q\}$
$\quad ps(p \bowtie q)$
$= \quad \{ps(p \bowtie q) = r \bowtie t, \text{ where } r, t \text{ are given by LF4 and LF3}\}$
$\quad ps(q^* \oplus p) \bowtie ps(p \oplus q)$
$= \quad \{\text{Letting } u = q^* \oplus p,\ v = p \oplus q\}$
$\quad (ps\, u) \bowtie (ps\, v)$

Now, we show that $u \bowtie v = L^* \oplus L$:

$u \bowtie v$
$= \quad \{u = q^* \oplus p,\ v = p \oplus q\}$
$(q^* \oplus p) \bowtie (p \oplus q)$
$= \quad \{\oplus, \bowtie \text{ commute}\}$
$(q^* \bowtie p) \oplus (p \bowtie q)$
$= \quad$ {Apply the definition of * to the first term}
$(p \bowtie q)^* \oplus (p \bowtie q)$
$= \quad \{L = p \bowtie q\}$
$L^* \oplus L \quad \square$

Remarks. A more traditional way of describing a prefix-sum algorithm, such as the simple scheme of Figure 4, is to name explicitly the quantities being computed and establish relationships among them. Let y_{ij} be computed by the ith processor at the jth level. Then, for all i, j, $0 \le i < 2^n$, $0 \le j < n$, where n is the logarithmic length of the list:

$$y_{i0} = x_i, \quad \text{and}$$

$$y_{i,j+1} = \begin{Bmatrix} y_{i-2^j,j}, & i \ge 2^j \\ 0, & i < 2^j \end{Bmatrix} \oplus y_{ij}.$$

The correctness criterion is

$$y_{in} = x_0 \oplus \cdots \oplus x_i.$$

This description is considerably more difficult to manipulate. The parallelism in it is harder to see. The proof of correctness requires manipulations of indices: for this example, we have to show that for all i and j,

$$y_{ij} = x_k \oplus \cdots \oplus x_i$$
$$\text{where} \quad k = \max(0, i - 2^j + 1).$$

The Ladner-Fischer scheme is even more difficult to specify in this manner. Algebraic methods seem preferable for describing uniform operations on aggregates of data.

5. HIGHER-DIMENSIONAL ARRAYS

A major part of parallel computing involves arrays of one or more dimensions. An array of m dimensions (dimensions are numbered 0 through $m - 1$) is represented by a powerlist of depth $m - 1$. Conversely, since powerlist elements are similar, a powerlist of depth $m - 1$ may be regarded as an array of dimension m. For instance, a matrix of r rows and c columns may be represented as a powerlist of c elements, each element being a powerlist of length r storing the items of a column; conversely, the same matrix may be represented by a powerlist of r elements, each of which is a powerlist of c elements.

$$A = \left\langle \begin{array}{cc} \wedge & \wedge \\ 2 & 4 \\ 3 & 5 \\ \vee & \vee \end{array} \right\rangle \qquad B = \left\langle \begin{array}{cc} \wedge & \wedge \\ 0 & 1 \\ 6 & 7 \\ \vee & \vee \end{array} \right\rangle$$

$$A \mid B = \left\langle \begin{array}{cccc} \wedge & \wedge & \wedge & \wedge \\ 2 & 4 & 0 & 1 \\ 3 & 5 & 6 & 7 \\ \vee & \vee & \vee & \vee \end{array} \right\rangle \qquad A \bowtie B = \left\langle \begin{array}{cccc} \wedge & \wedge & \wedge & \wedge \\ 2 & 0 & 4 & 1 \\ 3 & 6 & 5 & 7 \\ \vee & \vee & \vee & \vee \end{array} \right\rangle$$

$$A \mid' B = \begin{array}{c} \begin{array}{cc} \wedge & \wedge \end{array} \\ \left\langle \begin{array}{cc} 2 & 4 \\ 3 & 5 \\ 0 & 1 \\ 6 & 7 \end{array} \right\rangle \\ \begin{array}{cc} \vee & \vee \end{array} \end{array} \qquad A \bowtie' B = \begin{array}{c} \begin{array}{cc} \wedge & \wedge \end{array} \\ \left\langle \begin{array}{cc} 2 & 4 \\ 0 & 1 \\ 3 & 5 \\ 6 & 7 \end{array} \right\rangle \\ \begin{array}{cc} \vee & \vee \end{array} \end{array}$$

Fig. 5. Applying $\mid$, $\bowtie$, $\mid'$, $\bowtie'$ over matrices. Matrices are stored by columns. Typical matrix format is used for display, though each matrix is to be regarded as a powerlist of powerlists.

In manipulating higher-dimensional arrays, we prefer to think in terms of array operations rather than operations on nested powerlists. Therefore, we introduce construction operators analogous to $\mid$ and $\bowtie$ for *tie* and *zip* along any specified dimension. We use $\mid'$, $\bowtie'$ for the corresponding operators in dimension 1, $\mid''$, $\bowtie''$ for the dimension 2, etc. The definitions of these operators are in Section 5.2; for the moment it is sufficient to regard $\mid'$ and $\bowtie'$ as the pointwise applications of $\mid$ and $\bowtie$, respectively, to the argument powerlists. Thus, for similar power matrices A and B that are stored columnwise (i.e., each element is a column), $A \mid B$ is the concatenation of A and B by rows, and $A \mid' B$ is their concatenation by columns. Figure 5 shows applications of these operators on specific matrices.

Given these constructors, we may define a matrix to be either

a singleton matrix $\langle\langle x \rangle\rangle$, or
$p \mid q$ where p, q are similar matrices, or
$u \mid' v$ where u, v are similar matrices.

Analogous definitions can be given for n-dimensional arrays. Observe that the length of each dimension is a power of 2. As in the case of a powerlist, the same matrix can be constructed in several different ways, first by constructing the rows, then the columns, or vice versa. We will show in Section 5.2 that

$$(p \mid q) \mid' (u \mid v) = (p \mid' u) \mid (q \mid' v),$$

or that $\mid, \mid'$ commute.

Note. We could have defined a matrix using $\bowtie$ and $\bowtie'$ instead of $\mid$ and $\mid'$. As $\mid$ and $\bowtie$ are duals in the sense that either can be used to construct or uniquely deconstruct a powerlist, $\mid'$ and $\bowtie'$ are also duals, as shown in Section 5.2. Therefore, we will freely use all four construction operators for matrices.

Example (Matrix Transposition). Let τ be a function that transposes matrices. From the definition of a matrix, we have to consider three cases in defining τ:

$$\begin{aligned}\tau\langle\langle x\rangle\rangle &= \langle\langle x\rangle\rangle\\ \tau(p \mid q) &= (\tau\, p) \mid' (\tau\, q)\\ \tau(u \mid' v) &= (\tau\, u) \mid (\tau\, v).\end{aligned}$$

The description of function τ, though straightforward, has introduced the possibility of an inconsistent definition. For a 2×2 matrix, for instance, either of the last two deconstructions apply, and it is not obvious that the same result is obtained independent of the order in which the rules are applied. We show that τ is indeed a function.

We prove the result by structural induction. For a matrix of the form $\langle\langle x\rangle\rangle$, only the first deconstruction applies, so the claim holds. Next, consider a matrix to which both of the last two deconstructions apply. Such a matrix is of the form $(p \mid q) \mid' (u \mid v)$ which, as remarked above, is also $(p \mid' u) \mid (q \mid' v)$. Applying one step of each of the last two rules in different order, we get

$$\begin{aligned}&\tau((p \mid q) \mid' (u \mid v))\\ = &\quad \{\text{applying the last rule}\}\\ &(\tau(p \mid q)) \mid (\tau(u \mid v))\\ = &\quad \{\text{applying the middle rule}\}\\ &((\tau\, p) \mid' (\tau\, q)) \mid ((\tau\, u) \mid' (\tau\, v))\end{aligned}$$

and

$$\begin{aligned}&\tau((p \mid' u) \mid (q \mid' v))\\ = &\quad \{\text{applying first the middle rule, then the last rule}\}\\ &((\tau\, p) \mid (\tau\, u)) \mid' ((\tau\, q) \mid (\tau\, v))\\ = &\quad \{\mid, \mid' \text{ commute}\}\\ &((\tau\, p) \mid' (\tau\, q)) \mid ((\tau\, u) \mid' (\tau v))\end{aligned}$$

From the induction hypothesis, $(\tau\, p)$, $(\tau\, q)$, etc., are well defined. Hence,

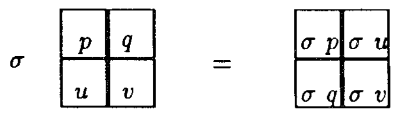
$$\tau((p \mid q) \mid' (u \mid v)) = \tau((p \mid' u) \mid (q \mid' v)).$$

Crucial to the above proof is the fact that $\mid$ and $\mid'$ commute; this is reminiscent of the "Church-Rosser Property" [Church 1941] in term-rewriting systems. Commutativity is so important that we discuss it further in the next subsection. It is easy to show that

$$\begin{aligned}\tau(p \bowtie q) &= (\tau\, p) \bowtie' (\tau\, q)\\ \tau(u \bowtie' v) &= (\tau\, u) \bowtie (\tau\, v).\end{aligned}$$

Transposition of a square power matrix can be defined by deconstructing the matrix into quarters, transposing them individually, then rearranging

Fig. 6. Schematic of the transposition of a square powermatrix.



them, as shown in Figure 6. From the transposition function τ for general matrices, we get a function σ for transpositions of square matrices:

$$\sigma\langle\langle x\rangle\rangle = \langle\langle x\rangle\rangle$$
$$\sigma((p\,|\,q)\,|'\,(u\,|\,v)) = ((\sigma\, p)\,|'\,(\sigma\, q))\,|\,((\sigma\, u)\,|'\,(\sigma\, v))$$

Note the effectiveness of pattern matching in this definition.

5.1 Pointwise Application

Let f be a function mapping items of type α to type β. Then f' maps a powerlist of α-items to a powerlist of β-items:

$$f'\langle x\rangle = \langle fx\rangle$$
$$f'(r\,|\,s) = (f'\,r)\,|\,(f'\,s).$$

Similarly, for a binary operator *op*:

$$\langle x\rangle\ op'\ \langle y\rangle = \langle x\, op\, y\rangle$$
$$(r\,|\,s)\ op'\ (u\,|\,v) = (r\ op'\ u)\,|\,(s\ op'\ v).$$

We have explicitly defined these two forms because we use one or the other in all our examples; f' for a function f of arbitrary arity is similarly defined. Observe that f' applied to a powerlist of length N yields a powerlist of length N. The number of primes over f determines the dimension at which f is applied; the outermost dimension is numbered 0. Therefore writing $\bowtie$, for instance, without primes, simply zips two lists. The operator for pointwise application also appears in Backus [1978] and in Steele and Hillis [1986].

Common special cases for the binary operator *op* are $|$ and $\bowtie$ and their pointwise application operators. In particular, writing $\bowtie^m$ to denote

$$\bowtie\overbrace{''\ldots'}^{m},$$

we define, $\bowtie^0\ = \bowtie$ and for $m > 0$:

$$\langle x\rangle \bowtie^m \langle y\rangle = \langle x \bowtie^{m-1} y\rangle$$
$$(r\,|\,s) \bowtie^m (u\,|\,v) = (r \bowtie^m u)\,|\,(s \bowtie^m v).$$

From the definition of f', we conclude that f' and $|$ commute. Below, we prove that f' commutes with $\bowtie$.

THEOREM 5.1.1. *f', $\bowtie$ commute.*

PROOF. We prove the result for unary f; the general case is similar. Proof is by structural induction.

Base.

$$\begin{array}{ll} & f'(\langle x\rangle \bowtie \langle y\rangle) \\ = & \{\langle x\rangle \bowtie \langle y\rangle = \langle x\rangle \mid \langle y\rangle\} \\ & f'(\langle x\rangle \mid \langle y\rangle) \\ = & \{\text{definition of } f'\} \\ & f'\langle x\rangle \mid f'\langle y\rangle \\ = & \{f'\langle x\rangle, f'\langle y\rangle = \langle f\ x\rangle, \langle f\ y\rangle. \text{ These are singleton lists}\} \\ & f'\langle x\rangle \bowtie f'\langle y\rangle \end{array}$$

Induction.

$$\begin{array}{ll} & f'((p \mid q) \bowtie (u \mid v)) \\ = & \{\mid, \bowtie \text{ in the argument commute}\} \\ & f'((p \bowtie u) \mid (q \bowtie v)) \\ = & \{f', \mid \text{commute}\} \\ & f'(p \bowtie u) \mid f'(q \bowtie v) \\ = & \{\text{induction}\} \\ & ((f'\ p) \bowtie (f'\ u)) \mid ((f'\ q) \bowtie (f'\ v)) \\ = & \{\mid, \bowtie \text{ commute}\} \\ & ((f'\ p) \mid (f'\ q)) \bowtie ((f'\ u) \mid (f'\ v)) \\ = & \{f', \mid \text{commute}\} \\ & (f'(p \mid q)) \bowtie (f'(u \mid v)) \quad \square \end{array}$$

THEOREM 5.1.2. *For a scalar function f, $f' = f$.*

PROOF. Proof by structural induction is straightforward. □

THEOREM 5.1.3. *If f, g commute then so do f', g'.*

PROOF. By structural induction. □

The following results about commutativity can be derived from Theorems 5.1.1–5.1.3. In the following, m, n are natural numbers.

C1. For any f and $m > n$: $f^m, |^n$ commute, and f^m, $\bowtie^n$ commute.
C2. For $m \neq n$: $|^m, |^n$ commute, and $\bowtie^m$, $\bowtie^n$ commute.
C3. For all m, n: $|^m$, $\bowtie^n$ commute.
C4. For any scalar function f: $f, |^m$ commute, and f, $\bowtie^n$ commute.

C1 follows by applying induction on Theorems 5.1.1 and 5.1.3 and the fact that $f', |$ commute. C2 follows from C1; C3 follows from C1, L3, and Theorem 5.1.3; C4 from C1 and Theorem 5.1.2.

5.2 Deconstruction

In this section, we show that any powerlist that can be written as $p \mid^m q$ for some p, q can also be written as $u \bowtie^m v$ for some u, v and vice versa; this is analogous to L1 for dual deconstruction. Analogous to L2, we show that such deconstructions are unique.

THEOREM 5.2.1 DUAL DECONSTRUCTION. *For any p, q and $m \geq 0$, if $p \mid^m q$ is defined then there exist u, v such that $u \bowtie^m v = p \mid^m q$. Conversely, for any u, v and $m \geq 0$, if $u \bowtie^m v$ is defined then there exist some p, q such that $p \mid^m q = u \bowtie^m v$.*

We do not prove this theorem, for its proof is similar to the theorem given below.

THEOREM 5.2.2 UNIQUE DECONSTRUCTION. *Let $\otimes$ be $\mid$ or $\bowtie$. For any natural number m, $(p \otimes^m q = u \otimes^m v) \equiv (p = u \wedge q = v)$.*

PROOF. Proof is by induction on m:

$m = 0$. The result follows from L2.

$m = n + 1$. Assume that $\oplus = \mid$. The proof is similar for $\oplus = \bowtie$. We prove the result by structural induction on p.

Base.

$$
\begin{aligned}
& p = \langle a\rangle,\ q = \langle b\rangle,\ u = \langle c\rangle,\ v = \langle d\rangle \\
& \langle a\rangle \mid^{n+1} \langle b\rangle = \langle c\rangle \mid^{n+1} \langle d\rangle \\
\equiv\ & \{\text{definition of } \mid^{n+1}\} \\
& \langle a \mid^n b\rangle = \langle c \mid^n d\rangle \\
\equiv\ & \{\text{unique deconstruction using L2}\} \\
& a \mid^n b = c \mid^n d \\
\equiv\ & \{\text{induction on } n\} \\
& (a = c) \wedge (b + d) \\
\equiv\ & \{\text{L2}\} \\
& (\langle a\rangle = \langle c\rangle) \wedge (\langle b\rangle = \langle d\rangle)
\end{aligned}
$$

Induction.

$$
\begin{aligned}
& p = p_0 \mid p_1,\ q = q_0 \mid q_1,\ u = u_0 \mid u_1,\ v = v_0 \mid v_1 \\
& (p_0 \mid p_1) \mid^{n+1} (q_0 \mid q_1) = (u_0 \mid u_1) \mid^{n+1} (v_0 \mid v_1) \\
\equiv\ & \{\text{definition of } \mid^{n+1}\} \\
& (p_0 \mid^{n+1} q_0) \mid (p_1 \mid^{n+1} q_1) = (u_0 \mid^{n+1} v_0) \mid (u_1 \mid^{n+1} v_1) \\
\equiv\ & \{\text{unique deconstruction using L2}\} \\
& (p_0 \mid^{n+1} q_0) = (u_0 \mid^{n+1} v_0) \wedge (p_1 \mid^{n+1} q_1) = (u_1 \mid^{n+1} v_1) \\
\equiv\ & \{\text{induction on the length of } p_0, q_0, p_1, q_1\} \\
& (p_0 = u_0) \wedge (q_0 = v_0) \wedge (p_1 = u_1) \wedge (q_1 = v_1) \\
\equiv\ & \{\text{L2}\} \\
& (p_0 \mid p_1) = (u_0 \mid u_1) \wedge (q_0 \mid q_1) = (v_0 \mid v_1). \quad \square
\end{aligned}
$$

Theorems 5.2.1 and 5.2.2 allow a richer variety of pattern matching in function definitions, as we did for matrix transposition. We may employ $\mid^m$, $\bowtie^n$ for any natural m, n to construct a pattern over which a function can be defined.

5.3 Embedding Arrays in Hypercubes

An n-dimensional *hypercube* is a graph of 2^n nodes, $n \geq 0$, where each node has a unique n-bit label. Two nodes are *neighbors*, i.e, there is an edge between them, exactly when their labels differ in a single bit. Therefore, every node has n neighbors. We may represent a n-dimensional hypercube as a powerlist of depth n; each level except the innermost consists of two powerlists. The operators $|^m$, $\bowtie^n$ for natural m, n can be used to access the nodes in any one or any combination of dimensions.

We conclude with an example that shows how higher-dimensional structures such as hypercubes are easily handled in our theory. Given an array of size $2^{m_0} \times 2^{m_1} \times \cdots 2^{m_d}$, we claim that its elements can be placed at the nodes of a hypercube of the dimension $m_0 + m_1 + \cdots + m_d$ such that two "adjacent" data items in the array are placed at neighboring nodes in the hypercube. Here, two data items of the array are *adjacent* if their indices differ in exactly one dimension and by 1 modulo N where N is the size of that dimension. This is called "wrap around" adjacency.

The following embedding algorithm is described in Leighton [1992, Section 3.1.2]; it works as follows: if the array has only one dimension with 2^m elements, then we create a gray code sequence, Gm (see Section 4.3). Abbreviate Gm by g. We place the ith item of the array at the node with label g_i. Adjacent items, at positions i and $i+1$ ($+$ is taken modulo 2^m), are placed at nodes g_i and g_{i+1}, which differ in exactly one bit, by the construction.

This idea can be generalized to higher-dimensional arrays as follows. Construct gray code sequences for each dimension independently; store the item with index $(i_0, i_1, \ldots, i_d)$ at the node $(g_{i_0}; g_{i_1}; \ldots; g_{i_d})$ where ";" denotes the concatenations of the bit strings. By definition, adjacent items differ by 1 in exactly one dimension, k. Then, their gray code indices are identical in all dimensions except k, and they differ in exactly one bit in dimension k.

We describe a function *em* that embeds an array in a hypercube. Given an array of size $2^{m_0} \times 2^{m_1} \times \cdots 2^{m_d}$, it permutes its elements to an array

$$\underbrace{2 \times 2 \times \cdots \times 2}_{m}$$

where $m = m_0 + \cdots + m_d$, and the permutation preserves array adjacency as described. The algorithm is inspired by the gray code function of Section 4.3. In the following, S matches only with a scalar and P with a powerlist:

$$\begin{aligned} em\langle S \rangle &= \langle S \rangle \\ em\langle P \rangle &= em\,P \\ em(u \mid v) &= \langle em\,u \rangle \mid \langle em\,(rev\,v) \rangle. \end{aligned}$$

The first line is the rule for embedding a single item in a 0-dimensional hypercube. The next line simply says that an array having length 1 in a dimension can be embedded by ignoring that dimension. The last line says that we can embed a nonsingleton array by embedding the left half of dimension 0 and the reverse of the right half in the two component hypercubes of a larger hypercube.

6. REMARKS

6.1 Related Work

Applying uniform operations on aggregates of data have proved to be extremely powerful in APL [Iverson 1962]; see Backus [1978] and Bird [1989] for algebras of such operators. One of the earliest attempts at representing data-parallel algorithms is in Preparata and Vuillemin [1981]: "an algorithm... performs a sequence of basic operations on pairs of data that are successively $2^{(k-1)}, 2^{(k-2)}, \ldots, 2^0 = 1$ locations apart." An algorithm operating on 2^N pieces of data is described as a sequence of N parallel steps of the above form where the kth step, $0 < k \leq N$, applies in parallel a binary operation OPER on pairs of data that are $2^{(N-k)}$ apart. Preparata and Vuillemin show that this paradigm can be used to describe a large number of known parallel algorithms and that any such algorithm can be efficiently implemented on the Cube Connected Cycle connection structure. Their style of programming was imperative, making it difficult to apply algebraic manipulations to such programs. Their programming paradigm fits in well within our notation.

Mou and Hudak [1988] and Mou [1991] propose a functional notation to describe divide-and-conquer-type parallel algorithms. Their notation is a vast improvement over Preparata and Vuillemin's in that changing from an imperative style to a functional style of programming allows more succinct expressions and the possibility of algebraic manipulations. The effectiveness of this programming style on a scientific problem may be seen in Wang and Mou [1991]. They have constructs similar to *tie* and *zip*, though they allow unbalanced decompositions of lists. An effective method of programming with vectors has been proposed in Blelloch [1990; 1993]. He proposed a small set of "vector-scan" instructions that may be used as primitives in describing parallel algorithms. Unlike our method, his allows control over the division of the list and the number of iterations, depending on the values of the data items, a necessary ingredient in many scientific problems. Jones and Sheeran [1990] have developed a relational algebra for describing circuit components. A circuit component is viewed as a relation, and the operators for combining relations are given appropriate interpretations in the circuit domain.

Kapur and Subramaniam [1994] have implemented the powerlist notation for the purpose of automatic theorem proving. They have proved many of the algorithms in this article using an inductive theorem prover *R*ewrite *R*ule *L*aboratory, or RRL, that is based on equality reasoning and rewrite rules. They are now extending their theorem prover so that the similarity constraints on the powerlist constructors do not have to be stated explicitly.

One of the fundamental problems with the powerlist notation is to devise compilation strategies for mapping programs to specific architectures. The architecture that is the closest conceptually is the hypercube. Kornerup [1994] has developed certain strategies whereby each parallel step in a program is mapped to a constant number of local operations and communications at a hypercube node.

Combinational circuit verification is an area in which the powerlist notation may be fruitfully employed. Adams [1994] has proved the correctness of adder circuits using this notation. A ripple-carry adder is typically easy to describe and prove, whereas a carry-lookahead adder is much more difficult. Adams described both circuits in our notation and proved their equivalence in a remarkably concise fashion. In his work, he obtained a succinct description of the carry-lookahead circuit by employing the prefix-sum function treated in Section 4.7.

6.2 Powerlists of Arbitrary Length

We restricted the lengths of powerlists to be of the form 2^n, $n \geq 0$, because we could then develop a simple theory. For handling arbitrary-length lists, Steele (personal communication, 1993) suggests padding enough "dummy" elements to a list to make its length a power of 2. The advantage of this scheme is that we still retain the simple algebraic laws of a powerlist. Another approach is based on the observation that any positive integer is either 1 or $2 \times m$ or $2 \times m + 1$, for some positive integer m; therefore, we deconstruct a nonsingleton list of odd length into two lists p and q, and an element e, where e is either the first, middle, or last element. For instance, the following function, *rev*, reverses a list:

$$\begin{aligned} rev\,\langle x \rangle &= \langle x \rangle \\ rev\,(p \mid q) &= (rev\, q) \mid (rev\, p) \\ rev\,(p \mid e \mid q) &= (rev\, q \mid e \mid rev\, p). \end{aligned}$$

The last line of this definition applies to a nonsingleton list of odd length; the list is deconstructed into two lists, p and q of equal length, and the middle element e (We have abused the notation, applying $\mid$ to three arguments). Similarly, the function *lf* for a prefix-sum may be defined by:

$$\begin{aligned} lf\langle x \rangle &= \langle x \rangle \\ lf(p \bowtie q) &= (t^* \oplus p) \bowtie t \\ lf(e \bowtie p \bowtie q) &= e \bowtie (e \oplus (t^* \oplus p)) \bowtie (e \oplus t) \\ \text{where} \qquad t &= lf(p \oplus q). \end{aligned}$$

In this definition, the singleton list and lists of even length are treated as before. A list of odd length is deconstructed into e, p, q, where e is the first element of the argument list and $p \bowtie q$ constitutes the remaining portion. In this case, the prefix-sum is obtained by appending the element e to the list obtained by applying $e \oplus$ to each element of $lf(p \bowtie q)$. We have used the convention that $e \oplus L$ is the list obtained by applying $e \oplus$ to each element of list L.

6.3 The Interplay between Sequential and Parallel Computations

The notation proposed in this article addresses only a small aspect of parallel computing. Powerlists have proved to be highly successful in expressing computations that are independent of the specific data values; such is the case, for instance, in the Fast Fourier Transform, the Batcher merge, and the prefix-sum. Typically, however, parallel and sequential computations are interleaved. While the Fast Fourier Transform and Batcher merge represent highly parallel computations, binary search is inherently sequential (there are other parallel search strategies). Gaussian elimination represents a mixture; the computation consists of a sequence of pivoting steps where each step can be applied in parallel. Thus, parallel computations may have to be performed in a certain sequence, and the sequence may depend on the data values during a computation. More general methods as in Blelloch [1990] are then required.

The powerlist notation can be integrated into a language that supports sequential computation. In particular, this notation blends well with ML [Milner et al. 1990] and LISP [McCarthy et al. 1962; Steele and Hillis 1986]. A mixture of linear lists and powerlists can exploit the various combinations of sequential and parallel computing. A powerlist consisting of linear lists as components admits of parallel processing in which each component is processed sequentially. A linear list whose elements are powerlists suggests a sequential computation where each step can be applied in parallel. Powerlists of powerlists allow multidimensional parallel computations, whereas a linear list of linear lists may represent a hierarchy of sequential computations.

ACKNOWLEDGMENTS

This article has been enriched by comments and suggestions from Will Adams, Al Carruth, who suggested the term powerlist, Jorge Cobb, Edsger W. Dijkstra, C. A. R. Hoare, Rajeev Joshi, Markus Kaltenbach, Deepak Kapur, Jacob Kornerup, Scott Page, Vijaya Ramachandran, Guy Steele, Jr., Alex Tomlinson, and Evelyn Tumlin. Ernie Cohen was singularly helpful at an early stage of the research. I am grateful to the Austin Tuesday Afternoon Club which read and commented on a draft of this manuscript, and to Adams and Kornerup for their comments on the second draft. Deepak Kapur and M. Subramaniam have increased my faith in this work by implementing it. The current draft has benefitted from the comments of Andrew Appel and two anonymous referees.

REFERENCES

ADAMS, W 1994 Verifying adder circuits using powerlists. Tech Rep TR 94-02, Dept of Computer Science, Univ. of Texas, Austin (March).

BACKUS, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug.), 613–641.

BATCHER, K. 1968. Sorting networks and their applications In *Proceeding of the AFIPS Spring Joint Computer Conference*. Vol 32. AFIPS Press, Reston, Va., 307–314.

BIRD, R. S. 1989. Lectures on constructive functional programming. In *Constructive Methods in Computing Science*, M. Broy, Ed. NATO ASI Series F: Computer and Systems Sciences. Springer-Verlag, New York, 151–216.

BLELLOCH, G. E. 1993. NESL: A nested data-parallel language Tech. Rep CMU-CS-93-129, School of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa (April).

BLELLOCH, G. E. 1990. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, Mass.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Mass.

CHURCH, A. 1941. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, N.J.

COOLEY, J. M. AND TUKEY, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Math. Comput. 19*, 90, 297–301.

GRAY, F. 1953. Pulse code communication. U.S. Patent 2,632,058.

IVERSON, K. 1962. *A Programming Language*. John Wiley and Sons, New York.

JONES, G. AND SHEERAN, M. 1990. Circuit design in Ruby. In *Formal Methods for VLSI Design*, J. Staunstrup, Ed. North Holland, New York.

KAPUR, D. AND SUBRAMANIAM, M. 1994. Automated reasoning about parallel algorithms using powerlists. State Univ. of New York, Albany. Unpublished.

KARP, R. M. AND RAMACHANDRAN, V. 1990. Parallel algorithms for shared memory machines. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier North-Holland, New York.

KNUTH, D. E. 1973. *Sorting and Searching*. Vol. 3. *The Art of Computer Programming*. Addison-Wesley, Reading, Mass.

KORNERUP, J. 1994. Mapping powerlists onto hypercubes. Ph.D. thesis, The Univ. of Texas, Austin. To be published.

LADNER, R. E. AND FISCHER, M. J. 1980. Parallel prefix computation. *J. ACM 27*, 4, 831–838

LEIGHTON, F. T. 1992. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann Publishers, San Mateo, Calif.

MCCARTHY, J., ABRAHAMS, P. W., EDWARDS, D. J., HART, T. P., AND LEVIN, M. I. 1962. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Mass.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Mass.

MISRA, J. 1994. Powerlist: A structure for parallel recursion. In *A Classical Mind: Essays in Honour of C.A R. Hoare*, A. Roscoe, Ed. Prentice Hall International, New York, 295–316.

MOU, Z. 1991. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*. 451–461.

MOU, Z. G. AND HUDAK, P. 1988. An algebraic model for divide-and-conquer algorithms and its parallelism. *J. Supercomput. 2*, 3 (Nov.), 257–278.

PREPARATA, F. P. AND VUILLEMIN, J. 1981. The cube-connected cycles: A versatile network for parallel computation. *Commun. ACM 24*, 5 (May), 300–309.

STEELE, G. L., JR. AND HILLIS, D. 1986. Connection Machine Lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*. ACM, New York, 279–297.

TURNER, D. 1986. An overview of Miranda. *ACM SIGPLAN Not. 21*, 156–166.

WANG, X. AND MOU, Z. 1991. A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*. IEEE, New York, 810–817.

Received October 1993; revised February 1994; accepted June 1994