# Parlists – A Generalization of Powerlists

Jacob Kornerup

Department of Computer Science, Southern Methodist University, Dallas, TX 75275;
kornerup@seas.smu.edu

## 1 Introduction

The powerlist notation [5] has proven to be a major step forward in describing parallel algorithms succinctly. It allows the programmer to work at a high level of abstraction, by avoiding indexing notations, leading towards efficient implementations on parallel architectures [2]. The powerlist data structure is a list whose length is a power of two. In the powerlist notation it is possible to elegantly specify algorithms such as the Discrete Fast Fourier Transform without resorting to "index gymnastics" [5]. For such algorithms this restriction on the lengths is not serious, as they are often presented this way in the literature. However, for most algorithms the restriction is unnatural. In this paper we present an extension of the powerlist notation to lists of arbitrary positive lengths and work through a number of examples[1]. This new data structure is called "ParList", which is short for *parallel list.*

## 2 ParList Theory

A ParList is a non-empty list, whose elements are all of the same type, either scalars from the same base type, or (recursively) ParLists that enjoy the same property. Two ParLists are *similar* if they have the same length and their elements are similar; two scalars are similar when they are from the same base type. We categorize ParLists according to their length. The shortest ParList has length 1, it is called a *singleton*; $\langle x \rangle$ denotes the singleton containing the scalar $x$.

A non-singleton ParList $v$ can be *deconstructed* into a single element and a ParList whose length is one less than that of $v$, using the $\triangleright$ ("cons") and the $\triangleleft$ ("snoc") operator:

$$v = a \triangleright p \ \wedge \ v = q \triangleleft b \tag{1}$$

where $a$, $b$ and the elements of $p$ and $q$ are similar to the elements of $v$, and $p$ and $q$ are similar ParLists. In (1) $a$ is the first element of $v$ and $b$ is the last element of $v$.

A ParList, $p$, of even length has the property that it can be deconstructed using the $\bowtie$ ("zip") and the $\mid$ ("tie") operator:

$$p = u \bowtie v \ \wedge \ p = r \mid s \tag{2}$$

[1] For a detailed presentation of these results, including proofs of claims, see the full paper [3], available as ftp://ftp.cs.utexas.edu/pub/techreports/tr97-17.ps.Z

where $u$ is a ParList containing the elements of $p$ at even positions, and $v$ is the ParList containing the elements of $p$ at odd positions. Similarly, $r$ is the ParList containing the first half of $p$ and $s$ is the second half of $p$; the ParLists $r, s, u$ and $v$ are all similar.

### 2.1 Axioms

In the following we extend the axioms of the powerlist theory [5] to an axiomatization of the ParList algebra. Let $p, q, u, v$ be similar ParLists over the same base type as the elements $a, b, c$ and $t$ be a non-singleton ParList over the same base type. The ParList algebra has five constructors: $\langle\ \rangle, |, \bowtie, \triangleright$ and $\triangleleft$. They are all *isomorphisms* on their respective domains, i.e. they each satisfy two axioms like the ones we show for $\triangleleft$ below:

$$p \triangleleft a = q \triangleleft b \ \equiv\ a = b \ \wedge\ p = q \tag{3}$$

$$(\forall t : t \text{ is not a singleton} : (\exists\, b, q :: t = q \triangleleft b)) \tag{4}$$

The following axioms define the the full ParList algebra:

$$\langle a \rangle \bowtie \langle b \rangle = \langle a \rangle \mid \langle b \rangle \tag{5}$$

$$(p \mid q) \ \bowtie\ (u \mid v) = (p \bowtie u) \ \mid\ (q \bowtie v) \tag{6}$$

$$(a \triangleright (p \mid q)) \ \bowtie\ ((u \mid v) \triangleleft b) = (a \triangleright (u \bowtie p)) \ \mid\ ((v \bowtie q) \triangleleft b) \tag{7}$$

$$a \triangleright \langle b \rangle = \langle a \rangle \mid \langle b \rangle \tag{8}$$

$$\langle a \rangle \triangleleft b = \langle a \rangle \mid \langle b \rangle \tag{9}$$

$$a \triangleright (p \triangleleft b) = (a \triangleright p) \triangleleft b \tag{10}$$

$$a \triangleright (p \bowtie q) = (u \bowtie p) \triangleleft b \equiv a \triangleright q = u \triangleleft b \tag{11}$$

$$a \triangleright (p \mid q) = (u \mid v) \triangleleft c \equiv (\exists\, b :: a \triangleright p = u \triangleleft b \ \wedge\ b \triangleright q = v \triangleleft c) \tag{12}$$

The axioms (5) and (6) come from the powerlist algebra; note the symmetry between $\bowtie$ and $\mid$ in axiom (6). Without an operational model the roles of $\bowtie$ and $\mid$ can be interchanged in the powerlist algebra. This is not the case when we consider the ParList algebra. If we interpret $\triangleright$ and $\triangleleft$ as prepending and appending an element to a ParList then the contrast between (11) and (12) and between (7) and (6) precisely capture the operational difference between $\bowtie$ and $|$.

Let $\oplus$ be a binary operator, defined on a scalar type. We lift $\oplus$ to operate on ParList over elements of that type with the following laws:

$$\langle a \rangle \oplus \langle b \rangle = \langle a \oplus b \rangle \tag{13}$$

$$(a \triangleright p) \oplus (b \triangleright q) = (a \oplus b) \triangleright (p \oplus q) \tag{14}$$

$$(p \bowtie q) \oplus (u \bowtie v) = (p \oplus u) \ \bowtie\ (q \oplus v) \tag{15}$$

### 2.2 Functions in ParList

Functions over ParList are defined by three different cases based on the length of the argument ParList: singleton, even length and odd length. Each case is defined

using pattern-matching on the argument ParList: $\langle\ \rangle$ for singletons, $\bowtie$ or $|$ for even length lists, and $\triangleright$ or $\triangleleft$ for odd length lists. We insist that $\triangleright$ and $\triangleleft$ only be used for ParLists of odd length in function definitions, since we want to exploit parallelism as much as possible. When the argument has an even length, the computation should be expressed using a balanced divide-and-conquer strategy. Arguments of odd length should be treated as an alignment step, introduced by necessity.

As an example, we define the function *rev* that reverses its argument.

$$rev.\langle a\rangle = \langle a\rangle \tag{16}$$

$$rev.(p \bowtie q) = rev.q \bowtie rev.p \tag{17}$$

$$rev.(a \triangleright p) = rev.p \triangleleft a \tag{18}$$

Note that the choice of $\bowtie$ and $\triangleright$ as destructors was arbitrary. A definition using $|$ and/or $\triangleleft$ in their place yields the same function. In the definition of *rev*, (17) expresses that each recursive case is independent and can be evaluated in parallel. The step described by (18) corresponds to a sequential "alignment" step, necessary before a balanced recursive step can be performed. In the case of *rev* the "alignment" step does not have to be sequential; depending on the parallel architecture (and the concrete implementation of ParList) *rev* can be evaluated in constant time. This would be the case on a CREW PRAM with the straightforward implementation of ParList.

It is possible to reuse proof of properties of powerlist functions, when the function definition is extended with a case for odd length lists. It is only necessary to specify and verify the odd case, assuming that the existing proof of the even case does not rely on properties that are specific to powerlists. E.g. had we already proved that $rev.(rev.p) = p$ for the powerlist function defined by (16) and (17), we would only need to prove that $rev.(rev.(a \triangleright p)) = a \triangleright p$ using (18).

### 2.3 Prefix Sum

Prefix sum is a fundamental parallel algorithm; it is used in many algorithms as a building block, e.g. carry lookahead addition (see Sect. 3). The prefix sum of a ParList $p$ over a data type Y, with the property that $(Y, +, 0)$ is a monoid, can be defined [5] as the (unique) solution to the equation (in $u$):

$$u = (0 \rightarrow u) + p \tag{19}$$

where the operator $\rightarrow$ takes a element and a ParList and "pushes" a scalar into the list from the left and the rightmost element of the list is lost. $\rightarrow$ has a higher binding power than that of $\bowtie, |, \triangleright$ and $\triangleleft$ ; it is defined as follows:

$$a \rightarrow \langle b\rangle = \langle a\rangle \quad \wedge \quad a \rightarrow (p \triangleleft b) = a \triangleright p \quad \wedge \quad a \rightarrow (p \bowtie q) = a \rightarrow q \bowtie p \tag{20}$$

Exploring the defining equation for prefix sum (19), we can derive a scheme for computing the prefix sum, due to Ladner & Fischer [4]. Misra [5] derived the

base (21) and even (22) cases for powerlists; the derivation of the odd case case (23) can be found in the extended version of this paper [3]. The function *last*, used below, returns the last element of a ParList:

$$ps.\langle a\rangle = \langle a\rangle \tag{21}$$

$$ps.(p \bowtie q) = (0{\rightarrow}t + p) \ \bowtie \ t, \quad \text{where } t = ps.(p + q) \tag{22}$$

$$ps.(p \triangleleft a) = ps.p \ \triangleleft \ (last.(ps.p) + a) \tag{23}$$

## 3 Adder circuits

In [1] Will Adams presented powerlist descriptions for two arithmetic circuits that perform addition on natural numbers: the *ripple carry adder* and the *carry lookahead adder*. The ripple carry adder performs addition as it is first taught in grade school; it is an inheritly sequential method, yielding a linear time method in the number of bits to be added. The carry lookahead adder uses a prefix sum calculation to propagate carries, yielding a method that is logarithmic in the number of bits to be added, in a setting where sufficient parallelism available.

Adams proved that the ripple carry circuit correctly implements addition and that the carry lookahead and the ripple carry circuits are the same function. This result was achieved in the powerlist algebra. In the following we extend the definition of the addition circuits and the equivalence result to the ParList algebra. The derivation of the odd case for the carry lookahead reader and the equivalence between the circuits in the odd case can be found in [3].
The ripple carry adder takes three arguments: the first argument is the carry-in bit and the second and third argument are the two ParLists of bits that are to be added. The result is a pair; the first component of the pair is a ParList containing the result of the addition, and the second component is the carry-out bit from the addition. The following defines *rc*, where (24) and (25) are taken from [1]:

$$rc.b.\langle x\rangle.\langle y\rangle = (\langle (x + y + b) \bmod 2\rangle, (x + y + b) \div 2) \tag{24}$$

$$rc.b.(p \mid q).(r \mid s) = (t, d) \tag{25}$$
$$\text{where } t = u \mid v \ \wedge \ (u, c) = rc.b.p.r \ \wedge \ (v, d) = rc.c.q.s$$

$$rc.c.(p \triangleleft a).(q \triangleleft b) = (u \triangleleft y, x) \tag{26}$$
$$\text{where } x = (a + b + d) \div 2 \ \wedge \ y = (a + b + d) \bmod 2 \ \wedge \ (u, d) = rc.c.p.q$$

The carry lookahead adder has the same signature as the ripple carry adder, except that the elements are taken from the set $\{0,1,\pi\}$, where $\pi$ corresponds to a "propagate" action for the carry-in value to a position. To specify the carry lookahead adder, Adams introduces the associative scalar operators $\bullet$, $\star$ and $\odot$ defined by:

$$x \bullet y = \begin{cases} x \text{ if } x = y \\ \pi \text{ if } x \neq y \end{cases} \qquad x \star y = \begin{cases} y \text{ if } y \neq \pi \\ x \text{ if } y = \pi \end{cases} \qquad x \odot y = \begin{cases} x & \text{if } y \neq \pi \\ \neg y & \text{if } y = \pi \end{cases} \tag{27}$$

$$\text{where } \neg 0 = 1 \ \wedge \ \neg 1 = 0 \ \wedge \ \neg\pi = \pi$$

Adams [1] defines the carry lookahead adder by

$$cl.b.p.q = (t, d) \quad (28)$$
$$\text{where } t = s \odot r \ \wedge \ d = last.s \star last.r \ \wedge \ r = p \bullet q \ \wedge \ s = ps.(b \rightarrow r)$$

where $ps$ is computed using the associative operator $\star$ (that has $\pi$ as its neutral element). Expanding the odd case of the definition of $cl$ we get:

$$cl.c.(p \lhd x).(q \lhd y) = (a, w) \quad (29)$$
$$\text{where } w = u \odot v \ \wedge \ a = last.u \star last.v \ \wedge \ v = (p \lhd x) \bullet (q \lhd y) \ \wedge \ u = ps.(b \rightarrow v)$$

This can be simplified to:

$$cl.c.(p \lhd x).(q \lhd y) \ = \ (t \lhd (d \odot (x \bullet y)), d \star (x \bullet y)) \quad (30)$$
$$\text{where} \quad cl.b.p.q = (t, d)$$

enabling us to prove [3] the equivalence between the adders in the odd case:

$$rc.c.(p \lhd a).(q \lhd b) \ = \ cl.c.(p \lhd a).(q \lhd b) \quad (31)$$

## 4 Conclusion

ParList appears to be an appropriate generalization of the powerlist notation. The powerlist examples presented above had straightforward extensions to the ParList algebra. The set of shared axioms makes it possible to reuse proofs of properties of the corresponding powerlist functions when proving the same properties of ParList functions.

## 5 Acknowledgments

The basic ideas behind the extensions presented in this paper are due to my advisor Jayadev Misra; he shared them with me and encouraged me to develop the ParList theory and to write this paper. Rajeev Joshi had many useful comments to drafts of this paper.

## References

1. Will E. Adams. Verifying adder circuits using powerlists. Technical Report CS-TR-94-02, University of Texas at Austin, Department of Computer Sciences, March 1994.
2. Jacob Kornerup. Mapping a functional notation for parallel programs onto hypercubes. *Information Processing Letters*, 53:153–158, 1995.
3. Jacob Kornerup. Parlists - a generalization of powerlists (extended version). Technical Report CS-TR-97-15, University of Texas at Austin, Department of Computer Sciences, June 1997.
4. Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.
5. Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.