

Mapping a Functional Notation for Parallel Programs onto Hypercubes

Jacob Kornerup¹

*Department of Computer Sciences, The University of Texas at Austin, Austin, TX
78712-1168, USA²*

Abstract

The theory of powerlists was recently introduced by Jayadev Misra [7]. Powerlists can be used to specify and verify certain parallel algorithms, using a notation similar to functional programming languages. In contrast to sequential languages the powerlist notation has constructs for expressing balanced divisions of lists.

We study how *Prefix Sum*, a fundamental parallel algorithm, can be tailored for efficient execution on hypercubic architectures. Then we derive a strategy for mapping most powerlist functions to efficient programs for hypercubic architectures.

Keywords: Program derivation; Parallel algorithms; Functional programming; Programming calculi; Hypercubes; Prefix sum

1 Introduction

The field of parallel algorithm design has become a major area of research over the last decade. However, the field has yet to develop a standard language for expressing these algorithms. The *Powerlist* notation, introduced by Jayadev Misra [7], gives us a succinct representation of a certain class of parallel algorithms, amenable to algebraic proofs of correctness.

In this paper we will study the *prefix sum* problem, by calculating an efficient algorithm for hypercubic architectures and then deriving a strategy for implementing most powerlist functions efficiently on the same class of architectures.

¹This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 003658-219 and by the National Science Foundation Award CCR-9111912.

²E-mail: kornerup@cs.utexas.edu

The focus of this paper is on correct and rigorous derivations; efficiency claims will only be backed by operational reasoning.

2 Powerlists

Functional programming languages typically employ *lists* where the basic constructors (adding or removing a single element) allow for sequential processing of the list elements. The Powerlist notation [7] uses balanced division of lists in order to allow for parallel processing. We will give a brief introduction to the notation, and refer the reader to [7] for further reading.

A *powerlist* is a list of length equal to a nonnegative power of two. The elements of the list are all of the same type and size, either scalars (uninterpreted values from outside the theory) or powerlists themselves. A powerlist with the first 4 natural numbers is written as:

$$\langle 0 \ 1 \ 2 \ 3 \rangle$$

The powerlist data structure is defined inductively in the following way: The smallest powerlist $\langle \alpha \rangle$ contains one element, it is called a *singleton*. Two powerlists of equal length and component type can be combined to form a powerlist of twice the length and the same component type using the operators \bowtie (“zip”) and $|$ (“tie”). Zip produces a powerlist that has alternating elements from its arguments, whereas tie produces a powerlist with the elements from the first argument followed by the elements from the second argument. Both zip and tie preserve the order of the elements from each argument list in the resulting list.

$$\langle 0 \ 1 \ 2 \ 3 \rangle \bowtie \langle 4 \ 5 \ 6 \ 7 \rangle = \langle 0 \ 4 \ 1 \ 5 \ 2 \ 6 \ 3 \ 7 \rangle$$

$$\langle 0 \ 1 \ 2 \ 3 \rangle | \langle 4 \ 5 \ 6 \ 7 \rangle = \langle 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \rangle$$

Any non-singleton powerlist can be written uniquely as the zip of two powerlists and as the tie of two powerlists. Proofs of properties on powerlists are done by structural induction: a property holds for all powerlists if we can show it for singletons, and assuming it holds for u and v we can show it for $u \bowtie v$ (or $u | v$).

There is no way to directly address a particular element of a list in the notation. The only way to access the elements of a list is to break down the list using \bowtie and $|$ as deconstructors.

The law relating zip and tie is called *Commutativity* (Richard Bird [1] calls this property *abides*):

$$(p \mid q) \bowtie (u \mid v) = (p \bowtie u) \mid (q \bowtie v)$$

Functions are defined using pattern matching known from functional programming languages such as ML [5] and MirandaTM [8]. For instance, the function R returns the powerlist where the order of the elements of the argument list are reversed:

$$R.(u \mid v) = R.v \mid R.u$$

$$R.\langle \alpha \rangle = \langle \alpha \rangle$$

R can also be defined using zip as the *deconstructor*:

$$R.(u \bowtie v) = R.v \bowtie R.u$$

Unless we state otherwise all functions defined in this paper act as the identity function on singleton lists, like R above, and we omit this case from the function definition. Similarly we omit the base case in the proofs of a property that holds trivially for functions like the above.

For \oplus a scalar binary operator $p \oplus q$ is defined by the the commutativity laws for \oplus , \bowtie and \mid :

$$(p \mid q) \oplus (u \mid v) = (p \oplus u) \mid (q \oplus v)$$

$$(p \bowtie q) \oplus (u \bowtie v) = (p \oplus u) \bowtie (q \oplus v)$$

and the law for singletons:

$$\langle \alpha \rangle \oplus \langle \beta \rangle = \langle \alpha \oplus \beta \rangle$$

Informally $p \oplus q$ is a powerlist of the same length as p (and q) where the elements are the result of applying \oplus to the corresponding elements of p and q in that order.

Notation

We denote function application by a left associating, infix period and function composition by \circ . The binding power of the operators used is described by the

following table where the lines are listed in decreasing order and operators on the same line have equal binding power:

$$\begin{array}{c}
 * \\
 \cdot \\
 \circ \oplus \\
 \bowtie \mid \\
 =
 \end{array}$$

We will use uppercase letters to denote functions, lowercase letters from the end of the alphabet to denote powerlists, lowercase letters from the beginning of the alphabet to denote scalars, and α and β to denote either a scalar or a powerlist.

3 Hypercubes

Like powerlists, hypercubes only come in sizes that are powers of two. They also share the property that two hypercubes of the same size can be combined into a single hypercube of twice the size. Many commercial supercomputer architectures are based on the hypercube, e.g. Intel iPSC/860 and Thinking Machines CM-2. We will not get into the details of these particular architectures, but rather study abstract hypercubes; nor will we develop an operational model for programs running on hypercubes. This implies that we cannot formally calculate the time complexity of powerlist functions mapped onto hypercubes.

An n -dimensional hypercube can be viewed as a graph with 2^n nodes, each uniquely labeled with an n -bit string. Two nodes are connected by an edge if their labels differ in exactly one position, so each node has n neighbors. We note that the diameter (maximum distance between any two nodes) is n .

We will not quantify the difference in time between neighbor communications and communication between arbitrary nodes on the hypercube. For our purposes communication between neighbors is cheap and communication between non-neighbor nodes is expensive and should be avoided. We consider the mapping of a powerlist function to a hypercube to be *efficient* if each parallel step of a corresponding mapping to a CREW PRAM is equivalent to a constant number of basic operations and communications with neighbors on the hypercube.

Two hypercubes each of size 2^n can be combined and labeled in $n + 1$ different

ways, in an “orderly” fashion, to form a hypercube of size 2^{n+1} , one for each position: connect the nodes from the two cubes with the same label by an edge, and relabel each node to an $n + 1$ bit index by shifting the bits from a fixed position one position to the left. The nodes from the first cube all obtain a zero bit in the fixed position, whereas the nodes from the second cube obtain a one bit.

The hypercube topology is very versatile, most other architectures can be embedded on the hypercube; Leighton [3] shows a number of these embeddings. There is a strong connection between powerlists and hypercubes: label each element of a powerlist of length 2^n , with a bitstring (of length n) representing the position of the element in the list, this element can be mapped to the node with the same label on a hypercube of size 2^n . We refer to this representation as the *standard* encoding. By the construction above, it follows by induction that the zip (tie) of the representation of two lists can be implemented efficiently by combining the representing cubes in the low (high) order bit.

4 Prefix Sum on a hypercube

The prefix sum algorithm is one of the most fundamental parallel algorithms. Given a list of scalars and an associative, binary operator \oplus on these scalars, the prefix sum returns a list of the same length where each element is the result of applying the operator on the elements up to and including the element in that position in the original list. For example, if \oplus is addition over the integers we have: $PS.\langle 3\ 2\ 0\ 5 \rangle = \langle 3\ 5\ 5\ 10 \rangle$

In order to specify the algorithm we assume that the operator \oplus has an identity element 0. The operator $*$ on powerlists shifts the elements of the list one position to the right and adds a zero in the leftmost position (the rightmost element is lost by this operation):

$$\langle a \rangle^* = \langle 0 \rangle$$

$$(u \bowtie v)^* = v^* \bowtie u$$

The prefix sum of a list r , $PS.r$, can be specified [7] as the unique solution to the equation (in z):

$$z : z = z^* \oplus r$$

A well known algorithm for computing the prefix sum is due to Ladner and

Fischer [4]. In the powerlist notation it can be written as [7]:

$$PS.(p \bowtie q) = t^* \oplus p \bowtie t \quad \text{where } t = PS.(p \oplus q)$$

It is a worthwhile exercise for the reader to prove that the Ladner and Fischer algorithm is a solution to the above equation.

Deriving a hypercube algorithm

The standard encoding of powerlists on a hypercube for Ladner and Fischer's algorithm is not efficient, since the $*$ operation cannot be performed efficiently on a hypercube (adjacent elements of the list can be as far apart on the hypercube as its diameter). A similar problem arises with the function R , defined in section 1, that reverses the order of the elements of a list. We will return to this problem in the next section. As noted in the previous section both zip and tie can be performed efficiently on a hypercube using the standard encoding. Thus we can obtain an efficient algorithm by eliminating the $*$ operation.

The defining equation for prefix sum can be generalized to the function F in two arguments:

$$F.p.q = (PS.p)^* \oplus q$$

It follows from the defining equation for $PS.r$ that:

$$PS.r = F.r.r$$

We can now explore the definition of F :

$$\begin{aligned} & F.(p \bowtie q).(u \bowtie v) \\ = & \{ \text{Definition of } F \} \\ & (PS.(p \bowtie q))^* \oplus (u \bowtie v) \\ = & \{ \text{Ladner \& Fischer, where } t = PS.(p \oplus q) \} \\ & ((t^* \oplus p) \bowtie t)^* \oplus (u \bowtie v) \\ = & \{ \text{Definition of } * \} \\ & (t^* \bowtie (t^* \oplus p)) \oplus (u \bowtie v) \\ = & \{ \text{Commutativity } \oplus, \bowtie \} \end{aligned}$$

$$\begin{aligned}
& t^* \oplus u \ \bowtie \ (t^* \oplus p) \oplus v \\
= & \{ \text{Associativity of } \oplus \} \\
& t^* \oplus u \ \bowtie \ t^* \oplus (p \oplus v) \\
= & \{ t = PS.(p \oplus q) \} \\
& (PS.(p \oplus q))^* \oplus u \ \bowtie \ (PS.(p \oplus q))^* \oplus (p \oplus v) \\
= & \{ \text{Definition of } F \} \\
& F.(p \oplus q).u \ \bowtie \ F.(p \oplus q).(p \oplus v)
\end{aligned}$$

This gives the following recursive definition of F :

$$\begin{aligned}
& F.\langle a \rangle.\langle b \rangle = \langle b \rangle \\
& F.(p \bowtie q).(u \bowtie v) = F.(p \oplus q).u \ \bowtie \ F.(p \oplus q).(p \oplus v)
\end{aligned}$$

By using two variables the $*$ operator has disappeared, thus the algorithm can be implemented efficiently on the hypercube. This algorithm is well known in the literature [6], and is considered as part of the folklore; its close connection to the algorithm by Ladner and Fischer is interesting.

5 Mapping Powerlists onto Hypercubes

So far we have studied the standard encoding of powerlists onto hypercubes. As we saw in the previous section, this poses a problem with certain operators on the hypercube, such as the $*$ operator. In this section we will introduce the *reflected Gray coding*. This encoding can be viewed as a domain transformation like the Fourier Transform, transforming the operands into a domain where operations like $*$ can be performed efficiently. We will then study how a class of functions using the fundamental operators can be implemented efficiently under this encoding.

The reflected Gray coding of a list permutes the elements in such a way that neighboring elements in the original list are placed in positions of the coded list whose indices written as a binary string only differ in one position:

$$G.(u \mid v) = G.u \mid G.(R.v)$$

As an example:

$$G.\langle a \ b \ c \ d \ e \ f \ g \ h \rangle = \langle a \ b \ d \ c \ h \ g \ e \ f \rangle$$

The inverse function of G is IG , defined by:

$$IG.(u | v) = IG.u | R.(IG.v)$$

5.1 The Gray coded operators

We will now study how operators in the powerlist notation can be implemented in the Gray coded domain. For a binary operator \dagger and unary operator \natural , this amounts to defining the Gray coded counterparts: \dagger^G and \natural^G with the properties:

$$\begin{aligned} G.u \dagger^G G.v &= G.(u \dagger v) \\ \natural^G(G.u) &= G.(\natural u) \end{aligned}$$

The simplest operator to study is the scalar operator \oplus . We define \oplus^G , the Gray coded version of \oplus by:

$$G.u \oplus^G G.v = G.(u \oplus v)$$

Since G is a permutation function, it can be shown that:

$$G.(u \oplus v) = G.u \oplus G.v$$

There is no point in introducing a special \oplus^G operator since $\oplus^G = \oplus$ from the above.

In order to implement \bowtie under the Gray coded mapping we define the operator \bowtie^G satisfying:

$$G.u \bowtie^G G.v = G.(u \bowtie v)$$

By defining a permutation function E , that is efficient to implement on a hypercube, with the property

$$G.(u \bowtie v) = E.(G.u \bowtie G.v)$$

the complexity of \bowtie^G is the same as that of \bowtie . We will proceed by exploring what properties will be needed of E in order to prove the inductive step for the above equation.

$$E.(G.(u | v) \bowtie G.(p | q))$$

$$\begin{aligned}
&= \{ \text{Definition of } G \} \\
&\quad E.((G.u \mid G.(R.v)) \bowtie (G.p \mid G.(R.q))) \\
&= \{ \text{Commutativity } \bowtie, \mid \} \\
&\quad E.((G.u \bowtie G.p) \mid (G.(R.v) \bowtie G.(R.q))) \\
&= \{ \text{Define } E.(u \mid v) = E.u \mid O.v, \text{ see below} \} \\
&\quad E.(G.u \bowtie G.p) \mid O.(G.(R.v) \bowtie G.(R.q)) \\
&= \{ \text{Induction, see below} \} \\
&\quad G.(u \bowtie p) \mid G.(R.q \bowtie R.v) \\
&= \{ \text{Property of } R \} \\
&\quad G.(u \bowtie p) \mid G.(R.(v \bowtie q)) \\
&= \{ \text{Definition of } G \} \\
&\quad G.((u \bowtie p) \mid (v \bowtie q)) \\
&= \{ \text{Commutativity } \bowtie, \mid \} \\
&\quad G.((u \mid v) \bowtie (p \mid q))
\end{aligned}$$

Two equations were left unproven in the above:

$$\begin{aligned}
E.(u \mid v) &= E.u \mid O.v \\
O.(G.u \bowtie G.v) &= G.(v \bowtie u)
\end{aligned}$$

We can take the first equation as the definition of E , along with $E.\langle a \ b \rangle = \langle a \ b \rangle$, whereas the proof of the second equation is similar to the one given above, yielding the following definition of O :

$$\begin{aligned}
O.((u \mid v) \mid (p \mid q)) &= O.(u \mid v) \mid E.(p \mid q) \\
O.\langle a \ b \rangle &= \langle b \ a \rangle
\end{aligned}$$

$E.(u \bowtie v)$ is the permutation on $u \bowtie v$ that swaps each element of u with index (in u) of odd parity with the element in v with the same index. The two lists are then zipped back together. If the list $u \bowtie v$ is encoded directly on the hypercube, this operation can be performed efficiently by swapping elements among the nodes with this property. It is a simple exercise to show that both E and O are their own inverses (involutions).

The $*$ operator is defined in terms of the fundamental operator \bowtie . We can define the Gray coded equivalent in terms of the Gray coded \bowtie^G operator:

$$(u \bowtie^G v)^{*^G} = v^{*^G} \bowtie^G u$$

This operator can be implemented in constant time on the hypercube, since neighboring elements of the list are neighbors on the hypercube under the Gray coded mapping. It can be proven [2] that:

$$(G.u)^{\star^G} = G.(u^{\star})$$

It can also be proven that \star can be implemented efficiently under Gray coding [2], and thus we have shown that under Gray coding the fundamental operators and some derived operators have efficient implementation on the hypercube.

From this it does not follow that all powerlist functions can be implemented as efficiently on a hypercube as on a CREW PRAM. The Gray coding of a powerlist of length 2^n takes $O(n)$ time, but a powerlist function that takes less than $O(n)$ time on a CREW PRAM, like the function \star , is not implemented efficiently due to the overhead introduced by the Gray coding. However, as shown below, when \star is used in Ladner and Fischers prefix sum algorithm, the Gray coded implementation on a hypercube is efficient.

The exact class of functions that have efficient implementations under the Gray coding has yet to be determined and is the subject of further work.

5.2 Ladner and Fischer revisited

As we observed, properties from the original theory carry over into the Gray coded domain. As an example we revisit the Ladner and Fischer algorithm for prefix sum. Using the Gray coded operators we can define the Gray coded version of Ladner and Fischer's algorithm:

$$PS^G.(p \bowtie^G q) = r^{\star^G} \oplus p \bowtie^G r \quad \text{where } r = PS^G.(p \oplus q)$$

As was the case for the \star operator it can be proven [2]:

$$PS^G \circ G = G \circ PS$$

obtaining an efficient implementation of Ladner and Fischer's algorithm for hypercubic architectures.

6 Conclusion

By using a notation that is free from indexing and other cluttering notions we were able to derive an algorithm for computing the prefix sum efficiently on a hypercube. By using a Gray coded mapping we showed how parts of the powerlist algebra can be efficiently encoded on a hypercube, in particular the Ladner and Fischer algorithm for prefix sum.

Acknowledgement

Without the support and guidance of Jayadev Misra this work could not have been done; he shared his ideas about the powerlist notation from very early on in its development. Al Carruth provided many helpful ideas and comments along the way. Greg Plaxton helped in sorting out the vast body of work in the area. Roland Backhouse (the IPL special issue editor) and two anonymous referees pointed to the weak treatment of the efficiency claims in an earlier version of this paper.

References

- [1] R. S. Bird: “Lectures on Constructive Functional Programming”, in “Constructive Methods in Computer Science”, edited by Manfred Broy, Springer Verlag, 1989.
- [2] J. Kornerup: “Mapping Powerlists onto Hypercubes”, Technical Report TR94-05, Department of Computer Sciences, The University of Texas at Austin, 1994.
- [3] F. T. Leighton: “Introduction to Parallel Algorithms and Architectures”, Morgan Kaufmann Publishers, 1992.
- [4] R. E. Ladner and M. J. Fischer: “Parallel prefix computation”, Journal of the ACM, 27:831–838, 1980.
- [5] R. Milner, M. Tofte and R. Harper: “The Definition of standard ML”, MIT Press, 1990.
- [6] E. W. Mayr and G. Plaxton: “Pipelined Parallel Computations, and Sorting on a Pipelined Hypercube” Technical Report STAN–CS-89-1261, Department of Computer Science, Stanford University, 1989.
- [7] J. Misra: “Powerlists: A Structure for Parallel Recursion”, to appear in ACM TOPLAS, 1994.
- [8] D. Turner: “An overview of Miranda”, ACM SIGPLAN Notices, 21:156–166, 1986.