Mapping Powerlists onto Hypercubes

Jacob Kornerup[*]
Dept. of Computer Sciences
Taylor Hall
The University of Texas at Austin
Austin, TX 78712
E–mail: kornerup@cs.utexas.edu

August 4, 1994

Abstract

The theory of powerlists was recently introduced by Jay Misra. It gives us the ability to specify and verify certain parallel algorithms and connection structures. The notation is similar to sequential functional programming languages (such as Miranda^{TM} [Tur86]) but with constructs for expressing balanced division of lists.

In the first part of this work we study how some known algorithms for the hypercube can be specified succinctly in the powerlist notation. These specifications can then be verified quite succinctly in comparison to the original proofs of the algorithms.

The second part of this work is to study how algorithms written in the powerlist notation can be mapped efficiently onto the hypercube. It turns out that many algorithms have a mapping to the hypercube that is as efficient as mappings to architectures that have all to all connections. This mapping is known in the literature as the Gray code. Operators on these Gray coded powerlists can be implemented efficiently on a hypercube. Algebraically the Gray coding is an isomorphism between powerlists expressions and their Gray coded equivalents.

0 Introduction

In the field of parallel algorithm research most papers focus on improving the previously best known results in terms of either time–complexity or parallel work (parallel time multiplied by number of processors utilized). This is achieved by

[*]This 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.

exhibiting new algorithms along with proofs of correctness and an analysis of their complexity. In these papers it is often the case that the proof of correctness given leaves something to be desired, since the important aspect of the paper as seen by the authors is the analysis of the complexity rather than the proof of correctness. Often these proofs are very operational or use some ill defined formalism. While we do not claim that these algorithms are faulty, although some of them may well be, we do believe that the lack of clear formalisms for expressing and proving these algorithms correct has been an obstacle in producing elegant solutions.

We do not propose to revolutionize the field, but we feel that the powerlist notation is a significant step forward in the process of deriving certain algorithms for parallel architectures. This notation was recently introduced by Jay Misra [Mis94], and acts the foundation for our work.

In this work we will focus on algorithms for hypercube architectures, since there is a very close correspondence between powerlists and abstract hypercubes. We will first study how some known algorithms for the hypercube can be expressed and verified in the powerlist notation, then we turn to the problem of mapping functions written in the powerlist notation to abstract hypercubes.

This work is a direct offspring of discussions I have had with Al Carruth; the proof of lemma 2 was done jointly with him. Without his comments this work would not have reached the stage it is at now. My advisor Jay Misra provided the theory that the work is based on. He shared it with us in a very early stage of development. Thanks to Greg Plaxton, Markus Kaltenbach, Will Adams and Marco Schneider for helpful comments along the way.

1 Powerlists

In this section we summarize the powerlist notation, as developed by Jay Misra [Mis94]. For a more complete description of the notation the reader should look there.

1.0 Definitions

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:

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

A powerlist $\langle\alpha\rangle$ containing one element is called a *singleton*. Unless we state otherwise all functions defined in this paper act as the identity function on singleton lists:

$$K.\langle \alpha \rangle = \langle \alpha \rangle$$

and we omit this from the function definition. Similarly we omit the base case in the proofs of a property that holds trivially for functions like the above.

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 $\mid$ ("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. The order of the elements in the argument lists is preserved in the resulting list for both zip and tie.

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

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

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 $(u \mid v)$).

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 $\mid$ as deconstructors.

The only law relating $\bowtie$ and $\mid$ is called *Commutativity*:

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

Functions are defined using pattern matching known from functional programming languages like Miranda^{TM} [Tur86]. 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 can also be defined using zip as the *deconstuctor*:

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

If $\oplus$ is a scalar binary operator then $p \oplus q$ 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. This can be seen from the commutativity laws for $\oplus$ and $\bowtie$ 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:

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

1.1 Notation

Throughout this paper we will use the proof format of Dijkstra and Scholten [DS90]: lines with formulas alternate with lines consisting of an relational symbol and a hint (inside curly braces) explaining why the relation holds between the formulas. If the hint is just the name of a function, this means that the definition of a function has been used in the step.

We denote function composition by $\circ$ and function application by an infix period. The binding power of the operators used is described by the following table where the lines are listed in decreasing order and operators on a the same line have equal binding power:

$$\begin{array}{c} \star \\ . \\ \circ \quad \oplus \quad \ddagger \quad \dagger \quad \otimes \\ \bowtie \quad | \\ = \end{array}$$

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.

2 Hypercubes

Like powerlists, hypercubes only comes 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. In this work we will not get into the details of these particular architectures, but rather study abstract hypercubes. For instance we will make the assumption that the abstract hypercube we are mapping to has enough nodes. We will not develop a notation to describe hypercubes, instead we will note the close correspondence between hypercubes and powerlists and reason using the theory of powerlists.

A (n–dimensional) hypercube can be viewed as 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. In this work 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–neighboring nodes is expensive, and should be avoided. We consider an algorithm *efficient* if each parallel step consists of a constant number of basic operations and communications with neighbors.

Two hypercubes each of size 2^n can be combined 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 efficiently (in a loose meaning of the word) on the hypercube; Leighton [Lei92] shows a number of these embeddings. The connection between powerlists and hypercubes is even stronger: 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 encoding 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.

3 Hypercube Algorithms in Powerlists

In this section we will show how two algorithms from the literature can be expressed in the powerlist notation: prefix sum and matrix multiplication. To show how a proof of correctness appears we include one for the prefix sum algorithm.

3.0 Prefix Sum

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. In order to specify the problem we assume that the operator $\oplus$ has an identity element 0, if an element with property is not part of the scalar type it is added.

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).

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

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

The prefix sum of a list l, $PS.l$, can be specified [Mis94] as the unique solution to the equation (in z)

$$z:\ z = z^\star \oplus l$$

A well known algorithm for computing the prefix sum is due to Ladner and Fischer [LF80]. In the powerlist notation it can be written as:

$$LF.(p \bowtie q) = (LF.(p \oplus q))^{\star} \oplus p \;\; \bowtie \;\; LF.(p \oplus q)$$

It can be shown that $LF.l$ is a solution to the defining equation for $PS.l$ [Mis94].

The direct mapping of powerlists to a hypercube for the algorithm above is not efficient since the * operation cannot be performed efficiently on a hypercube. We will return to this problem in the next section. Instead we look at another algorithm designed for the prefix sum problem on a hypercube

$$HPS.u = H.(u \otimes u)$$

$$H.(u \bowtie v) = H.(u \dagger v) \;\; \bowtie \;\; H.(u \ddagger v)$$

$$H.(\langle a \rangle \otimes \langle b \rangle) = \langle a \rangle$$

Here $\otimes$ is pointwise pairing and $\dagger$ and $\ddagger$ are the scalar operators defined as follows

$$(x \otimes y) \dagger (z \otimes w) = x \otimes (y \oplus w)$$

$$(x \otimes y) \ddagger (z \otimes w) = (y \oplus z) \otimes (y \oplus w)$$

The correctness of HPS follows from the following lemma, by setting $u = v$

Lemma 0

$H.(u \otimes v) = (PS.v)^{\star} \oplus u$

Proof By induction

Base case:

$H.(\langle a \rangle \otimes \langle b \rangle)$

$=$ { Definition of H }

$\langle a \rangle$

$=$ { $(PS.\langle b \rangle)^{\star} = \langle 0 \rangle$ }

$(PS.\langle b \rangle)^{\star} \oplus \langle a \rangle$

Inductive step (let $t = PS.(p \oplus q)$):

$H.((u \bowtie v) \otimes (p \bowtie q))$

$=$ { Commutativity $\otimes, \bowtie$ }

$H.((u \otimes p) \bowtie (v \otimes q))$

$=$ { H }

$H.((u \otimes p) \dagger (v \otimes q)) \;\; \bowtie \;\; H.((u \otimes p) \ddagger (v \otimes q))$

$=\quad \{\ \dagger \text{ and } \ddagger\ \}$

$\quad H.(u \otimes (p \oplus q)) \ \bowtie\ H.((p \oplus v) \otimes (p \oplus q))$

$=\quad \{\text{ Induction hypothesis }\}$

$\quad t^\star \oplus u \ \bowtie\ t^\star \oplus (p \oplus v)$

$=\quad \{\text{ Associativity of } \oplus\ \}$

$\quad t^\star \oplus u \ \bowtie\ (t^\star \oplus p) \oplus v$

$=\quad \{\ \text{Commutativity } \bowtie, \oplus\ \}$

$\quad (t^\star \bowtie (t^\star \oplus p)) \oplus (u \bowtie v)$

$=\quad \{\text{ Definition } \ ^\star\ \}$

$\quad ((t^\star \oplus p) \bowtie t)^\star \oplus (u \bowtie v)$

$=\quad \{\text{ By the definition of Ladner and Fischer, } t = PS.(p \oplus q)\ \}$

$\quad (PS.(p \bowtie q))^\star \oplus (u \bowtie v)$

End Proof

Note that in the above proof there is usually only one next step to take, dictated by the formula at hand.

3.1 Matrix Multiplication

The following algorithm for matrix multiplication was developed by Dekel, Nassimi and Sahni [DNS81]. The following description of the algorithm is not satisfactory, but it is typical of the descriptions found in the literature. It is included to expose this fact, but also to explain an algorithm that is difficult to understand, even when specified in the powerlist notation.

Given matrices A and B of size $2^n \times 2^n$. The algorithm first replicates the elements on a 2^{3n} node hypercube in the following way:

- The ij'th element of A is replicated to all nodes with label $bin(i); bin(j); bin(x)$ for all x such that $0 \leq x \ \wedge\ x < 2^n$
- The jk'th element of B is replicated to all nodes with label $bin(x); bin(j); bin(k)$ for all x such that $0 \leq x \ \wedge\ x < 2^n$

here ; denotes concatenation of bitstrings and $bin(x)$ is the binary string (of length n) representing x.

At the next step of the algorithm each node multiplies its A value and its B value producing a C value. Then for each bit position between n and $2n-1$, nodes with labels that differ only in that position exchange their C value and add the received value to the C value. After the algorithm has finished the ikth element of $A \times B$ is the C value at node $bin(i); \vec{0}; bin(k)$ where $\vec{0}$ is a string of n zeros.

To describe the matrix multiplication in powerlists we need to expand the notation to higher dimension structures. These structures are also referred to as powerarrays. A powerarray is either a singleton $\langle \alpha \rangle$ or constructed from two similar shaped powerarrays by using a member of one of the two families of combinators $\bowtie_i$ and $|_i$ where i is a natural number. $\bowtie_i$ ($|_i$) zips (ties) its two arguments together along the i'th dimension. Any two different operators from the above families commute. The theory behind powerarrays is beyond the scope of this paper, it is currently being pursued by Will Adams and Jay Misra [AM93].

Assuming that the input matrices are encoded with columns in dimension 0 and rows in dimension 1 we can express the algorithm as follows:

$$
\begin{aligned}
&M.a.b = S.(L.a * H.b)\\
&L.(u \mid_1 v) = (L.u \mid_2 L.v) \mid_0 (L.u \mid_2 L.v)\\
&L.(u \mid_0 v) = L.u \mid_1 L.v\\
&L.\langle a \rangle = \langle a \rangle\\
&H.(u \mid_1 v) = (H.u \mid_1 H.v) \mid_2 (H.u \mid_1 H.v)\\
&H.(u \mid_0 v) = H.u \mid_0 H.v\\
&H.\langle a \rangle = \langle a \rangle\\
&S.(u \mid_1 v) = S.u + S.v\\
&S.(u \mid_0 v) = S.u \mid_0 S.v\\
&S.(u \mid_2 v) = S.u \mid_1 S.v\\
&S.\langle a \rangle = \langle a \rangle
\end{aligned}
$$

The proof of this algorithm is omitted, it is fairly involved since the recursive definition of matrix multiplication is complicated although very regular. It takes 12 proof steps using the definitions of the above functions and commutativity.

4 Mapping Powerlists onto Hypercubes

In this section we will show how fundamental operators on powerlists can be implemented efficiently on a hypercube. As noted above the * operator cannot be implemented efficiently if the standard encoding is used. This is due to the fact that 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.

The technique we will use is to encode the powerlists with a reflected Gray code. This encoding can be viewed as a domain transformation like the Fast Fourier Transform, transforming the operands into a domain where the operations can be performed efficiently. Algebraically it is an isomorphism between the algebra of powerlists and the algebra of Gray coded powerlists.

4.0 Gray Codes

Gray coding is a standard technique in computer science. It was originally developed for coding integers in binary, in order to minimize the effect of corrupted bits in the transmission of integer values across a noisy channel. The inventor was Dr. Frank Gray and in 1953 the method was patented by his employer, Bell Labs (US patent number 2,632,058) [Wil89].

The 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. This description does not define Gray codes uniquely, but the following definition of a permutation of a powerlist is the one usually referred to in the literature:

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

Since G is a permutation function it has an inverse, IG defined by

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

The proof that IG is the inverse of G is straightforward and omitted for the sake of brevity.

4.1 Implementing the Operators

In order to implement a powerlist operator $\bigtriangledown$ on Gray coded lists we need to define an operator $\bigtriangledown^G$ that makes the following diagram commute:

$$\begin{tikzcd}
{} \arrow[r, "\bigtriangledown"] \arrow[d, "{(G,G)}"'] & {} \arrow[d, "G"] \\
{} \arrow[r, "\bigtriangledown^G"'] & {}
\end{tikzcd}$$

Scalar operators are the simplest to implement under the Gray coded mapping. Since G is a permutation function we have:

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

This is the property expressed by the diagram above. There is no point in introducing a $\oplus^G$ operator since we have $\oplus^G = \oplus$ from the above.

4.1.0 Implementing Zip

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

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

To see how $\bowtie^G$ can be implemented efficiently we will define the functions F, O, and E and study their properties. F is defined by:

$$F.((x \bowtie y) \bowtie (p \bowtie q)) = (x \bowtie q) \bowtie (p \bowtie y)$$

$$F.\langle a\ b\rangle = \langle a\ b\rangle$$

We note that F is its own inverse.

Lemma 1

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

Proof By induction (Two base cases: lists of length 2 and 4, induction step based on lists of length 8)

Base cases: lists of length 2 are trivial since both F and G are the identity on these lists)

$G.(F.\langle a\ b\ c\ d\rangle)$

$= \quad \{\ F\ \}$

$G.\langle a\ b\ d\ c\rangle$

$= \quad \{\ G, \bowtie, G\ \}$

$G.\langle a\ c\rangle \bowtie G.\langle b\ d\rangle$

Inductive Step:

$G.(F.(((p \mid q) \bowtie (r \mid s)) \bowtie ((u \mid v) \bowtie (x \mid y))))$

$= \quad \{\ F\ \}$

$G.(((p \mid q) \bowtie (x \mid y)) \bowtie ((u \mid v) \bowtie (r \mid s)))$

$= \quad \{$ Commutativity $\bowtie, \mid$ twice $\}$

$G.(((p \bowtie x) \bowtie (u \bowtie r)) \mid ((q \bowtie y) \bowtie (v \bowtie s)))$

$= \quad \{\ G\ \}$

$G.((p \bowtie x) \bowtie (u \bowtie r)) \mid G.(R.((q \bowtie y) \bowtie (v \bowtie s)))$

$= \quad \{$ Property of R twice, $F\ \}$

$G.(F.((p \bowtie r) \bowtie (u \bowtie x)) \mid G.(F.((R.s \bowtie R.q) \bowtie (R.y \bowtie R.v))$

$= \quad \{$ Induction hypothesis $\}$

$(G.(p \bowtie r) \bowtie G.(u \bowtie x)) \mid (G.(R.s \bowtie R.q) \bowtie G.(R.y \bowtie R.v))$

$=$ { Commutativity $\bowtie$, $|$, Property of R }

$(G.(p \bowtie r) \mid G.(R.(q \bowtie s))) \bowtie (G.(u \bowtie x) \mid G.(R.(v \bowtie y)))$

$=$ { G }

$(G.((p \bowtie r) \mid (q \bowtie s))) \bowtie (G.((u \bowtie x) \mid (v \bowtie y)))$

$=$ { Commutativity $\bowtie$, $|$ }

$(G.((p \mid q) \bowtie (r \mid s))) \bowtie (G.((u \mid v) \bowtie (x \mid y)))$

End Proof

The function F is introduced to prove important facts about the following functions:

$$\begin{array}{ll}
E.(u \mid v) = E.u \mid O.v & \text{for length } u \geq 4 \\
O.(u \mid v) = O.u \mid E.v & \text{for length } u \geq 4 \\
E.\langle a\ b\ c\ d\rangle = \langle a\ b\ d\ c\rangle & \\
O.\langle a\ b\ c\ d\rangle = \langle b\ a\ c\ d\rangle & \\
E.\langle a\ b\rangle = \langle a\ b\rangle & \\
O.\langle a\ b\rangle = \langle a\ b\rangle &
\end{array}$$

$E.(u \bowtie v)$ 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 zip'ed 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.

The fact that both E and O are their own inverses follows from:

Lemma 2

$E.x = (G \circ F \circ IG).x$
$O.x = (G \circ R \circ F \circ R \circ IG).x$

Proof By induction on the length of x

Base case (for length of x equal to 1 and 2 it is trivial)

$(G \circ F \circ IG).\langle a\ b\ c\ d\rangle$

$=$ { IG }

$(G \circ F).\langle a\ b\ d\ c\rangle)$

$=$ { F }

$G.\langle a\ b\ c\ d\rangle$

$(G \circ R \circ F \circ R \circ IG).\langle a\ b\ c\ d\rangle$

$=$ { IG }

$(G \circ R \circ F \circ R).\langle a\ b\ d\ c\rangle$

$=$ { R }

$(G. \circ R \circ F).\langle c\ d\ b\ a\rangle$

$= \quad \{\ G\ \}$

$\langle a\ b\ d\ c\rangle$

$= \quad \{\ E\ \}$

$E.\langle a\ b\ c\ d\rangle$

$= \quad \{\ F\ \}$

$(G\circ R).\langle c\ d\ a\ b\rangle$

$= \quad \{\ R\ \}$

$G.\langle b\ a\ d\ c\rangle$

$= \quad \{\ G\ \}$

$\langle b\ a\ c\ d\rangle$

$= \quad \{\ O\ \}$

$O.\langle a\ b\ c\ d\rangle$

Inductive step:

$(G\circ F\circ IG).(u \mid v)$

$= \quad \{\ IG\ \}$

$(G\circ F).(IG.u \mid R.(IG.v))$

$= \quad \{$ Property of F $\}$

$G.(F.(IG.u) \mid (F\circ R\circ IG).v)$

$= \quad \{\ G\ \}$

$(G\circ F\circ IG).u \mid (G\circ R\circ F\circ R\circ IG).v$

$= \quad \{$ Induction hypothesis $\}$

$E.u \mid O.v$

$= \quad \{\ E\ \}$

$E.(u \mid v)$

$(G\circ R\circ F\circ R\circ IG).(u \mid v)$

$= \quad \{\ IG\ \}$

$(G\circ R\circ F\circ R).(IG.u \mid R.(IG.v))$

$= \quad \{\ R\ \}$

$(G\circ R\circ F).(IG.v \mid R.(IG.u))$

$= \quad \{$ Property of F $\}$

$(G\circ R).((F\circ IG).v \mid (F\circ R\circ IG).u)$

$= \quad \{\ R\ \}$

$G.((R\circ F\circ R\circ IG).u \mid (R\circ F\circ IG).v)$

$= \quad \{\ G\ \}$

$(G\circ R\circ F\circ R\circ IG).u \mid (G\circ F\circ IG).v$

$= \quad \{$ Induction hypothesis $\}$

$O.u \mid E.v$

$= \quad \{\ O\ \}$

$O.(u \mid v)$

End Proof

In the above proof we used the property that $F.(u \mid v) = F.u \mid F.v$. This property is easy to show and its proof is omitted for the sake of brevity.

We are now ready to prove the following theorem relating E and G:

Theorem 0

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

Proof

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

$=$ { Lemma 2 }

$G.(F.(IG \circ G).(u \bowtie v))$

$=$ { IG is inverse of G }

$G.(F.(u \bowtie v))$

$=$ { Lemma 1 }

$G.u \bowtie G.v$

End Proof

The operator $\bowtie^G$ can be implemented efficiently on a hypercube, since E is efficiently implementable on the hypercube and we have:

Lemma 3

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

Proof

$E.(u \bowtie v)$

$=$ { $G.(IG.x) = x$ }

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

$=$ { Theorem 0 }

$G.(IG.u \bowtie IG.v)$

$=$ { $\bowtie^G$ }

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

$=$ { $G.(IG.x) = x$ }

$u \bowtie^G v$

End Proof

4.1.1 Implementing Tie

Just as we introduced $\bowtie^G$ as the operation that made the transformation diagram commute, we can introduce $|^G$ by:

$$G.p \mid^G G.q = G.(p \mid q)$$

It is fairly straightforward to show that if we define

$$FLIP.((u \mid v) \mid (p \mid q)) = (u \mid v) \mid (q \mid p)$$

$$FLIP.\langle x\ y\rangle = \langle x\ y\rangle$$

then we have:

Theorem 1

$$FLIP.(u \mid v) = u \mid^G v$$

For the sake of brevity the proof is omitted.

$FLIP$ is also easy to implement on a hypercube: nodes with a one in the highest bit of the label exchange their value with their neighbor in the next to highest dimension. So $|^G$ can also be implemented efficiently on the hypercube.

4.1.2 Implementing the Star Operator

Just as the other Gray coded operators the Gray coded star operator $\star^G$ is defined by a commuting property:

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

By substituting $IG.u$ for u we get

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

The operator $\star^G$ is efficiently implementable on a hypercube since adjacency is preserved under G and IG. If the length of the powerlist u is known $G.((IG.u)^\star)$ can be implemented efficiently if each node computes the neighbor that has the next value in the Gray code sequence and sends the powerlist element on to that node. It can be proven that

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

This is the same equation as the one defining *, except that all operators are substituted with their Gray coded counterparts. This follows from an unproven conjecture that functions defined exclusively with the Gray coded operators obey the same laws as their unencoded counterparts. This claim is supported by the observation below.

4.2 Generalizing the Mapping

As we observed, properties from the original theory seem to 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 Fischers algorithm:

$$LF^G.(p \bowtie^G q) = (LF^G.(p \oplus q))^{\star^G} \oplus p \;\; \bowtie^G \;\; LF^G.(p \oplus q)$$

The following theorem shows the correspondence between the two versions of the algorithm:

Theorem 2
$LF^G \circ G = G \circ LF$

Proof Induction, base case is omitted:

$$
\begin{aligned}
& (LF^G \circ G).(p \bowtie q)) \\
= \quad & \{ \bowtie^G \} \\
& LF^G.(G.p \bowtie^G G.q) \\
= \quad & \{ LF^G \} \\
& (LF^G.(G.p \oplus G.q))^{\star^G} \oplus G.p \;\; \bowtie^G \;\; LF^G.(G.p \oplus G.q) \\
= \quad & \{ \oplus \text{ is scalar} \} \\
& (LF^G.(G.(p \oplus q)))^{\star^G} \oplus G.p \;\; \bowtie^G \;\; LF^G.(G.(p \oplus q)) \\
= \quad & \{ \text{Induction hypothesis} \} \\
& (G.(LF.(p \oplus q)))^{\star^G} \oplus G.p \;\; \bowtie^G \;\; G.(LF.(p \oplus q)) \\
= \quad & \{ \star^G \} \\
& G.((LF.(p \oplus q))^\star) \oplus G.p \;\; \bowtie^G \;\; G.(LF.(p \oplus q)) \\
= \quad & \{ \oplus \text{ is scalar} \} \\
& G.((LF.(p \oplus q))^\star \oplus p) \;\; \bowtie^G \;\; G.(LF.(p \oplus q))
\end{aligned}
$$

$= \quad \{ \bowtie^G \}$

$\quad G.((LF.(p \oplus q))^{\star} \oplus p \ \bowtie \ LF.(p \oplus q))$

$= \quad \{ \ LF \ \}$

$\quad (G \circ LF).(p \bowtie q)$

End Proof

The Ladner and Fischer Prefix sum algorithm can be implemented efficiently on a hypercube since all the operators in the definition of LF^G can be implemented efficiently.

It is worth noting how mechanical the above proof is. It uses the commuting property between G and the Gray coded operators. With a general argument using these facts one should be able to prove that any function defined in terms of the Gray coded operators will also have the commuting property. This proof would be a structural induction proof on the structure of the functions that can be defined using the Gray coded operators. At the present this proof is not complete. We have not yet completed the the study of the algebraic structure of the two involved domains.

5 Related Work

The work by Mou and Hudak [MH88] introduces an algebraic programming notation that is more general than the powerlist notation. This notation was developed more as a programming notation than for doing proofs of correctness. It has been implemented on a hypercube (Connection Machine), and this implementation is worth studying in order to assess how difficult it would be to implement the powerlist notation on an actual hypercube architecture.

The hypercube is a member of a larger class of *hypercubic* architectures [Lei92] that includes the Butterfly, Shuffle–exchange graph [Sto71], and Cube Connected Cycles [PV81]. It is worthwhile to study whether functions written in powerlists can be implemented efficiently on these architectures.

The powerlist notation is similar to RUBY [JS90] a notation to specify VLSI circuits. RUBY can be used to transform a VLSI design into an equivalent systolic design. There should be a wealth of knowledge that can be integrated in the theory of powerlists.

6 Future Work

Many algorithms for the hypercube can be found in the literature. Most are being presented as "results": the complexity measure is better than previously published algorithms. Unfortunately, since the emphasis of these papers is to show that this holds, little emphasis is made on presenting the algorithm and its proof of correctness. We believe that for certain algorithms the powerlist notation is an elegant way to present the algorithm and its proof of correctness.

Among the algorithms from the literature we propose to study are:

- Batchers sorting algorithm for the hypercube [Bat68]
- Cubesort, an algorithm that generalizes Batchers sorting algorithm and improves the time complexity [CS92]
- Graphsort, a sorting algorithm for the hypercube that utilizes Gray coding [Gor89]

It is an obvious extension of the above work to examine whether the rest of the powerlist operators can be implemented efficiently on the hypercube, if the arguments are transformed using the Gray coding. If this is the case then we have a general translation scheme for mapping functions written in the powerlist notation onto the hypercube.

In this paper we have avoided giving a formal model of the hypercube. This led to some vague operational reasoning on how certain operators can be implemented on the hypercube. This is highly unsatisfactory, instead a formal model of the hypercube is needed. One way to approach this is to view the hypercube as a certain class of synchronous UNITY programs [CM88]. We can then prove formally that the operators indeed do as claimed, using existing techniques [CM88], [Kor91].

A formal model of the hypercube would give us a way to deal with the assumption that the hypercube has more nodes than the lists we are mapping to. This assumption is unrealistic, but it can be overcome by assigning logical hypercubes to each physical node in such a way that there are enough nodes. However, this might not work for an arbitrary function definition, as it might not be possible to statically determine how large the involved powerlists will be.

7 Literature

AM93 W. Adams and J. Misra: "Powerarrays", unpublished manuscript, 1993.

Bat68 K. Batcher: "Sorting networks and their applications", in Proceedings Joint Computer Conference, volume 32: pages 307–314, 1968.

CM88 K. M. Chandy, J. Misra: "Parallel Program Design – A Foundation", Addison–Wesley, 1988.

CS92 R. Cypher, J. L. C. Sanz: "Cubesort: A Parallel Algorithm for Sorting N Data items with S–Sorters", Journal of Algorithms 13:211–234, 1992

DNS81 E. Dekel, D. Nassimi and S. Sahni: "Parallel matrix and graph algorithms", SIAM Journal on Computing, 10,4:657–675 1981.

DS90 E. W. Dijkstra and C. S. Scholten: "Predicate Calculus and Program Semantics", Springer–Verlag, 1990.

Gor89 D. M. Gordon: "Parallel Sorting on Caley Graphs", Preprint, Department of Computer Scince, University of Georgia, 1989.

JS90 G. Jones and M. Sheeran: "Circuit Design in Ruby", in Jørgen Staunstrup, editor, Formal Methods for VLSI Design, North–Holland, 1990.

Kor91 J. Kornerup: "Verifying Synchronous Programs", Technical Report, Department of Computer Science, Technical University of Denmark, 1991.

Lei92 F. Thompson Leighton: "Introduction to Parallel Algorithms and Architectures", Morgan Kaufmann Publishers, San Mateo, California, 1992.

LF80 R. E. Ladner and M. J. Fischer: "Parallel prefix computation", Journal of the ACM, 27:831–838, 1980.

MH88 Z. G. Mou and P. Hudak: " An Algebraic Model for DIVIDE–and–Conquer and its Parallelism", Journal of Supercomputing 2:257–278, 1988.

Mis94 J. Misra: "Powerlists: A Structure for Parallel Recursion Preliminary Version)", in "A classical Mind: Essays in Honour of C.A.R. Hoare", edited by A.W. Roscoe, p295-316, Prentice Hall International, 1994

PV81 F. P. Preparata and J. Vuillemin: "The Cube–Connected Cycles: A Versatile Network for Parallel Computation", CACM 24,5:300–309, 1981.

Sto71 H. S. Stone: "Parallel processing with the perfect shuffle", IEEE Transactions on Computers C–20, no. 2: 153-161, 1971.

Tur86 D. Turner: "An overview of Miranda", ACM SIGPLAN Notices, 21:156–166, 1986.

Wil89 H. S. Wilf: "Combinatorial Algorithms: An Update" CBMS–NSF Regional Conference Series in Applied Mathematics, SIAM 1989