Lecture 29 Hypercubes and the Gray Representation

In this lecture we will take an algebraic approach to routing on the hypercube. We will develop some algebraic tools, which we will then use to analyze the hypercube implementation of parallel prefix described in the last lecture.

Let $\mathcal{Z}_2$ be the field of integers mod 2. The field $\mathcal{Z}_2$ has 2 elements $\{0, 1\}$. Its multiplication operation is the same as Boolean $\wedge$, and its addition operation is the same as Boolean exclusive-or.

Let $\mathcal{Z}_2[x]$ denote the ring of univariate polynomials with coefficients in $\mathcal{Z}_2$. A typical element of this ring is $1 + x + x^2 + x^5 + x^8 + x^9$. Note that all coefficients are either 0 or 1, and $+$ and $-$ are the same thing, since $1 = -1$ in $\mathcal{Z}_2$.

Now we take the elements of $\mathcal{Z}_2[x]$ modulo the polynomial x^n to get the quotient ring $\mathcal{Z}_2[x]/x^n$. This is just like asserting that $x^n = 0$. It implies that $x^m = 0$ for all $m \geq n$, since $x^m = x^n \cdot x^{m-n} = 0 \cdot x^{m-n} = 0$. Elements of $\mathcal{Z}_2[x]/x^n$ are thus polynomials of degree $n - 1$ or less, and there are exactly 2^n such polynomials, the same number as bit strings of length n. We therefore identify bit strings of length n and elements of $\mathcal{Z}_2[x]/x^n$ under the one-to-one correspondence

$$a_0 a_1 \cdots a_{n-1} \quad \mapsto \quad \sum_{i=0}^{n-1} a_i x^i \ .$$

For example, for $n = 5$, the bit string 10011 corresponds to the polynomial $1 + x^3 + x^4$. (*Warning*: the *least* significant bit in the binary representation

of a number is the coefficient of the *highest* degree term in the corresponding polynomial.)

Under this correspondence, shifting right one bit corresponds to multiplication by x in $\mathcal{Z}_2[x]/x^n$, and componentwise exclusive-or corresponds to addition in $\mathcal{Z}_2[x]/x^n$. Thus the procedure for converting from binary to Gray (shift right and exclusive-or with the original) corresponds to multiplying by $1+x$. In other words, if b_i and g_i are the polynomials in $\mathcal{Z}_2[x]/x^n$ corresponding to the binary and Gray representations of i respectively, then

$$\begin{aligned} g_i &= b_i + xb_i \\ &= (1+x)b_i \ . \end{aligned}$$

As we mentioned in the last lecture, this operation is invertible. Recall that to convert Gray to binary, we calculate the kth bit of the binary representation by taking the mod 2 sum of the kth bit of the Gray representation and all bits to its left. Algebraically, this corresponds to the fact that the polynomial $1+x$ has a multiplicative inverse in $\mathcal{Z}_2[x]/x^n$, namely $1+x+x^2+\cdots+x^{n-1}$:

$$\begin{aligned} &(1+x)\cdot(1+x+x^2+\cdots+x^{n-1}) \\ &\quad= (1+x+\cdots+x^{n-1})+(x+x^2+\cdots+x^n) \\ &\quad= 1+(x+x)+(x^2+x^2)+\cdots+(x^{n-1}+x^{n-1})+x^n \\ &\quad= 1+x^n \quad \text{since } q+q=0 \\ &\quad= 1 \quad \text{since } x^n=0. \end{aligned}$$

The procedure for converting from Gray to binary then corresponds to multiplication by $1+x+x^2+\cdots+x^{n-1}$. (In fact, an element of $\mathcal{Z}_2[x]/x^n$ is invertible iff its constant coefficient is 1. The inverse of $1+xp$ is $\sum_{i=0}^{n-1} x^i p^i$.)

In the kth stage of the parallel prefix circuit, we pass messages from node i to node $i+2^k$. The distance between these nodes on the hypercube is the number of bits on which g_i and g_{i+2^k} differ. This is often called the *Hamming distance*. We now show that the Hamming distance between the Gray representations of i and $i+2^k$ is 2 if $k \geq 1$ and 1 if $k=0$.

Let e_{ik} be the degree of the highest power of x that divides $b_i + b_{i+2^k}$. The significance of e_{ik} is that it measures the distance that the carry propagates when adding 2^k to i in binary. Specifically, the binary representation of 2^k has a 1 in bit position $n-k-1$ and 0 elsewhere. A carry is propagated to the left of the $n-k-1^{\text{st}}$ bit position as long as we see a 1 in b_i. The carry stops at the first bit position to the left of $n-k$ at which b_i contains a 0, and e_{ik} is that bit position (counting from the left and starting at 0), or 0 if no such position exists. The exclusive-or of the bit strings b_i and b_{i+2^k} is of the form $\cdots 00011111000 \cdots$, with 1 in bit positions e_{ik} through $n-k-1$ inclusive and 0 elsewhere.

In terms of the polynomial representation,

$$b_i + b_{i+2^k} \quad = \quad \sum_{j=e_{ik}}^{n-k-1} x^j \; .$$

Converting to Gray, we have

$$\begin{aligned} g_i + g_{i+2^k} &= (1+x)b_i + (1+x)b_{i+2^k} \\ &= (1+x) \cdot (b_i + b_{i+2^k}) \\ &= (1+x) \sum_{j=e_{ik}}^{n-k-1} x^j \\ &= x^{e_{ik}} + x^{n-k} \; . \end{aligned}$$

This says that the Gray representations of i and $i + 2^k$ differ only in bits e_{ik} and $n - k$. In the case $k = 0$, they differ only in bit e_{ik}, since $x^n = 0$. For example,

$$\begin{array}{rcl} b_i & = & 1001011101111111100010100111 \\ b_{2^k} & = & 0000000000000001000000000000 \\ b_{i+2^k} & = & 1001011110000000100010100111 \\ b_i + b_{i+2^k} & = & 0000000011111111000000000000 \\ g_i + g_{i+2^k} & = & 0000000010000000100000000000 \\ & & \qquad\qquad\;\; \uparrow \qquad\quad\; \uparrow \\ & & \qquad\qquad\;\; e_{ik} \qquad n-k \end{array}$$

We have shown that the Hamming distance between g_i and g_{i+2^k}, and hence the routing distance on the hypercube between processor i and $i + 2^k$, is at most 2. Thus in each stage of our parallel prefix circuit, messages must be passed a distance of at most 2. However, we still need to show how to route the messages so as to avoid collisions.

Let us use the following protocol. At stage 0, processor i passes its value to processor $i + 1$. Processor i can compute the Gray representation of the destination processor by flipping bit e_{i0} of its own Gray representation. There are no collisions, since $i \mapsto i + 1$ is a Hamiltonian circuit.

Subsequently, in stage k, messages are passed from i to $i + 2^k$ in two rounds. In the first round, each processor i of even parity flips bit $n - k$ of its Gray representation and sends its message to the processor with that Gray representation. (The *parity* of i is the low order bit of b_i, *i.e.* the coefficient of x^{n-1}, or the mod 2 sum of the bits of g_i.) Each processor i of odd parity flips bit e_{ik} of its Gray representation and sends its message to the processor with that Gray representation. In the second round, those processors receiving the messages flip the remaining bit and forward the messages to their final destinations.

There is no collision along wires, *i.e.* no messages are sent from i to j and simultaneously from j to i, because any two nodes with a direct connection in

the hypercube have different parity, and if i and j are of different parity then they are flipping different bits in each of the two rounds, so the two messages cannot be traveling along the same wire at the same time.

However, it is still conceivable that messages might collide at a vertex, *i.e.* i and j might both pass to ℓ in the first round. We show that this cannot happen either. If i and j are of different parity, then the messages are going to processors of different parity. If i and j are of the same parity, then either in round 1 or round 2 the $n - k^{\text{th}}$ bit is being flipped in both transmissions, and this is a one-to-one map.

Hypercube embeddings and message routing are an active topic of research. For more information and references, see [54, 55, 104].

Lecture 30 Integer Arithmetic in NC

30.1 Integer Addition

Addition of two n-bit binary numbers can be performed in $\log n$ depth with n processors. We will use parallel prefix to calculate the carry string. Once the carry is computed, the sum is easily computed in constant time with n processors by taking the exclusive-or of the two summands and the carry string.

The carry string is defined as follows:

- The lowest order carry bit is always 0.
- If the ith bits of the two summands (counting from the right) are both 0, then the $i+1^{\text{st}}$ bit of the carry will be 0, irrespective of the ith bit of the carry.
- If the ith bits of the two summands are both 1, then the $i+1^{\text{st}}$ bit of the carry will be 1, irrespective of the ith bit of the carry.
- If the ith bits of the two summands are 0 and 1, then the $i+1^{\text{st}}$ bit of the carry will be the same as the ith bit of the carry. In this case we say that the carry is *propagated* from i to $i+1$.

To compute the carry using parallel prefix, we will use a three element algebra $\{0, 1, p\}$ with associative binary operation $\cdot$ defined below. Intuitively, the

element 0 means, "carry 0", the element 1 means "carry 1", and the element p means, "propagate the carry from the previous bit position".

The binary operation $\cdot$ is defined by the following table:

$\cdot$	0	1	p
0	0	0	0
1	1	1	1
p	0	1	p

In other words, for any $x \in \{0, 1, p\}$,

$$\begin{aligned} 0 \cdot x &= 0 \\ 1 \cdot x &= 1 \\ p \cdot x &= x \,. \end{aligned}$$

Note that $\cdot$ is associative but not commutative: $0 \cdot 1 = 0$ but $1 \cdot 0 = 1$.

Let u be a string over $\{0, 1, p\}$ with a 0 in position 0, a 0 in position $i+1$ if the ith bits of the two summands are both 0, a 1 in position $i+1$ if the ith bits of the two summands are both 1, and a p in position $i+1$ if one of the ith bits of the two summands is 0 and the other is 1. The string u can be computed in constant time from a and b with n processors. The carry string is obtained by computing the suffix products of u.

Example 30.1 Let $a = 100101011101011$ and $b = 110101001010001$. The string u over $\{0, 1, p\}$ for these two numbers, the carry string obtained as the suffixes of u, and the binary sum are as illustrated.

$$\begin{array}{rcr} u &=& 1p01010p1ppp0p10 \\ \mathit{carry} &=& 1001010110000110 \\ a &=& 100101011101011 \\ b &=& 110101001010001 \\ \hline \mathit{sum} &=& 1011010100111100 \end{array}$$

□

30.2 Integer Multiplication

Consider a multiplication problem involving two n-bit binary numbers. The grade school algorithm for multiplication gives n partial sums, which then can

be added to get the product. For example,

$$
\begin{array}{r}
101101 \\
\times\ 101011 \\
\hline
101101 \\
101101 \\
000000 \\
101101 \\
000000 \\
+\ 101101 \\
\hline
11110001111
\end{array}
\tag{37}
$$

This can be done in time $O((\log n)^2)$ with $O(n^2)$ processors in a straightforward way. First compute all the bits of the partial sums, then add the partial sums in pairs in a tree-like fashion. It takes constant time to compute the partial sums with $O(n^2)$ processors, and $O((\log n)^2)$ to do the additions.

By being slightly more clever, we can reduce the time to $O(\log n)$ by reducing the problem of adding three n-bit binary numbers to adding two $n+1$-bit binary numbers. Look at the partial sums obtained by adding each 3-bit column individually:

$$
\begin{array}{r}
101100111 \\
101011100 \\
+\ 101111101 \\
\hline
10 \\
01 \\
11 \\
10 \\
10 \\
10 \\
11 \\
00 \\
+\ 11 \\
\hline
\end{array}
$$

Rearranging, we get

$$
\begin{array}{r}
10 \\
01 \\
11 \\
10 \\
10 \\
10 \\
11 \\
00 \\
+\ 11 \\
\hline
\end{array}
\quad \longrightarrow \quad
\begin{array}{r}
101111101 \\
+\ 101000110 \\
\hline
\end{array}
$$

Thus, in constant time we have reduced the problem of adding three binary numbers to adding two binary numbers. To apply this to the multiplication problem (37), we partition the partial sums into sets of three and perform this step in parallel for all the sets. This reduces the problem of adding n numbers to the problem of adding $\frac{2n}{3}$ numbers. We repeat this step until we have only two numbers, then we just add them using the $O(\log n)$ time addition algorithm described above. After the first stage, we have $\frac{2}{3}n$ numbers; after the second stage, $(\frac{2}{3})^2 n$, and so on. The number of numbers decreases geometrically, thus there are only $O(\log n)$ stages. Each stage takes $O(1)$ time and $O(n^2)$ processors.

30.3 Integer Division

We wish to do integer division with remainder in NC. That is, given binary numbers s and t, compute the unique quotient q and remainder r such that $s = qt + r$ and $0 \leq r < t$.

Our algorithm is based on *Newton's method*, a useful technique for approximating roots of differentiable functions. Newton's method works as follows. Starting from an initial guess x_0, compute a sequence of approximations

$$x_{i+1} \quad = \quad x_i - \frac{f(x_i)}{f'(x_i)} , \tag{38}$$

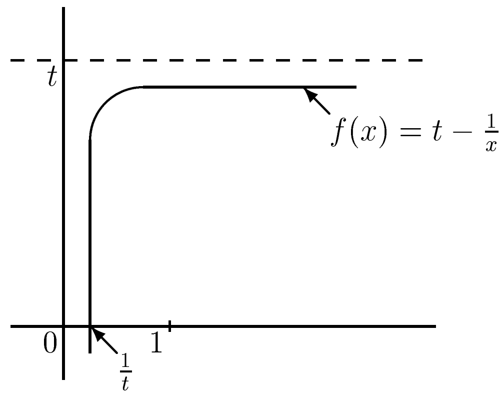
where $f' = df/dx$. For real-valued functions of a real variable, this is equivalent to finding the line tangent to the curve $y = f(x)$ at x_i and taking x_{i+1} to be the point where that line intersects the x axis.

In general, Newton's method is not guaranteed to converge to a root. However, if the function is well-behaved and the initial guess x_0 is close enough to a root, then the method converges very quickly: the number of bits of accuracy roughly doubles with each iteration. In this application, although we are using an approximation technique, we will be using only exact binary arithmetic (no floating point), and will obtain an exact solution.

We first show how to approximate the reciprocal $\frac{1}{t}$ of a given number t in binary. We will do this by approximating the root of the function

$$f(x) \quad = \quad t - \frac{1}{x}$$

using Newton's method.



In this case, $f'(x) = x^{-2}$, and (38) becomes

$$x_{i+1} = 2x_i - t{x_i}^2 .$$

We take as our first approximation x_0 the unique fractional power of 2 in the interval $(\frac{1}{2t}, \frac{1}{t}]$. This can be found in $O(\log n)$ time by finding the unique power of 2 in the interval $[t, 2t)$ and taking its reciprocal by reversing the order of the binary digits and placing a binary point after the first 0. We then iterate Newton's method to get the sequence of approximations $x_0, x_1, x_2, \ldots$. These approximations blast in toward $\frac{1}{t}$ quickly: we start with an error of at most $\frac{1}{2t}$, and at each step we roughly square the error, thus doubling the number of bits of accuracy. This is called *quadratic convergence.*

Lemma 30.2 *The sequence $x_0, x_1, \ldots$ obtained from Newton's method is non-decreasing and converges quadratically to $\frac{1}{t}$.*

Proof. By definition,

$$\frac{1}{2t} < x_0 \leq \frac{1}{t} ,$$

or in other words,

$$0 \leq 1 - tx_0 < \frac{1}{2} .$$

For $i \geq 0$,

$$\begin{aligned} 1 - tx_{i+1} &= 1 - t(2x_i - t{x_i}^2) \\ &= (1 - tx_i)^2 . \end{aligned}$$

It follows by induction that

$$\begin{aligned} 1 - tx_i &= (1 - tx_0)^{2^i} \\ &< 2^{-2^i} , \end{aligned}$$

thus

$$\frac{1}{t} - x_i \quad < \quad \frac{1}{2^{2^i} t} \; .$$

From these facts we can conclude that

$$\frac{1}{2t} < x_0 \leq x_1 \leq x_2 \leq \cdots \leq \frac{1}{t} \; .$$

$\square$

After $k = \lceil \log \log \frac{s}{t} \rceil$ iterations we have

$$1 - t x_k \quad < \quad \frac{t}{s} \; .$$

From this and the fact that $x_k \leq \frac{1}{t}$ we have that

$$0 \quad \leq \quad \frac{s}{t} - s x_k \quad < \quad 1 \; .$$

Therefore the desired integer part of $\frac{s}{t}$ is either $\lfloor s x_k \rfloor$ or $\lceil s x_k \rceil$, and the remainder can be found by subtracting.

Each Newton iteration took $O(\log n)$ time (we did not do enough iterations to let the numbers get too big) and we needed $\log \log \frac{s}{t} = O(\log n)$ iterations.

For some interesting ramifications of the division problem, including an $O(\log n)$-depth circuit for integer division under a slightly weaker uniformity condition, see [9].