Fixed Connection Networks

Hypercubes and Hypercubic Networks

Algorithms and Embeddings

Definition. An embedding of a graph $G = (V, E)$ into a graph $G' = (V', E')$ is a function ϕ from V to V'.

Definition. The dilation of the embedding ϕ is defined as follows. $dil(\phi) = max\{dist(\phi(u), \phi(v)) : (u, v) \in E\}$, where $dist(a, b)$ is the distance in edges between $a, b \in V'$.

Claim 1. A 1d-array can be embedded into a 2d-array with dilation 1.

Claim 2. A ring can be embedded into a 2d-array with dilation 1, if and only if the number of vertices of the array is even.

Fact 3. In a 2d-array the number of vertices which are within distance k from any vertex is at most $2k^2 + 2k + 1$.

Claim 4. A complete binary tree can not be embedded into a 2d-mesh with dilation 1 for any $n > 4$.

Proof. A binary tree of depth k has $2^{k+1} - 1$ vertices. On a 2d-mesh, within distance k from any vertex, there are at most $2k^2 + 2k + 1$ vertices. As $2^{k+1} - 1 > 2k^2 + 2k + 1$, $k > 4$, it is evident that no such embedding exists.

The **Hypercube** is the most versatile network. It can simulate one step of an $O(n)$-cell array, binary tree, mesh of trees in $O(1)$ time i.e. all algorithms examined so far can be implemented on the hypercube. The hypercube is a good choice for the interconnection network of a parallel computer. The only problem with such a choice is its degree $O(\lg n)$ as opposed to $\Theta(1)$ for most other networks. Derivative networks, also known as hypercubic networks, do not suffer from those problems (butterfly, de-Bruijn graph, cube-connected cycles, shuffle-exchange).

The n-dimensional hypercube has $N = 2^n$ vertices and $N \lg N/2$ edges. Two vertices are connected by an edge if they differ in *exactly one* bit position. Let $u = u_1 u_2 \ldots u_i \ldots u_n$ be a hypercube vertex. An edge is a dimension i edge if it links two vertices that differ in the i-th bit position. This way vertex u is connected to vertex $u^i = u_1 u_2 \ldots \bar{u_i} \ldots u_n$ with a dimension i edge. A path from vertex u to vertex v can be determined by correcting the bits of u to agree with those of v starting from dimension 1 in a "left-to-right" fashion. The bisection width of the hypercube is $bw = N/2$. This is a result of the following property of the hypercube. If all edges of dimension i are removed from an n dimensional hypercube, we get two hypercubes each one of dimension $n-1$.

Definition. On an n vertex graph a **perfect matching** is a set of $n/2$ edges that do not share any vertices.

Definition. On an n vertex graph a Hamiltonian cycle is a cycle of length n so that each vertex of the graph is touched by the cycle exactly once.

The dimension i edges of the hypercube form a perfect matching. Moreover, the removal of all edges of dimension i splits an n dimensional hypercube into two dimension $n-1$ hypercubes.

Question How many vertices does one need to remove to split the hypercube into two parts of equal size?

Theorem 1 An N-cell linear array (with wrap-around edges, a.k.a ring) is a *subgraph* of any N-vertex hypercube (i.e it contains a Hamiltonian path that traverses all vertices exactly once) for any $N \geq 4$.

Proof. (by induction) Base case: $N = 4$ true by inspection. Assume inductive hypothesis is true. Take an N-vertex hypercube. Remove dimension n edges. It is split into two hypercubes of dimension $n-1$. By induction, construct two identical Hamiltonian paths/cycles for the two halves. Let $a_1 a_2 x \ldots y a_1$, $a'_1 a'_2 x' \ldots y' a'_1$ be the two paths/cycles. Then, construct the following cycle $a_1 a'_1 y' \ldots x' a'_2 a_2 x \ldots y a_1$.

If one traces a hamiltonian cycle a gray code is formed. Formally

Definition An n-bit Gray code is an ordering of all n-bit binary numbers so that consecutive numbers differ in precisely one bit position.

Theorem 2 The $2^{d_1} \times 2^{d_2} \times \ldots \times 2^{d_r}$-cell r dimensional array is a subgraph of the $2^{d_1+\cdots+d_r}$-vertex hypercube.

Proof. (by construction) Map each 2^{d_i}-cell array to a d_i-dimension hypercube. Let g_i be this mapping. Map cell $(i_1, i_2, \ldots i_r)$ of the array to a $d_1 + \ldots d_r$-long binary string $g(i_1) \ldots g(i_r)$, where $g(i_i)$ is a binary string of d_i bits.

Theorem 3 The $N-1$ vertex complete binary tree **cannot** be embedded in the N-vertex hypercube, $N \geq 8$.

Proof. (by contradiction)

Obs 1. In an n-bit binary sequence the number of strings with an odd number of 1's is equal to the number of strings with an even number of 1's.

Let us assume that such an embedding is possible. Let the root r of the tree (a level 0 vertex) be mapped to some vertex v of the hypercube. Let the parity of v be even (i.e. its string contains an even number of 1's). Adjacent vertices of v in the hypercube must have oppositive parity (ie. odd) as two vertices are adjacent if they differ in exactly one bit (if this bit is 1 in v it must be 0 in its neighbor, a decrease by one of the 1's, if it is 0 in v it must be 1 in its neightbor, an increase by one of the 1's, i.e. odd parity for the neighbor of v in any of the two cases).

If r (level 0) is mapped to an even-parity vertex v, then level-1 vertices of the tree are mapped to odd-parity vertices in the hypercube (neighbors in the hypercube of even-parity vertices). Similarly level-2 vertices in the tree are mapped to even-parity vertices in the hypercube and so on. Let us consider the parity of the leaves of the tree. Let it be odd. Then so is the parity of their grandparents. On an $n-1$-vertex binary tree the number of leaves and their grandparents is $N/2 + N/8$. These odd-parity vertices are mapped to odd parity vertices in the hypercude. Therefore the n-dimension hypercube contains at least $N/2 + N/8 = 5N/8$ odd-parity vertices and at most $3N/8$ even parity vertices contradicting to the observation.

Theorem 4 An N-vertex double-rooted complete binary tree is a subgraph of an N-vertex hypercube.

Proof. The reason the previous construction didn't work was because two vertices, one of the left subtree of the root and one on the right subtree of the root were mapped to same parity vertices of the hypercube. Were they mapped to opposite parity vertices a contradiction would not have been possible. The introduction of double-rooted trees allows such a mapping. In such a tree the two roots would be mapped to opposite parity vertices and therefore two vertices at distance i from roots r_1 and r_2 respectively are mapped to vertices of opposite parity. This intuitive argument is proved by induction below.

Example of embedding

Corollary 1 $N/2$ vertex complete binary tree can be embedded into an N vertex hypercube.

Claim An $N \times \ldots \times N$ mesh of doubly rooted trees is a subgraph of the $(2N)^r$-vertex hypercube.

The set of vertices of a butterfly is represented by (w, i), where w is a binary string of length n and $0 \leq i \leq n$. Therefore $|V| = (n+1)2^n = (\lg N + 1)N$. Two vertices (w, i) and (w', i') are connected by an edge if $i' = i + 1$ and either (a) $w = w'$ or (b) w and w' differ in the i' bit. As a result $|E| = O(N \lg N)$, $d = 4$, $D = 2 \lg N = 2n$, and $bw = N$. Vertices with $i = j$ are called level-j vertices. If we remove the vertices of level 0 we get two butterflies of size $N/2$. If we collapse all levels of an n dimensional butterfly into one, we get a hypercube. If we remove the vertices of the last level we get two interleaved butterflies of size $N/2$ (number of vertices per level). A wrapped butterfly is obtained by collapsing the first and the last levels into one.

Definition. An algorithm is called **normal** if it uses on a hypercube only one dimension of hypercube vertices at a time and uses adjacent dimensions on consecutive steps.

Definition. An algorithm is called **fully-normal** if all n dimensions are used in sequence.

2. Cube connected Cycles

It is obtained from the hypercube by replacing a hypercube vertex r with a cycle r of length n. Two hypercube nodes a and b connected by a dimension i edge are mapped to the i-th nodes of cycles a and b in the CCC.

3. Shuffle-Exchange graph

An n-dimension s-e graph has 2^n nodes and $3 \cdot 2^{n-1}$ edges. Two vertices u and v are connected by an edge: (1) if u and v differ in the last bit (**exchange** edge) or (2) u is a left or right cyclic shift of v (**shuffle** edge).

An s-e graph is obtained from the hypercube by deleting all but dimension i and adding shuffle edges.

4. de-Bruijn graph

An n-dimension de-Bruijn graph has 2^n nodes and 2^{n+1} directed edges. Vertex $u = u_1 \dots u_n$ is connected by a 0 labeled (if we consider the labeled case) edge to $u_2 \dots u_n 0$ and by an 1 labeled edge to $u_2 \dots u_n 1$. The graph is directed and labeled. In-degree(u)=2 and Out-degree(u)=2 as well.

An n-dim de-bruijn graph is obtained from an $(n+1)$-dim s-e graph by contracting all the exchange edges.

Definition. A de-Bruijn sequence of length 2^r is a string of 2^r bits so that every substring of r bits appears once including wrap arounds.

From an $n-1$-dim de-bruijn graph a 2^n long de-Bruijn sequence is obtained by contracting all the exchange edges.

FFT- Introduction

Let w_n be the n-th primitive root of unity, ie. $w_n^n = 1$ and $w_n^j \neq 1$ for $0 < j < n$. Let F_n be an $n \times n$ matrix such that its (i, j)-th entry is w_n^{ij}. Let $\vec{y} = F\vec{x}$, for two vector $\vec{x}, \vec{y}$. Vector $\vec{y}$ can be computed in $O(\lg n)$ steps on a 2d-MOT with $O(n^2)$ processors. We are going to present an algorithm that runs on the same time but uses only $O(n)$ processors.

Let $\vec{u}, \vec{v}$ be two vectors defined as follows.

$$\vec{u} = F_{n/2} \begin{pmatrix} x_0 \\ x_2 \\ \dots \\ x_{n-2} \end{pmatrix}, \vec{v} = F_{n/2} \begin{pmatrix} x_1 \\ x_3 \\ \dots \\ x_{n-1} \end{pmatrix},$$

where $w_{n/2} = w_n^2$. Once $\vec{u}, \vec{v}$ are known, $\vec{y}$ can be obtained as follows.

$$y_i = \begin{cases} u_i + w_n^i v_i & 0 \leq i < n/2 \\ u_{i-n/2} + w_n^i v_{i-n/2} & n/2 \leq i < n \end{cases}$$

that is, there exists a divide-and-conquer approach to finding $\vec{y}$. The computation of all y_i in parallel requires one parallel step that is.

$$T(n) = T(n/2) + 1 \rightarrow T(n) = O(\lg n).$$

FFT on the butterfly

The FFT network was introduce for the purpose of performing FFTs. Each parallel step of an n point FFT is carried out on one level of a $\lg n$-dimensional butterfly.

For the sake of definition, let $bin(i)$ be the binary representation of decimal integer i representing a row of a butterlfly. Let $rev(i)$ be the reverse of $bin(i)$.

In the butterfly network that computes $\vec{y}$, input $\vec{x}$ is input through the vertices of level $\lg n$ and output is obtained through the vertices of level 0. This reverse butterfly network is depicted in the figure below so that level $\lg n$ appears on the left and level 0 on the right.

The proof is going to be constructive and will use induction. As in level 0 vertices y_i are computed, the inputs in level 1 will be u_i in the top $n/2$ rows andd v_i in the bottom $n/2$ rows. As $y_i = u_i + w_n^i v_i$ and $y_{i+n/2} = u_i + w_n^{i+n/2} v_i = u_i - w_n^i v_i$, the computations performed along the cross and level edges are obvious.

reverted Butterfly

Let us assume by induction (inductive hypothesis) that a $\lg n$ level buttefly computes $\vec{y}$ on $n/2$ inputs. Then a $\lg n + 1$-level butterfly on n input, consists of two $\lg n$-level butterflies if level 0 nodes are removed. Let $u_0, u_1, \ldots u_{n/2-1}$, $v_0, v_1, \ldots v_{n/2-1}$ be the outputs of the two butterflies in the inductive step. The operation (related to level 0 and 1) discussed in the previous paragraph shows that y_i is going to be output in the i-th row of level 0 of the $\lg n + 1$-level butterfly.

FFT continued

Question Given that the top $n/2$ butterfly computes $\vec{u}$ and the bottom $\vec{v}$ what are the input x_j's to each such butterfly?

Answer Vector $\vec{u}$ requires $x_0, \ldots x_{n/2-1}$ as inputs. These are the even-numbered x's i.e. those x_j that have 0 in the lsb position of $bin(j)$. Therefore the input to the top butterfly (of the two that resulted after the deletion of level 0 vertices) are the even indexed x's. This 0 lsb is however the value of the msb of the top $n/2$ rows of the n butterfly (the top rows have msb(bin(i))=0). Similarly, the i-th lsb of x must be equal to the i-th msb of the row this x is input. Unfolding the recursion we get that x_i is input to row $rev(i)$ of the butterfly.

A node $\langle a, j \rangle$ performs the following computation.

- the high-numbered node of previous level (i.e. the one with 1 in $j+1$-st bit position of a) is multiplied with $w^i_{n/2^j} = w^{i2^j}_n$.
- the low-numbered node of previous level (i.e. the one with 0 in $j+1$-st bit position of a) is then added to this result.

Remark. $w^{i2^j}_n$ is a function of row i and level j. Therefore all the values can be precomputed in time $O(\lg n)$ and stored locally.

Question. Hypercube implementation?

Odd-even merge-sort

Batcher's odd-even merge sort algorithm was developed in the 1960's and can be implemented in an $O(n)$ size network to sort n keys in time $O\lg^2 n)$.

Parallel Merge-Sort
Ordinary merge sort

(1) Divide problem into two halves.

(2) Sort two halves recursively.

(3) Merge two sorted halves.

Conclusion Levels of recursion is $\lg n$.
A parallel merge-sort algorithm would work the same way. Its running time would be as follows.

$$T_{sort}(n) = T_{sort}(n/2) + T_{merge}(n/2)$$

where $T_{sort}(n)$ is the time to sort n keys in parallel with n processors, and $T_{merge}(n)$ is the time to merge 2 lists of n keys each in parallel using $2n$ processors

Parallel Merge

Merge two sorted halves each of size $m(= n/2, n/4, n/8, \ldots)$.

Merge m pairs of sublists of size 1 into m pairs of size 2.

Merge $m/2$ sublists of size 2 into ones of size 4.

Merge 4 sublists of size $m/2$ into ones of size m.

Merge 2 sublists of size m into ones of size $2m$.

Remark. Merging is done recursively in $O(\lg m)$ iterations. The merging algorithm that merges two sublists of size m each is given below.

Parallel Merge algorithm description

Let $A = a_0 \ldots a_{m-1}$ and $B = a_0 \ldots a_{m-1}$, where A, B are two sorted sequences to be merged.

Step 1. Form pairs $even(A) = a_0 a_2 \ldots a_{m-2}$ and $odd(A) = a_1 a_3 \ldots a_{m-1}$, $even(B) = b_0 b_2 \ldots b_{m-2}$ and $odd(B) = b_1 b_3 \ldots b_{m-1}$.

Step 2. Recursively merge $even(A)$ with $odd(B)$. Similarly, merge $odd(A)$ and $even(B)$. Let $C = sort(even(A) + odd(B))$ and $D = sort(odd(A) + even(B))$. Let the two sequences be of size c and d respectively.

Question In order to merge two m-long sequences A and B we ended up merging two sequences C and D of the same size!!!! Is there any benefit in performing this apparently needless task??

Step 3. Interlace C and D into list L_{pre}.

$$L_{pre} = c_0 d_0 c_1 d_1 \ldots$$

Then compare c_i and d_i.

Step 4. Resulting sequence L is sorted.

Example Let $A = 1\ 5\ 7\ 8$ and $B = 2\ 3\ 4\ 6$.

1. Then $even(A) = 1\ 7$ and $odd(A) = 5\ 8$. Also $even(B) = 2\ 4$ and $odd(A) = 3\ 6$.
2. $C = 1\ 3\ 6\ 7$ and $D = 2\ 4\ 5\ 8$ and
3. $L_{pre} = 1\ 2\ 3\ 4\ 6\ 5\ 7\ 8$.
4. $L = 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8$.

Odd-even merge sort

Theorem Odd-even merge-sort works as claimed.

Proof. A proof that odd-even merge-sort works utilizes the 0-1 sorting lemma. By induction let us assume that o-e merge-sort works for sizes less than or equal to $n-1$.

Therefore in order to sort n keys, we split them into 2 halves of size $n/2$ each. By the inductive hypothesis, o-e merge-sort sorts independently the two halves. It remains to be shown that the merging algorithm so described merges the two sorted sequences and the theorem is proved.

Lemma Merging algorithm works as claimed.

Proof of Lemma We use the 0-1 Sorting Lemma. Let A consists of a 0's and $m-a$ 1's and B of b 0's and $m-b$ 1's.

Then after step 1, $odd(A)$ has $\lfloor a/2 \rfloor$ 0's and $even(A)$ has $\lceil a/2 \rceil$ 0's. Similarly, $odd(B)$ has $\lfloor b/2 \rfloor$ 0's and $even(B)$ has $\lceil b/2 \rceil$ 0's. Then, after step 2, C has $c' = \lceil a/2 \rceil + \lfloor b/2 \rfloor$ 0's and D has $d' = \lceil b/2 \rceil + \lfloor a/2 \rfloor$ 0's, where $|c' - d'| \leq 1$.

Then, after step 3, we get

a. If $c = d$ or $c = d + 1$,

$$L = L_{pre} \quad \underbrace{0 \quad 0 \quad \ldots \quad 0}_{c+d \text{ 0's}} \quad 1 \quad 1 \quad 1 \quad 1$$

b. If $c = d - 1$,

$$L = L_{pre} \quad \underbrace{0 \quad 0 \quad \ldots \quad 0}_{2c \text{ 0's}} \quad \underbrace{1 \quad 0}_{\text{flip}} \quad 1 \quad 1$$

This completes the proof. Time required for the algorithm.

$$T_{merge}(n) = T_{merge}(n/2) + 1 \rightarrow T_{merge}(n) = O(\lg n).$$

$$T_{sort}(n) = T_{sort}(n/2) + T_{merge}(n/2) = T_{sort}(n/2) + O(\lg n) = O(\lg^2 n).$$

Merging Phase

Proof constructive by induction.

Base case: A 2-input butterfly merges two keys (easy by inspection).

Inductive assumption: an n-input butterfly merges two sorted sequences of size $n/2$ each. A merging operation starts from the left (level $\lg n$) proceeds to the right (level 0) and returns to the left of a reverted butterfly.

Let the butterfly be in reverse order ($\lg n$-level leftmost level, 0-level rightmost level). Let $a_0, a_1, \ldots$ be the input to the top $n/2$ rows and $b_0, b_1, \ldots$ be the input to the bottom $n/2$ rows The top half uses the level edges to transfer input keys to the next level. The bottom half uses the cross edges and output is mixed. We view the two halves as two butterflies with level $\lg n$ deleted. The two butterflies of size $n/2$ each have their inputs intermixed. The topmost butterfly has inputs $even(A)$ and $odd(B)$ and the bottomost butterfly has inputs $odd(A)$ and $even(B)$.

The two separate butterflies merge by induction their inputs.

After the merging step.

At level $\lg n - 1$, after the merging had been performed separately in the topmost and bottomost butterflies, let the outputs be $c_0, c_1, \ldots$ and $d_0, d_1, \ldots$ already intermixed. The crossed edges are utilized to perform the pairwise comparisons in step 3 of the algorithm.

Example of reverse butterflies (only two last levels shown).

Implementation on the butterfly of the sorting algorithm

Sorting phase In order to sort n keys.

We split them into $n/2$ pairs of sublists of size 1 and 1 and merge them independently using only the last two levels of the butterfly (ie. 1 1 indicates phases not levels). The first two levels can be considered as a set of $n/2$ 2-row butterflies.

We split the derived $n/2$ sublists of size 2 into $n/4$ pairs of sublists each of size 2 and merge them independently using the last three levels of the butterfly (ie 1 2 2 1).

We split the derived $n/4$ sublists of size 4 into $n/8$ pairs of sublists each of size 4 and merge them independently using the last four levels of the butterfly (ie 1 2 3 3 2 1).

In general in round i we merge $n/2^i$ pairs of sublists of size 2^{i-1} each that requires $2i$ steps. Total parallel time is

$$\sum_{i=1}^{\lg n} 2i = \lg n(\lg n + 1)$$

The algorithm can be modifies to work in hal as many rounds by observing that the first $\lg n$ steps move keys to the right for the purpose of mixing them than actual work is performed.