Hypercubes and Hypercubic Networks

Routing on fixed connection networks

For some review on the definitions related to Fixed Connection Networks, we refer to Subject 1, pages 12-16.

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.

```
x - x - x - x
            |
x - x - x - x  An embedding of a 16-proc linear array into a 4x4 2d-array
|              with dilation 1.
x - x - x - x
            |
x - x - x - x
```

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

```
              (0,k)
               x
             / | \             All the nodes in the square  are within distance
           /   |  \            k from the origin (0,0). Note distance is
(-k,0)   /     |   \ (k,0)     Manhattan distance horizontal-vertical steps
       x------o----x--         not Euclidean distance.
         \     |   /           Number of nodes in each quarter is
           \   |  /            1+2+ ... + k = k(k+1)/2 excluding (0,0)
            \  | /             Total is 4k(k+1)/2 + 1 <<is (0,0) contribution.
              \x/(0,-k)
```

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

Hypercube is one of the most versatile networks.

It can simulate one step of an $O(n)$-cell array, binary tree in $O(1)$ time. 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 $u = u_1 u_2 \dots u_i \dots u_n$ 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 $u^i = u_1 u_2 \dots \bar{u}_i \dots u_n$ 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 $a_1 a_2 x \dots y a_1$, $a_1' a_2' x' \dots y' a_1'$ be the two paths/cycles. Then, construct the following cycle $a_1 a_1' y' \dots x' a_2' a_2 x \dots y a_1$.

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 $2^{d_1+\ldots+d_r}$-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. A complete binary tree of height n has a dilation $n/2$ (ceiling of) embedding in a 2d-array.

```
x     x
|     |   A binary tree with 4 leaves embedded into a 2d-array.
x--x--x
|     |
x     x
```

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.

```
A doubly rooted tree             Another doubly-rooted tree
0-subcube                        1-subcube
    0100       0000               1000       1010
      x---------x                  x---------x
      |         |                  |         |
      |         |                  |         |
      |         |                  |         |
      |         |                  |         |
      |         |                  |         |
      |         |                  |         |
      x         x0010         1100 x         x
    x   x     x   x              x   x     x   x
   x x x x   x x x x            x x x x   x x x x

                 <  new roots     >
    0100       0000               1000       1010
      x---------x------------------x---------x
      |  \                      /            |
      |     \                /               |  Hypercube of twice the
      |         \         /                  |  original size
      |             \  /                     |
      |             /   \                    |
      |          /           \               |
      x         x0010         1100\x         x
    x   x     x   x              x   x     x   x
   x x x x   x x x x            x x x x   x x x x
```

Corollary 1 $N/2$ vertex complete binary tree can be embedded into an N 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 $u = u_1 \ldots u_n$ is connected by a 0 labeled (if we consider the labeled case) edge to $u_2 \ldots u_n 0$ and by an 1 labeled edge to $u_2 \ldots u_n 1$. 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.

Overview

The PRAM is a versatile model for parallel computing and easy to design algorithms for; there are many algorithms that have been implemented on the PRAM. The following question then arises for the PRAM.

Question. Can one simulate the PRAM on a fixed connection network?

One then needs to solve the problem of how processors get the data they actually need. In order for processors to communicate, they send messages to each other. Memory is attached to each processor; alternatively, a node of the interconnection network is either a processor unit or a memory unit.

Since a processor/node is connected with few others (the degree of the network is substantially less than n the number of processors) a message needs to go through other processors to reach its destination. The hardware responsible for this dispatch of messages is called the ”router”. It implements a ”routing algorithm” which is a set of rules governing the transmission/receipt of messages. The task is to route messages efficiently (*i.e. in time proportional to the diameter of the network*).

Problems/Issues/Terminology

1. Topology.
   - Hypercube
   - 2d-array
   - Butterfly
2. Processors
   - Synchronous
   - Asynchronous

Taxonomy

3. Message Flow Control
   - Store-and-Forward (breaks messages into packets, sends packets individually. A packet is received before forwarded. One packet per link).
   - Circuit Switching (i.e. telephones). Establish link, send messages along the link from source to destination, release link.
   - Mixed (Wormhole routing). Like c-s it first establishes a link. Like s-and-f breaks messages into flits sends them through the link in a pipelined fashion and then releases the link.
4. Routing Protocol
   - Minimal (take shortest path) vs non-minimal.
   - Oblivious vs non-oblivious. Route depends on source and destination only.
5. Queueing Discipline
   - FIFO
   - Farthest First (Closest First).
   - Deflection (no queue at all).
6. Type of routing
   - Static. Packets fixed before routing.
   - Dynamic. Dynamically generated at say some rate.

Taxonomy II

7. Performance measures

   - Deadlock: no message moves. Detection and avoidance problems.
   - Queue-size
   - Fault-tolerance
   - Total routing time.
   - Worst case routing time: Performance under extreme communication patterns.
   - Average case routing time: Destination uniformly at random chosen among possible ones.

8. Routing Algorithm

   - Deterministic: fixed.
   - Randomized: random choices allowed.

Communication Patterns

One-to-One communication One processor sends one message to another processor.

One-to-One routing Each processor sends at most one message and each processor receives at most one message. One-to-one communication is an instance of an one-to-one routing.

Many-to-One routing Every processor sends one message and one processor may receive more than one (i.e. many messages to single destination).

One-to-Many routing Every processor sends many messages and each processor receives at most one message.

Permutation routing is a special case where each processor sends exactly one message and receives exactly one message.

h**-relation** Each processor sends at most h messages and receives at most h messages. Alternatively, each processor sends at most h words and receives at most h words of information. A p-relation, where p is the number of processors is also known as total relation.

One-to-All broadcasting or **broadcasting**. One processor sends the same message to every other processor.

All-to-one reduction. Each processor sends one piece of information to a designated processor which will combine the results. Think of parallel-sum as a form of an all-to-one reduction.

All-to-All broadcasting Each processor sends a message to every other processor; the messages sent by various processors may be different. A p-relation is realized since each processor receives at most p messages and may send at most p as well, where p is the number of processors available.

One-to-All personalized communication. One processor sends a unique message to every other processor. It is different from broadcasting as one processor initially holds p different messages.

All-to-All personalized communication. Each processor sends a distinct message to every other processor. This operations is also known as **total-exchange**.

EREW PRAM Simulation

Claim 1a. Permutation routing is no harder than sorting on the butterfly.

The result holds on other hypercubic networks as well.

Proof. Reduce packet routing to sorting.

For each packet, label each packet with id of destination processor. Sort packets with respect to this label. For a permutation routing problem, a packet with label i, after sorting will reside in processor i, as all processors will receive exactly one packet.

Thm 1 n memory requests of an EREW PRAM can be simulated on an n-input butterfly in time proportional to Routing_Time.

Thm 2 n memory requests of an CRCW PRAM can be simulated on an n processor EREW PRAM in Sorting_Time (n).

Thm 3 n memory requests of an CRCW PRAM can be simulated on an n-input butterfly in time proportional to Sorting_Time plus Routing_Time.

Greedy Routing Algorithm: Worst Case Running Time

A greedy **routing** algorithm is an algorithm that follows a shortest path.

Thm. Given any routing problem on an N-row butterfly for which at most one packets originates from each level 0 processor and at most 1 packet ends in a $\lg N$-level processor, the greedy algorithm will route all the packets to their destination in $O(\sqrt{N})$ time.

Proof. Let u be a node of row 0 and level i of the butterfly. Exactly 2^{i-1} input processors of level 0 lead to u and $2^{\lg N-i}$ output processors of level $\lg N$ are reachable from u. Let n_i be the number of greedy paths that traverse u. It is $n_i \leq 2^i$ and $n_i \leq 2^{\lg N-i}$. Therefore the total running time of the greedy algorithm will be

$$\begin{aligned} Totaltime &= \sum_i delays\ on\ each\ node\ of\ level i of\ the\ path\ from\ source\ to\ destination \\ &= \sum_i (n_i - 1) = \sum_{i=1}^{\lg N/2} 2^i + \sum_{i=\lg N/2+1}^{\lg N/2} 2^{\lg N-i} - \lg N = O(\sqrt{N}). \end{aligned}$$

The maximum queue size required at any processor is also $\Theta(\sqrt{N})$ (the case for $i = \lg N/2$).

Thm. There exists a routing problem that requires at least $\Omega(\sqrt{N})$ steps.

Examples off such problems are the following ones.
A transposition is a permutation of the form: $\pi(u_1 \ldots u_{\lg N/2} u_{\lg N/2+1} \ldots u_{\lg N}) = u_{\lg N/2+1} \ldots u_{\lg N} u_1 \ldots u_{\lg N/2}$.
A bit-reversal is a permutation of the form: $\pi(u_1 \ldots u_{\lg N}) = u_{\lg N} \ldots u_1$.

Such permutations are usually realized while designing real machines, while various performance tests take place to stress the router and to study the behavior of the processor interconnection network.

Greedy routing - Introduction

Theorem L1 . Routing an 1-relation requires at most $N-1$ steps on an N-cell linear array (bi-directional links are available).

Proof.

- There is never contention on using the same edge in the same direction.
- Whenever a packet wants to move it moves.
- At no time are 2 packets at the same processor (unless one is already on its own destination).
- packet reaches destination at d steps if d is distance between source and destination.
- packet moves unless it has reached its destination.

Theorem L2. Routing an 1-relation on a $\sqrt{N} \times \sqrt{N}$ 2d-array requires $2\sqrt{N}-2$ steps.

Proof. Use greedy algorithm. Move a packet from (s_1, s_2) to (d_1, d_2) as follows.

1. Correct column first, i.e. move from (s_1, s_2) to (s_1, d_2).
2. Correct row then, i.e. move from (s_1, d_2) to (d_1, d_2).

Analysis of Step 1. Step 1. is the time for linear array ($\sqrt{N}$ independent routing problems). Routing is performed in $\sqrt{N}-1$ steps (at most).

Analysis of Step 2. Step 2. is not symmetric; columns packets may pile up at a processor. Each processor has more than one source packets. **However each processor** is the destination of ONE AND ONLY ONE packet.

Use the following **Rule L2.** Give priority to packets going farthest away.

Lemma 2. Under Rule L2, step 2 requires $\sqrt{N}-1$ steps.

We prove Lemma 2 through Theorem 3. As a conclusion the routing can be realized in $2\sqrt{N}-2$ parallel steps.

Greedy routing

Thm 3. Consider an N-cell linear array each cell having an arbitrary number of packets but each processor cell receiving at most one. If **Rule L2** is used for arbitration, routing takes $N-1$ parallel steps.

Proof. Consider only a rightward movement of packets (such a movement does not interfere with a leftward movement). A similar argument would hold for a leftward movement to complete the proof.

Let us fix for the remainder of the proof $i \leq N$ and only consider packets destined for the i rightmost processors.

- The packets destined for the righmost i processors are called P-packets (for priority). All remaining are nP-packets (non-priority).
- P-packets have priority over nP-packets.
- There are at most i packets destined for the i rightmost processors.
- If two P-packets are at the same processor, the one moves that goes farthest to the right (Rule L2).

Consider rightmost P-packet p_1 at the start of the algorithm (tiebreaker is Rule L2).

- It cannot be delayed by nP- or P- packets and moves right and reaches rightmost i nodes in at most $N-i$ steps.

Consider second rightmost P-packet p_2 under Rule L2.

- It can only be delayed one step by p_1 and by no other packet. Thus it will reach its destination by $N-i+1$ step.

Similarly, the i-th rightmost P-packet will reach its destination in $N-i+i-1 = N-1$ steps.
There is nothing specific about i. Result holds for all i. This completes the proof.

Randomized and Deterministic - Random routing and the average case

What is an input instance? Packets with destinations.
Congestion c across an edge or through a vertex is the number of packets that travel along the edge or through the vertex.

We shall use the term "with high probability" (or "w.h.p.") to mean "with probability at least $1 - 1/N^c$", where c is a positive constant and N is some problem related parameter (eg. rows of a butterfly, nodes of a hypercube, etc). If a claims holds with such a probability it means that if we repeat an experiment N^c times, the experiment will fail on the average once.

Definition 1 *An h-relation is the routing problem where each processor sends or receives at most h messages.*

A permutation is an instance of an 1-relation.

Definition 2 **Deterministic routing** *is realized by an algorithm that makes deterministic choices (ie for the same routing problem it always acts the same).*

Definition 3 **Randomized routing** *is realized by an algorithm that internally makes random choices (ie for a routing problem it may act one way once and another way another time). An algorithm that makes random choices is called a randomized algorithm.*

Definition 4 **Random routing** *(as opposed to randomized routing) is the instance of deterministic routing where the destinations of packets are drawn uniformly at random from the set of all possible destinations.*

Under random routing we are interested in the average case performance of the deterministic routing algorithm.

Average case analysis of congestion

Theorem R1. Given an N-row butterfly, with a packets per level-0 processor, in the average case (random routing) of the greedy algorithm, the congestion at every node will be less than $C = \Theta(a) + o(\lg N)$, with high probability. With high probability, the running time T of the algorithm is

$$T = O(C) + \lg N + o(\lg N).$$

Proof of congestion bound. Consider a level i node u. It is reachable from 2^i nodes of level 0 and therefore it is the intermediate node of $a2^i$ input packets if these packets are destined to any of the $2^{\lg N - i}$ nodes of level $\lg N$ that are reachable from this node.

For a given packet that may go through u, the probability that one of the $2^{\lg N - i}$ destination nodes is chosen among $2^{\lg N} = N$ is $2^{\lg N - i}/2^{\lg N} = 1/2^i$.

The number of packets that may go through u is $a2^i$.

Therefore the probability that at least r of these packets cross u is given by the following expression.

$$Pr(\textit{at least } r \textit{ crossing } u) \;=\; \binom{a2^i}{r}(2^{-i})^r \leq (\frac{ae}{r})^r$$

$$Pr(\textit{at least } r \textit{ packets crossing at least one of } N \lg N \textit{ nodes}) \;\leq\; (N \lg N)(\frac{ae}{r})^r = P$$

We distinguish two cases

Case 1: If $a \geq \lg N/2$ then choose $r = 2ea = \Theta(a)$. Then, $P \leq 1/N^{3/2}$.

Case 2: If $a \leq \lg N/2$ then choose $r = 2e \lg N/ \lg(\lg(N/a))$. Then, $P \leq 1/N^{3/2}$.

Therefore, the probability of success (ie. node congestion at most r on EVERY NODE) is given by the following expression

$$Pr(\textit{at most } r \textit{ packets crossing ANY vertex of } N \lg N) = 1 - Pr(\textit{at least } r \textit{ packets crossing some node}) \geq 1 - 1/N^{3/2}.$$

Average case analysis of routing time

For the sake of an example if $N = 100$ this probability of success is at least $1 - 1/1000 = 99.9\%$. If $N = 10000$, it is at least 99.9999%.

The bound on congestion C in the Theorem covers both cases and is larger than the value of C established in each case.

The implicit assumption made is that the a packets per processor are initially ordered, i.e. we can refer to the 1st, 2nd, $\ldots$, a-th packet of some input (level 0) row i processor of the butterfly.

The theorem above provides a bound on node congestion only. We can turn it into a bound on the running time T if we first explain how the greedy algorithm resolves the case where two packets attempt to exit a node through the same link simultaneously. We resolve this case by using a random-rank protocol, that is, each packet is assigned a random priority key from an interval ($[1, O(a + \lg N)]$) and the packet with the minimum key crosses the link first. In case, and it is likely, that two packets have the same priority key, we assume the one with the lowest priority is the one originating from a lower indexed row. In case of a further tie, we use as a priority key the initial ordering of the packets per processor. One can prove an upper bound for the running time of the greedy algorithm like the one stated in Theorem R1.

Randomized routing

Although the average case analysis of the random routing guarantees, almost surely, acceptable performance on the average case it fails however to assure such performance in the worst case. In practice, there are input that suffer from much worse performance than the average one. Such input problems like transposition, bit-reversal are observed quite often in practice as the routing patterns of various problems. Therefore the guarantees offered by random routing (or average case arguments) are not very strong.

We are interested in describing the behavior of a routing algorithm under worst case conditions. For this we are going to introduce randomized routing. Under randomized routing, no matter what the input problem is (eg a transposition or bit-reversal permutation or the "average" routing instance) the running time of the algorithm will be within the claimed performance. I.e randomized algorithm works on any problem, EVEN the bad ones. From time to time, it may fail however to exhibit the claimed performance. This however would occur very rarely (i.e with very small probability) that is, on the average, perhaps once (one routing) in N^c runs of the algorithm. This would mean not that the routing instance was bad but that the choices made by the random number generator of the randomized algorithm were poor. One solution to such a case would be to rerun the algorithm on the same input instance by resetting the random number generator to some new seed value. We need only rerun the algorithm a couple of times before success is observed.

The only disadvantage of using a randomized algorithm is that its running time is twice as much as that of the overage case of the deterministic algorithm.

The question that arises is how we can turn a random routing problem using a deterministic (and greedy) algorithm into a randomized routing algorithm.

We are going to use **Valiant's Paradigm**:

REDUCE RANDOMIZED ROUTING TO TWO RANDOM ROUTING PROBLEMS

Valiant's Paradigm

Routing Problem For any packet i, let the source and destination be described by the pair $(s(i), d(i))$. Realize routing described by the set of pairs $(s(i), d(i))$, $\forall i$.

Valiant's Randomized Algorithm (1982)

Step 1. For every packet i, choose uniformly at random an intermediate destination $r(i)$.

Step 2. Route packet i from source $s(i)$ to intermediate destination $r(i)$ i.e. realize routing $(s(i), r(i))$.

Step 3. Route packet i from intermediate destination $r(i)$ to final destination $d(i)$ i.e. realize routing $(r(i), d(i))$.

The algorithm is randomized because in Step 1 random choices are made. Step 2 by itself is an instance of a random routing problem whose performance on the butterfly is well studied (Theorems 1 and 2). Step 3 is a reverse (mirror image) of a random routing problem and therefore its running time is that of Step2 as well.

Overall, with high probability,

Time (randomized routing) = 2 T(random routing).

The following theorem can then be derived.

Theorem 0.1 *Given an N-row butterfly, with a packets per level-0 processor, there exists a randomized routing algorithm whose running time is, with high probability,*

$$T' = O(a) + 2\lg N + o(\lg N).$$

Note that the congestion in this algorithm can be quite high $C = \Theta(a)$. In fact, we can route on the butterfly using constant-sized queues by using Ranade's algorithm.

Theorem 0.2 *Given an N-row butterfly, with a packets per level-0 processor, there exists a randomized routing algorithm whose running time is, with high probability,*

$$T' = O(a) + \lg N + o(\lg N),$$

The queues at each processor are of maximum size $Q \geq 1$.

Note that maximum queue size per processor is independent of congestion C, whereas in the previous algorithms Q could grow as high as C. The constants hidden in the $O(.)$ and $o(.)$ notations are, however, larger.

Random routing provides good performance on the average routing instance (packet destinations). It may fail, however, on certain worst case instances (eg. transposition, bit-reversal, etc), that is, its running time on such instance would be much worse ($\Theta(\sqrt{N})$ for the case of certain permutations on the butterfly). Unfortunately, such worst case instances are quite often in practice. On the other hand, a randomized routing algorithm will perform well no matter what the input routing instance is. Occasionally, it may fail to run within the expected time bound. In such a case we can just rerun the algorithm on the same input problem instance (but with different random numbers used). Randomized routing algorithms, however, are on the average slower than the corresponding deterministic greedy algorithm on the same input instance (by a factor of two in the examples discussed).

The results on the butterfly can be translated into similar results on the hypercube. In the later case, however, we assume that each processor can receive/send in one step packets from all its $\lg N$ links.

In a $\sqrt{N} \times \sqrt{N}$ mesh, the greedy algorithm routes an 1-relation in $2\sqrt{N} - 2$ steps. The maximum queue size can grow quite high (up to $O(\sqrt{N})$). A randomized routing algorithm will route in time $O(\sqrt{N})$ using $O(\lg N)$-size queues only. A more complicated algorithm would route in $2\sqrt{N} + O(\log N)$ steps with constant size queues.