Routing on

Fixed Connection Networks

(Arrays and Hypercubes)

Review

The PRAM is a versatile model for parallel computing, it is easy to design algorithms for and for this reason there are many algorithms that have been implemented on the PRAM. A question arises wheter one can 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 or a node of the interconnection network is either a processor unit or a memory unit. Since a processor/node is connected with few others (degree $< n$) 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 eficiently (*i.e. in time proportional to the diameter of the network*).

Problems/Issues/Terminology

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

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 network.

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.
   - 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 processors send to one).

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-rellation, 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-All broadcasting Each processor sends a message to every other processor; the messages sent by various processor 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**.

Relation to Sorting

Claim 1a. One-to-one packet 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.

Problem What if a processor does not receive a packet (i.e. not a permutation routing)?

Each processor will perform the following.
S1. If it has a message to send, it labels it with destination id. It creates a dummy message (whether it has or not a packet to send) with id the sending processor (ie. its own id).
S2. At most $2n$ messages are sorted and each processor receives at most 2 messages, where n is an upper bound on the number of messages and the number of inputs of the butterfly.
S3. Each processor checks adjacent messages (residing in adjacent processors). If a dummy and a real message coexist it deletes the dummy.
S4. n messages are then left. Sorting again sends real messages to their destination. Alternately, it can use information of dummy messages to route real messages back to origin.

Routing_Time $= O(sorting) + O(\lg n)$.

We next show how one can simulate a EREW PRAM on such a network.

EREW Pram Simulation

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

Proof. The shared memory of the PRAM is evenly distributed among the processors of the output column of the butterfly. We could have assumed that the input processors hold the memory as well. In either case one set of processors gets the memory from the other in $\lg n$ steps. The memory requests initiate from the input column processors. Request m_i originating from processor j is first resolved locally i.e. the row number of the processor of level $\lg n$ that holds it is found. Then a routing operation is issued where destination processor of a memory request m_i is the processor storing m_i. At the completion of the routing operation, if m_i involves a write it is immediately performed, if it involves a read operation the processor holding m_i reads its contents, fills the packet received with the contents of m_i and sends it back to the originating processor along the path recorded in the first phase. Total time is 2 Routing_Time.

Greedy Routing Algorithm: Worst Case Running Time

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

Thm. Given any routng problem on an N-row butterfly for which at most one packets originate 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 processor 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 machine interconnection network.

Greedy routing - Introduction

Thm. 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 packets 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.

In more complex networks, say something that looks like a Y these arguments do not work.

Example

Greedy routing

Thm. 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 the algorithm.

(1) Stage (1) is as for linear array ($\sqrt{N}$ independent routing problems). Routing is performed in $\sqrt{N} - 1$ steps (at most).

(2) For stage (2) one cannot use a symmetric argument for columns as packets may pile up at a processor.

Rule. Give priority to packets going farthest away.

Lemma 2. Under Rule stage (2) requires $\sqrt{N} - 1$ steps for a total of $2\sqrt{N} - 2$ for the both stages.

It is crucial in the proof that each processor is the destination of at most one message.

Proof of Lemma 1.

Greedy routing Continued

Thm 3. Consider a $\sqrt{N}$-cell linear array each cell having an arbitrary number of packets but each processor receiving at most one. If Rule is used, routing take $\sqrt{N}-1$ steps.

Proof. Similar to odd-even transposition sort. Consider a rightward movement of packets (such a movement does not interfer with a leftward movement; a similar argument would hold for this latter case as well).

- Fix $i \leq \sqrt{N}$ and consider packets destined to i rightmost processors $\sqrt{N}-i \ldots \sqrt{N}-1$. Call these "priority packets" ("P").
- They are never delayed by "non-priority" ("nP") packets.
  - if another "P" packet competes against such, the one moves that goes farthest.
  - if "nP" ignore it as a "P" always wins.

Consider rightmost "P" packet at the start of the algorithm (in case of a tie the one to move first ie. with highest destination).

- It cannot be delayed by "nP" packets and moves right unhindered even by "P" packets (farthest to go) and reaches rightmost i nodes in at most $\sqrt{N}-i$ steps. (Never delayed by a "P" packet.

Consider second rightmost "P" packet. (in case of a tie the one to move first ie. with highest destination).

- It can only be delayed by previous packets and only once and never elsewhere i.e. $1+\sqrt{N}-i$ steps to reach i rightmost processors.

Similarly, the i-th righmost packet requires $(i-1)+\sqrt{N}-i=\sqrt{N}-1$ steps to reach the i rightmost processors.
Set $i=\sqrt{N}$, and a $\sqrt{N}-1$ time bound is proved for the second phase.
Theorem 3 and Lemma 2 give an overall running time of $2\sqrt{N}-2$.

Congestion c acrross 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.

Theorem 1 *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*

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

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, ..., 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 1.

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 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,*

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

Note that the congestion in this algorithm can be quite high $C = \Theta(a)$.

Randomized Routing

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.