Lecture 4: Randomized routing & The probabilistic method

4.1 Routing in a parallel computer

In this section we consider a beautiful example of the power of randomization in parallel computing.

Consider a parallel machine with n processors, labeled $0, \ldots, n-1$. The processors are linked together in a network, so that they can communicate. However, not every pair of processors has a link between them, which means that some processors can only communicate via other processors. A fairly standard topology for such a network of processors is the *hypercube*. Here the number of processors, n, is a power of 2 and there is a (bidirectional) link between processors i and j if and only if the binary representation of i and j differ in exactly one bit. For example, a hypercube with $n = 8$ processors will have the following links:

$$(0,1), (0,2), (0,4), (1,3), (1,5), (2,3), (2,6), (3,7), (4,5), (4,6), (5,7), (6,7).$$

The link $(1,3)$ exists, for instance, because 001 (=1) and 011 (=3) differ only in the second bit. From now on we will consider the two opposite links of a bidirectional link as separate links. (So for $n = 8$ we would, besides the links above, also have the links $(1,0)$, $(2,0)$, etc.)

Now suppose each processor i, for $1 \leqslant i \leqslant n$, wants to send a message $mess(i)$ to some other processor $dest(i)$. We consider the case of *permutation routing*, where the set $\{dest(1), \ldots, dest(n)\}$ of destinations forms a permutation of $1, \ldots, n$. In other words, each processor i wants to send exactly one message and wants to receive exactly one message. Sending these messages is done in *rounds*. In each round, a processor can send at most one message to each of its neighbors. Each processor i has a collection of $\log n$ buffers, one for each outgoing link. We denote the buffer for the link (i,j) by $\mathcal{B}_i(j)$. Buffer $\mathcal{B}_i(j)$ will store messages that processor i needs to forward to its neighbor j, but that have to wait because i also needs to forward other messages to j. Each processor i executes the following algorithm in every round:

Algorithm *RoutingStrategy*(i)
1. ▷ **Send phase:**
2. **for** each outgoing link (i,j)
3. **do** Select a message from $\mathcal{B}_i(j)$ and send it along link (i,j).
4. ▷ **Receive phase:**
5. **for** each incoming message whose destination is not i
6. **do** Store that message in $\mathcal{B}_i(j)$, where j is the next processor on its route.

The main question now is how the routes are chosen. The goal is to do this in such a way that the total number of rounds needed before every message has arrived at its destination is as small as possible. Thus we would like the routes to be short. Moreover, we do not want too many routes to use the same links in the network because that will lead to congestion: the buffers of these congested links will have to store many messages, and consequently these messages will have to wait a long time before they are finally forwarded. This raises another issue that we still have to address: how does a processor i select from its buffers $\mathcal{B}_i(j)$ which messages to send in each round? Note that a processor i does not necessarily know the destinations $dest(j)$ of the messages of the other processors. Hence, the route-planning algorithm should be *oblivious*: each processor should determine $route(mess(i))$ based only on i and $dest(i)$, not on any $dest(j)$ for $j \neq i$.

A simple routing strategy is the following:

- Each buffer $\mathcal{B}_i(j)$ is implemented as a queue, and the incoming messages in each round are put into the queue in arbitrary order. (Note: *arbitrary* means that the order in which the messages are put in the queue does not matter, it does *not* mean that we put them into the queue in random order.)
- The route followed by each message is determined using the *bit-fixing strategy*, which is defined as follows. Suppose we have already constructed some initial portion of $route(mess(i))$ and let j be the last processor on this initial portion. To determine the next processor j' on the route we look at the bits in the binary representation of $dest(i)$ from left to right, and determine the leftmost bit b that is different from the corresponding bit of the binary representation of j. We then take j' such that its binary representation is the same as that of j, except for the bit b. (Note that because we are routing on a hypercube, the link (j, j') exists.) For example, if $n = 32$ and processor 01101 wants to send a message to 00110, then the route would be

$$01101 \rightarrow 00101 \rightarrow 00111 \rightarrow 00110.$$

This routing strategy is pretty simple and it is oblivious. Moreover, it has the advantage that a processor does not need to include the whole route for $mess(i)$ into the message header, it suffices to put $dest(i)$ into the header. (Based on $dest(i)$ and its own processor number j, the processor j knows where to send the message $mess(i)$ to.) Unfortunately, the number of rounds can be fairly large: there are sets of destinations on which this routing strategy needs $\Omega(\sqrt{n})$ rounds. In fact, one can prove that for *any* deterministic routing strategy there is a set of destinations that will require $\Omega(\sqrt{n/\log n})$ rounds. Surprisingly, using randomization one can do much better. The trick is to let each processor i first route its message to a random intermediate destination:

Algorithm *DetermineRoute*(i)
1. $r_i \leftarrow Random(0, n-1)$
2. Construct a route from processor i to processor r_i using the bit-fixing strategy, and construct a route from processor r_i to processor $dest(i)$ using the bit-fixing strategy. Let $route(mess(i))$ be the concatenation of these two routes.

Next we analyze the expected number of rounds this randomized routing algorithm needs. For simplicity we will slightly change the algorithm so that it consists of two phases: in the first phase all messages $mess(i)$ will be routed from i to r_i, and in the second phase all messages will be routed from r_i to $dest(i)$. Thus the messages wait at their intermediate destination until all messages have arrived at their intermediate destination. We will analyze the number of rounds needed in the first phase. By symmetry the same analysis holds for the second phase. From now on, we let $route(i)$ denote the route from i to r_i, as prescribed by the bit-fixing strategy. Our analysis will be based on the following lemma, which we will not prove:

Lemma 4.1 *Let S_i be the set of messages whose route uses at least one of the links in $route(i)$. Then the delay incurred by $mess(i)$ in phase 1 is at most $|S_i|$. In other words, $mess(i)$ reaches its intermediate destination r_i after at most $|route(i)| + |S_i|$ rounds, where $|route(i)|$ is the length of the path $route(i)$.*

To analyze the algorithm, we introduce indicator random variables X_{ij}:

$$X_{ij} := \begin{cases} 1 & \text{if } j \neq i \text{ and } \mathit{route}(i) \text{ and } \mathit{route}(j) \text{ share at least one link} \\ 0 & \text{otherwise} \end{cases}$$

Now fix some processor i. By the previous lemma, $\mathit{mess}(i)$ will reach r_i after at most

$$|\mathit{route}(i)| + \sum_{0 \leqslant j < n} X_{ij}$$

steps.

First we observe that $|\mathit{route}(i)|$ is equal to the number of bits in the binary representation of i that are different from the corresponding bits in the representation of r_i. Hence, $\mathrm{E}[|\mathit{route}(i)|] = (\log n)/2$.

Next we want to bound $\mathrm{E}[\sum_{0 \leqslant j < n} X_{ij}]$. To this end we use the fact that the expected length of each route is $(\log n)/2$, as observed earlier. Hence, the expected total number of links used over all routes is $(n/2)\log n$. On the other hand, the total number of links in the hypercube is $n \log n$—recall that (i, j) and (j, i) are considered different links. By symmetry, all links in the hypercube have the same expected number of routes passing through them. Since we expect to use $(n/2)\log n$ links in total and the hypercube has $n \log n$ links, the expected number of routes passing through any link is therefore $\frac{(n/2)\log n}{n \log n} = 1/2$. This implies that if we look at a single link on $\mathit{route}(i)$ then we expect less than $1/2$ other routes to use this link. Since $|\mathit{route}(i)| \leqslant \log n$, we get

$$\mathrm{E}[\sum_{0 \leqslant j < n} X_{ij}] \quad < \quad |\mathit{route}(i)|/2 \quad \leqslant \quad (\log n)/2.$$

This is good news: message $\mathit{mess}(i)$ is expected to arrive at its intermediate destination r_i within $\log n$ rounds. But we are not there yet. We want to achieve that *all* messages arrive at their destination quickly. To argue this, we need a high-probability bound on the delay of a message. To get such a bound we note that for fixed i, the random variables X_{ij} are independent. This is true because the intermediate destinations r_j are chosen independently. This means we are in the setting of independent Poisson trials: each X_{ij} is 1 or 0 with a certain probability, and the X_{ij} are independent. Hence we can use tail estimates for Poisson trials—see p.2 of the handouts on basic probability theory—to conclude that for $\delta > 0$

$$\Pr[X > (1+\delta)\mu] \leqslant \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu,$$

where $X = \sum_{0 \leqslant j < n} X_{ij}$ and $\mu = \mathrm{E}[X]$. Plugging in $\delta = \frac{3\log n}{\mu} - 1$ and using that $\mu \leqslant (\log n)/2$ (note that this implies $\delta \geqslant 5$) one can deduce that

$$\Pr[X > 3\log n] \leqslant (1/2)^{2\log n} = 1/n^2.$$

This implies that with probability at least $1 - 1/n$ *all* messages arrive with a delay of at most $3\log n$ steps—much better than the $\Omega(\sqrt{n/\log n})$ lower bound for deterministic routing strategies.

4.2 The probabilistic method

Randomization can not only be used to obtain simple and efficient algorithms, it can also be used in combinatorial proofs. This is called the *probabilistic method*. The general idea is to use one of the following two facts.

- Let X be a random variable. There is at least one elementary event for which the value X is less than or equal to $\mathrm{E}[X]$, and there is at least one elementary event for which the value of X is at least $\mathrm{E}[X]$.
- Consider a collection S of objects. If an object chosen at random from S has a certain property with probability greater than zero, then there must be an object in S with that property.

  Note: to prove the existence of an object in S with the desired property it is sufficient to prove that a random object in S has the property with positive probability. If you can even prove that a random object has the desired property with "large" probability, say at least some constant $p > 0$ that is independent of $|S|$, and you can check whether the object has the property, then you immediately obtain a Las Vegas algorithm for finding an object with the property.

We will consider two simple applications of the probabilistic method.

MaxSat. Let $x_1, \ldots, x_n$ be a set of n boolean variables. A boolean formula is a CNF formula—in other words, is in conjunctive normal form—if it has the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

where each clause C_j is the disjunction of a number of literals (a literal is a variable x_i or its negation $\overline{x_i}$). For example, the following is a CNF formula with three clauses.

$$(x_1 \vee x_3 \vee \overline{x_4}) \wedge (x_2 \vee \overline{x_3} \vee \overline{x_5}) \wedge (x_1 \vee \overline{x_5}).$$

Given a CNF formula with m clauses, the MaxSat problem is to find a truth assignment satisfying as many clauses as possible.

Instead of looking at this algorithmic question, we can also consider the combinatorial question of how many clauses one can always satisfy. The following simple example shows there are CNF formulas where we cannot satisfy more than half the clauses:

$$(x_1) \wedge (\overline{x_1}) \wedge (x_2) \wedge (\overline{x_2}) \wedge \cdots \wedge (x_{m/2}) \wedge (\overline{x_{m/2}}).$$

Can we always satisfy at least half the clauses? And what if all the clauses have more than a single variable, can we perhaps satisfy more than $m/2$ clauses? The next theorem shows that this is indeed the case.

Theorem 4.2 *Any CNF formula with m clauses such that each clause has at least k variables has a truth assignment satisfying at least $(1 - (1/2)^k)m$ clauses.*

Proof. Let $C_1 \wedge \cdots \wedge C_m$ be a CNF formula. Take a random truth assignment: For each variable x_i independently, flip a fair coin; if it comes up Heads then we set $x_i =$ **true**, otherwise we set $x_i :=$ **false**. For each clause C_j, let X_j be the indicator random variable

that is 1 if C_j is true and 0 otherwise. Then the number of satisfied clauses is $\sum_{j=1}^{m} X_j$. By linearity of expectation we have

$$\mathrm{E}[\sum_{j=1}^{m} X_j] = \sum_{j=1}^{m} \mathrm{E}[X_j] = \sum_{j=1}^{m} \Pr[\ C_j \text{ is satisfied }].$$

Now consider a clause C_j with $\ell \geqslant k$ literals. Since C_j is the disjunction of its literals, the only way in which C_j can be false is that all its literals are false. Since the truth values of the random variables are set randomly and independently, this happens with probability at most $(1/2)^\ell$. Hence,

$$\Pr[\ C_j \text{ is satisfied }] \geqslant 1 - (1/2)^\ell \geqslant 1 - (1/2)^k.$$

It follows that the expected number of satisfied clauses is at least $(1-(1/2)^k)m$. We conclude that there must be one truth assignment that satisfies at least this many clauses. □

It follows for instance that any CNF formula has a truth assignment with at least $\lceil m/2 \rceil$ satisfied clauses. Also, any 3-CNF formula—any CNF formula where every clause has three literals—has a truth assignment where at least $7m/8$ clauses are satisfied.

Independent set. Let $G = (V, E)$ be a graph. A subset $V^* \subset V$ is called an *independent set* in G if no two vertices in V^* are connected by an edge in E.

Theorem 4.3 *Let $G = (V, E)$ be a graph with $|E| > |V|/2$. Then G has an independent set of size at least $\frac{|V|^2}{4|E|}$.*

Proof. Pick a random subset $V_R \subset V$ by taking each vertex with probability p independently, where p is a parameter to be determined later. Note that the expected number of vertices in V_R is $p|V|$. Let G_R be the subgraph induced by R, that is, $G_R = (V_R, E_R)$ where $E_R = \{(u, v) : u, v \in V_R \text{ and } (u, v) \in E\}$. An edge $(u, v) \in E$ is present in E_R with probability p^2. For an edge $e = (u, v) \in E$, define X_e as the indicator random variable that is 1 if $e \in E_r$ and 0 otherwise. Using linearity of expectation we can bound the expected number of edges in E_R by

$$\mathrm{E}[|E_R|] = \mathrm{E}[\sum_{e \in E} X_e] = \sum_{e \in E} \mathrm{E}[X_e] = p^2|E|.$$

One idea would be to choose $p < 1/\sqrt{|E|}$. Then the expected number of edges in E_R is less than 1, so that V_R would be an independent set with positive probability. This would give an independent set of size $p|V| < |V|/\sqrt{|E|}$. The statement of the theorem, however, promises a much larger independent set, namely $(1/4) \cdot (|V|/\sqrt{|E|})^2$. We therefore proceed differently. Instead of taking V_R itself as the independent set, we remove for each edge $e \in E_R$ one of its endpoints from V_R. Let V^* denote the resulting set of vertices. Clearly V^* is an independent set, and the size of V^* is at least

$$\mathrm{E}[|V_R| - |E_R|] = \mathrm{E}[|V_R|] - \mathrm{E}[|E_R|] \geqslant p|V| - p^2|E|.$$

Setting $p = |V|/(2|E|)$—note that $0 < p < 1$ because $|E| > |V|/2$—gives the desired bound. □