Shared Coins

Randomization allows us to solve problems that cannot be solved by deterministic algorithms. In the last tutorial we saw a randomized algorithm that solves the consensus problem. The algorithm used a shared coin:

**shared coin** A shared coin with agreement parameter \( \rho \) is an algorithm without inputs, which has probability \( \rho \) for all the outputs to be 0, and probability \( \rho \) for all the outputs to be 1. In the rest of the executions there may not be agreement.

Given a shared coin with agreement parameter \( \rho \), the expected number of phases until all non-faulty process decide is \( 1 + \frac{1}{\rho} \), giving an \( O(n \cdot 1/\rho) \) total number of steps.

This tutorial presents shared coin algorithms with a constant agreement parameter. In these algorithms the processes flip coins until the amount of coins that were flipped reaches a certain threshold. An array of \( n \) single-writer multireader registers records the number of coins each process has flipped, and their sum. Each process decides on the value of the majority of the coin flips it reads from the array.

The goal is for the processes to read similar sets of coins, in order to agree on the same majority value. For this to happen, we bound the total number of coins that are flipped (by any process) after some process observes that the threshold was exceeded. A very simple way to guarantee this property is to have processes frequently read the array in order to detect quickly that the threshold was reached. This, however, increases the total step complexity. Therefore, we have to resolve the tradeoff between achieving a small total step complexity and a large (constant) agreement parameter.

**A shared coin with \( O(n^3) \) total work**

Algorithm 1 presents a shared coin algorithm with a constant agreement parameter and \( O(n^3) \) total step complexity. Using a shared coin algorithm with \( O(n^3) \) total step complexity and a constant agreement parameter, implies a randomized consensus algorithm with \( O(n^4) \) total step complexity.

Each process flips a coin and writes the outcome to its location in `NumFlips` and `SumCoins` arrays. Then it scans the `NumFlips` array and sums the total coin flips. If it sees at least \( n^2 \) coin flips it collects the `SumCoins` array and returns the majority.

**Claim 1** The total number of coin flips is at most \( n^2 + n + 1 \)

**Complexity** In the worst case, a single process flips \( n^2 \) coins and for each one collects the array of size \( n \), resulting in total step complexity of \( n^3 \).

**Agreement parameter** Coin flips have binomial distribution \( X \sim B(m, p) \), \( n^2 < m < n^2 + n \). When \( p = \frac{1}{2} \) we can use binomial approximation \( X \sim N(\frac{m}{2}, \frac{m}{4}) \), meaning that in \( t^2 \) coin flips the probability that the absolute sum is larger than \( t \) is constant. By the central limit theorem, \( \rho = \frac{1}{4} \).
Small improvement Collect the array every $\frac{n}{\log(n)}$ local coin flips, resulting in total step complexity of $n^2 \cdot \log(n)$.

Algorithm 1 Shared coin algorithm with $O(n^3)$ total work

Initially total-flips = 0, total-sums = 0
Shared variables: NumFlips[i] = 0, SumCoins[i] = 0

1: while total-flips < $n^2$ do
2: coin = flip(-1, 1)
3: NumFlips[i]++ = 1
4: SumCoins[i]++ = coin
5: collect (NumFlips) and sum into total-flips
6: collect (SumCoins) and sum into total-sums
7: return sign(total-sums)

A shared coin with $O(n^2)$ total work

Algorithm 2 presents a shared coin algorithm with a constant agreement parameter and $O(n^2)$ total step complexity. The novel idea of this algorithm is to utilize a multi-writer register called $Done$ that serves as a binary termination flag; it is initialized to false. A process that detects that enough coins were flipped, sets done to true. This allows a process to read the array only once in every $n$ of its local coin flips, but check the register done before each local coin flip.

Algorithm 2 Shared coin algorithm with $O(n^2)$ total work

Initially num = 0, total-flips = 0, total-sums = 0
Shared variables: Done = false, NumFlips[i] = 0, SumCoins[i] = 0

1: while Done == false do
2: coin = flip(-1, 1)
3: num++ = 1
4: NumFlips[i]++ = 1
5: NumOnes[i]++ = coin
6: if num mod n == 0 then
7: collect (NumFlips) and sum into total-flips
8: if total-flips > $n^2$ then Done = true
9: collect (SumCoins) and sum into total-sums
10: return sign(total-sums)